

Aurimas Aleksandras Nausėdas

2023



Agenda

1. Introduction
2. PyTorch Basics
3. Train A Simple Neural Net
4. PyTorch in Action
5. Data Loading, Loss & Optimizers
6. Visualisation
7. Save & Load
8. Conclusion



Terms

- PyTorch
- Tensor
- FloatTensor
- Autograd
- Variable
- Computational Graph
- Torch.nn
- Torch.nn.functional
- GPU
- GRU
- Dataloader
- Batch size
- Iterations
- Epoch
- Loss Function
- Optimizer
- Visdom
- TensorBoard

Introduction



What is PyTorch?

- It's a Python-based scientific computing package targeted at two sets of audiences¹
- A replacement for NumPy to use the power of GPUs
- A deep learning research platform that provides maximum flexibility and speed

1 – https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html



What is PyTorch?



CPU

Complicate & sequential processing



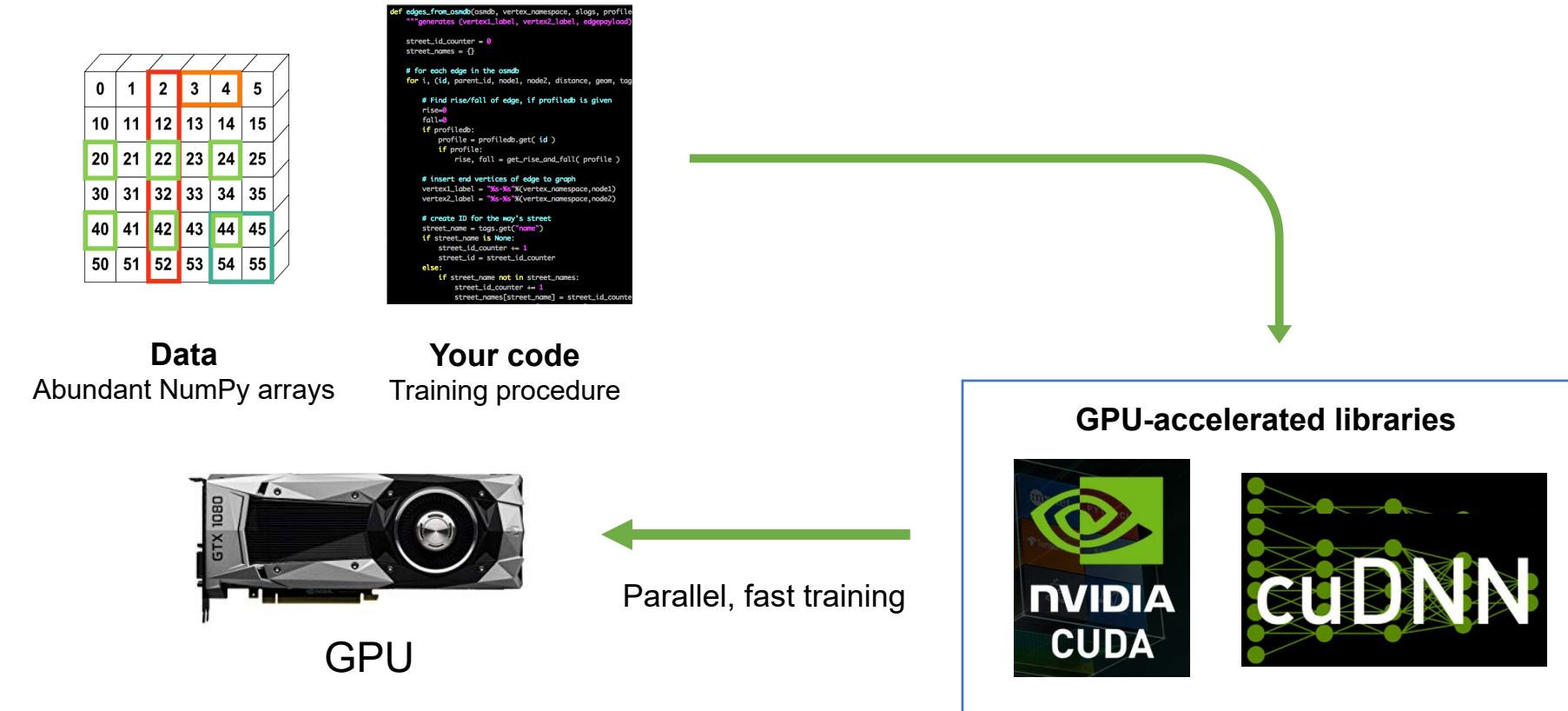
GPU

Simple but fast & parallel computing

We would like to use GPU to accelerate the training of our neural nets



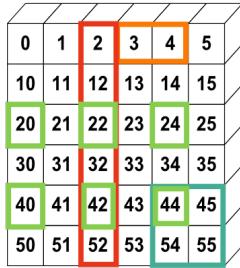
What is PyTorch?



However, CUDA is a low-level language and not available on all machines..



What is PyTorch?



```
def edges_from_omdb(omdb, vertex_namespace, slogs, profile):
    """generates (vertex_label, vertex_label, edgepayload)
    street_id_counter = 0
    street_names = {}

    # for each edge in the omdb
    for i, (id, parent_id, node1, node2, distance, grom, tag)

        # find rise/fall of edge, if profiledb is given
        rise = None
        fall = None
        if profiledb:
            profile = profiledb.get( id )
            if profile:
                rise, fall = get_rise_and_fall( profile )

        # insert end vertices of edge to graph
        vertex1_label = "%s-%s"%(vertex_namespace,node1)
        vertex2_label = "%s-%s"%(vertex_namespace,node2)

        # create ID for the way's street
        street_name = tags.get('name')
        if street_name is None:
            street_id_counter += 1
            street_id = street_id_counter
        else:
            if street_name not in street_names:
                street_id_counter += 1
                street_names[street_name] = street_id_counter

        # insert edge
        edges.append((vertex1_label, vertex2_label, edgepayload))
    return edges
```

Data
Abundant NumPy arrays

Your code
Training procedure



GPU

Parallel, fast training



Deep learning framework



TensorFlow



PYTORCH



Caffe2



GPU-accelerated libraries





Why PyTorch?

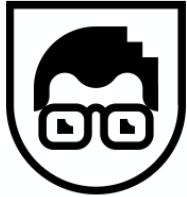


Andrej Karpathy 
@karpathy

Following

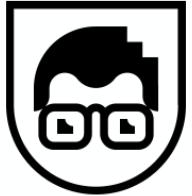
I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved.

Life is short, you need PYTORCH



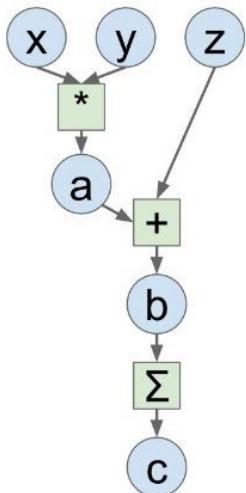
Why PyTorch?

- It is pythonic - concise, close to Python conventions
- Strong GPU support
- Autograd - automatic differentiation
- Many algorithms and components are already implemented
- Similar to NumPy



Why PyTorch?

Computation Graph



Numpy

```

import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x

```

Tensorflow

```

import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                  feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out

```

PyTorch

```

import torch

N, D = 3, 4

x = torch.rand((N, D), requires_grad=True)
y = torch.rand((N, D), requires_grad=True)
z = torch.rand((N, D), requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

```



Three Levels of Abstraction

- **Tensor:** Imperative ndarray
- **Variable:** Node in a computational graph (data, grad)
- **Module:** A neural network layer

```
In [1]: import torch  
x = torch.FloatTensor([[1.0,2.0],[3.0,4.0]])  
y = torch.FloatTensor(torch.randn(2,2))  
z = torch.randn(2,2).type(torch.FloatTensor)
```

```
In [2]: var = torch.autograd.Variable(x)
```

```
In [3]: fc = torch.nn.Linear(2,2)
```



Major Components of PyTorch

Package	Description
torch	a Tensor library like NumPy, with strong GPU support
torch.autograd	a tape based automatic differentiation library that supports all differentiable Tensor operations in torch
torch.nn	a neural networks library deeply integrated with autograd designed for maximum flexibility
torch.optim	an optimization package to be used with torch.nn with standard optimization methods such as SGD, RMSProp, LBFGS, Adam etc.
torch.utils	DataLoader, Dataset and other utility functions for convenience

PyTorch Basics



PyTorch Tensors

- Like numpy arrays, but they can run on GPU.
- No built-in notion of computational graph, or gradients, or deep learning.
- Here you see a fit of a two-layer net using PyTorch Tensors.²
- The biggest difference between a numpy array and a PyTorch Tensor is that a PyTorch Tensor can run on either CPU or GPU.²

```
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 CUDA type

```
import torch

dtype = torch.cuda.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

Create random tensors
for data and 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
```



PyTorch Tensors

Forward prop: compute predictions and loss



```
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

Backward prop:
Manually compute
gradients

```
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

Gradient descent
step on weights

```
import torch

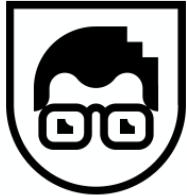
dtype = torch.FloatTensor

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
```



Initializing a Tensor

Directly from data

Tensors can be created directly from data. The data type is automatically inferred.

```
data = [[1, 2], [3, 4]]  
x_data = torch.tensor(data)
```

From a NumPy array

Tensors can be created from NumPy arrays (and vice versa - see [Bridge with NumPy](#)).

```
np_array = np.array(data)  
x_np = torch.from_numpy(np_array)
```



Attributes of a Tensor

Tensor attributes describe their shape, datatype, and the device on which they are stored.

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

Out:

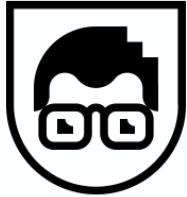
```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```



Operations on Tensors

By default, tensors are created on the CPU. We need to explicitly move tensors to the GPU using `.to` method (after checking for GPU availability). Keep in mind that copying large tensors across devices can be expensive in terms of time and memory!

```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
    tensor = tensor.to('cuda')
```



Operations on Tensors

Standard numpy-like indexing and slicing:

```
tensor = torch.ones(4, 4)
print('First row: ', tensor[0])
print('First column: ', tensor[:, 0])
print('Last column:', tensor[..., -1])
tensor[:, 1] = 0
print(tensor)
```

Out:

```
First row: tensor([1., 1., 1., 1.])
First column: tensor([1., 1., 1., 1.])
Last column: tensor([1., 1., 1., 1.])
tensor([[1., 0., 1., 1.],
       [1., 0., 1., 1.],
       [1., 0., 1., 1.],
       [1., 0., 1., 1.]])
```



Operations on Tensors

Arithmetic operations

```
# This computes the matrix multiplication between two tensors. y1, y2, y3 will have the same value
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(tensor)
torch.matmul(tensor, tensor.T, out=y3)

# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
```



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

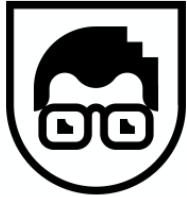
```
import torch
from torch.autograd import Variable

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()

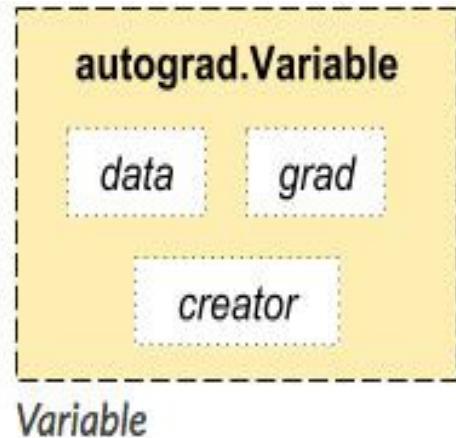
    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

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



Variable

The **autograd** package provides automatic differentiation for all operations on Tensors.



“**autograd**. Variable is the central class of the package. It wraps a Tensor, and supports nearly all of operations defined on it.

Once you finish your computation you can call `.backward()` and have all the gradients computed automatically.”



Computational Graphs

Computational Graphs

Numpy

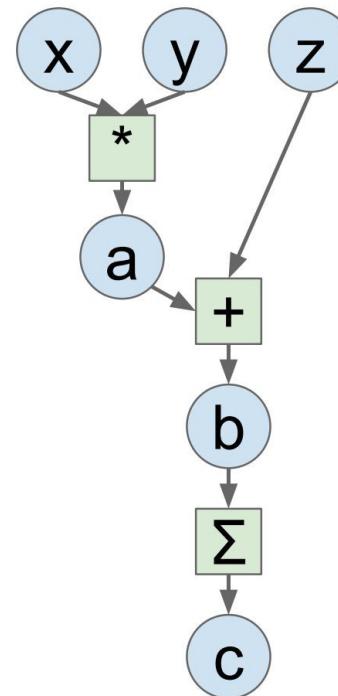
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

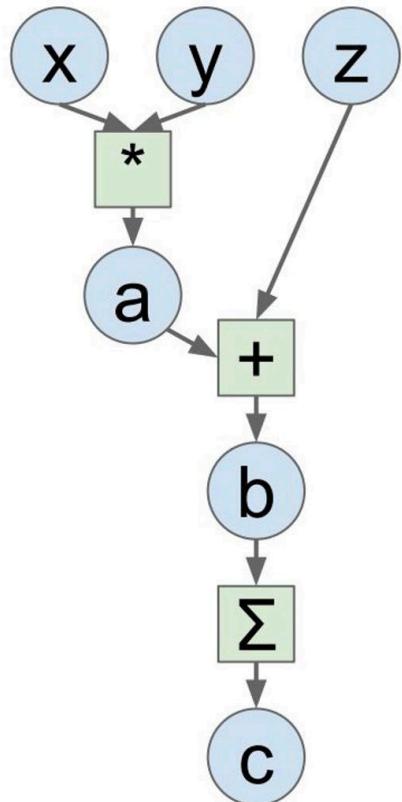
a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```





Computational Graphs



Define **Variables** to start building a computational graph

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

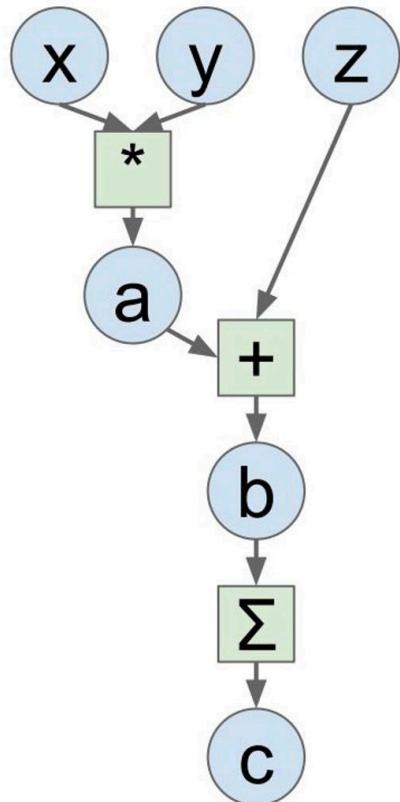
a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```



Computational Graphs



Forward prop
looks just like
numpy

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

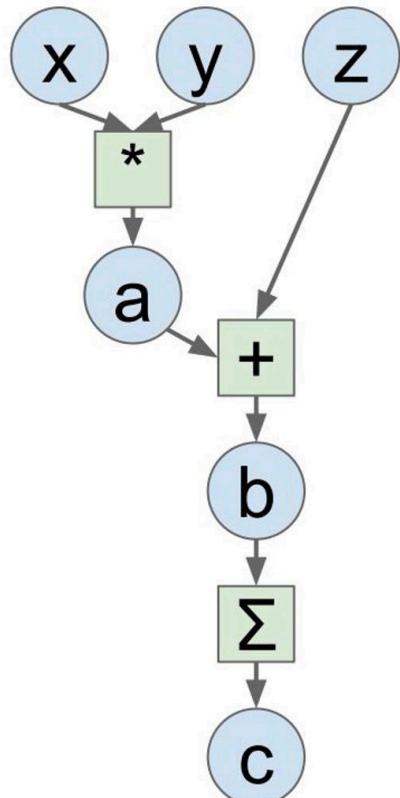
a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```



Computational Graphs



Calling `c.backward()`
computes all
gradients

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

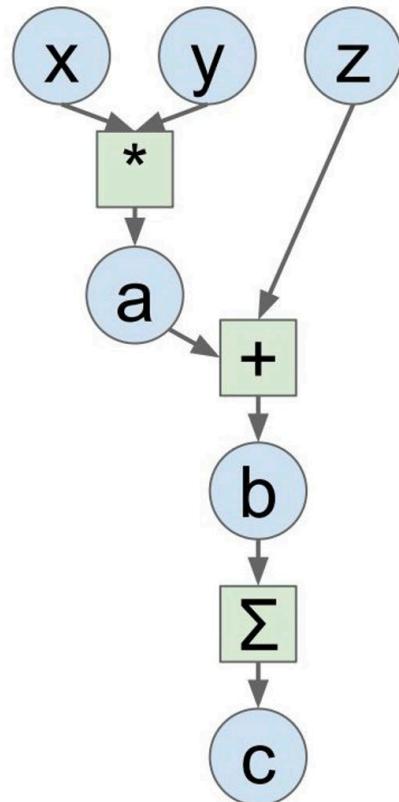
a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```



Computational Graphs



Run on GPU by
calling .cuda()

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True) red box
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```



Module

⊖ torch.nn

Parameters

⊖ Containers

⊖ Convolution Layers

Conv1d

Conv2d

Conv3d

ConvTranspose1d

ConvTranspose2d

ConvTranspose3d

⊖ torch.nn

Parameters

⊖ Containers

⊖ Convolution Layers

⊖ Pooling Layers

MaxPool1d

MaxPool2d

MaxPool3d

MaxUnpool1d

MaxUnpool2d

MaxUnpool3d

AvgPool1d

AvgPool2d

AvgPool3d

FractionalMaxPool2d

LPPool2d

AdaptiveMaxPool1d

AdaptiveMaxPool2d

AdaptiveMaxPool3d

AdaptiveAvgPool1d

AdaptiveAvgPool2d

AdaptiveAvgPool3d

⊖ Loss functions

L1Loss

MSELoss

CrossEntropyLoss

NLLLoss

PoissonNLLLoss

KLDivLoss

BCELoss

BCEWithLogitsLoss

MarginRankingLoss

HingeEmbeddingLoss

MultiLabelMarginLoss

SmoothL1Loss

SoftMarginLoss

MultiLabelSoftMarginLoss

CosineEmbeddingLoss

MultiMarginLoss

TripletMarginLoss

Other layers:
Dropout, Linear,
Normalization Layer



Module – torch.nn

Containers	Module, Sequential,ModuleList,ParameterList
Convolution Layers	Conv1d,Conv2d,Conv3d...
Recurrent Layers	RNN,LSTM,GRU,RNNCell...
Linear Layers	Linear, Bilinear
Non-linear Activations	ReLU,Sigmoid,Tanh,LeakyReLU...
Loss Functions	NLLLoss,BCELoss,CrossEntropyLoss...



Module – torch.nn.functional

Torch.nn

```
In [1]: import torch
import torch.nn as nn
from torch.autograd import Variable

relu = nn.ReLU(inplace=True)

x = Variable(torch.randn(10,128))
x = relu(x)
```

Torch.nn.functional

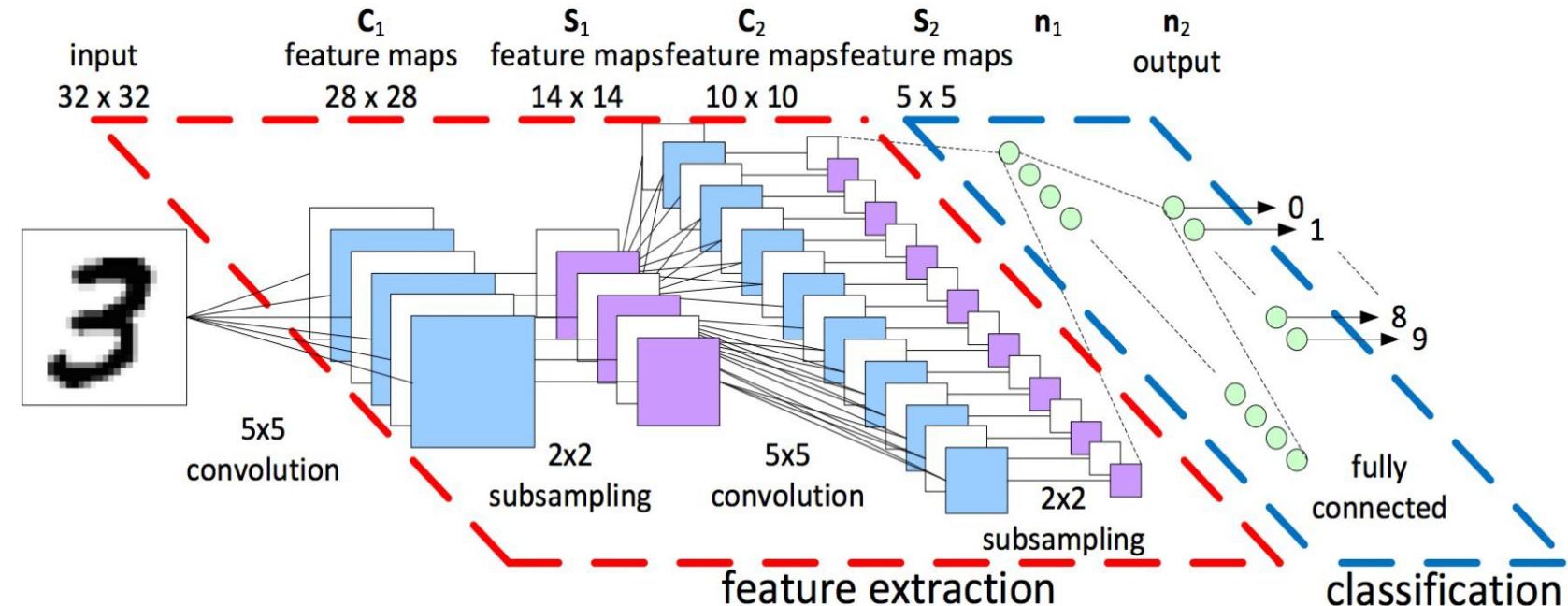
```
In [1]: import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable

x = Variable(torch.randn(10,128))
x = F.relu(x)
```

Train a Simple Neural Net



Train a simple Neural Net



1. Forward: compute output of each layer
2. Backward: compute gradient
3. Update: update the parameters with computed gradient



PyTorch: nn

Higher-level wrapper for working with Neural nets

Similar to Keras and friends ...
But only one and it's good

```
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 define common loss functions

```
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

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

```
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

```
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
```

Backward prop:
compute all gradients





PyTorch: nn

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
```



PyTorch: optim

Use an **optimizer** for different rules

```
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: optim

```
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)

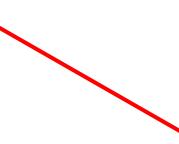
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

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()
```

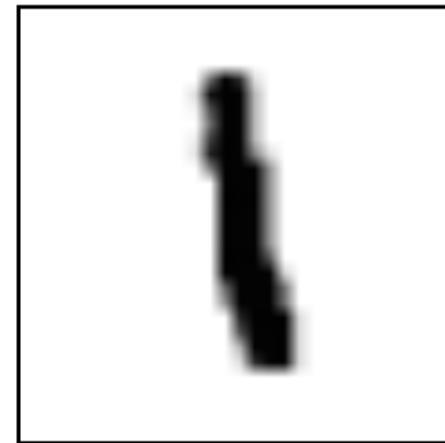
Update all parameters
after computing gradients



PyTorch in Action

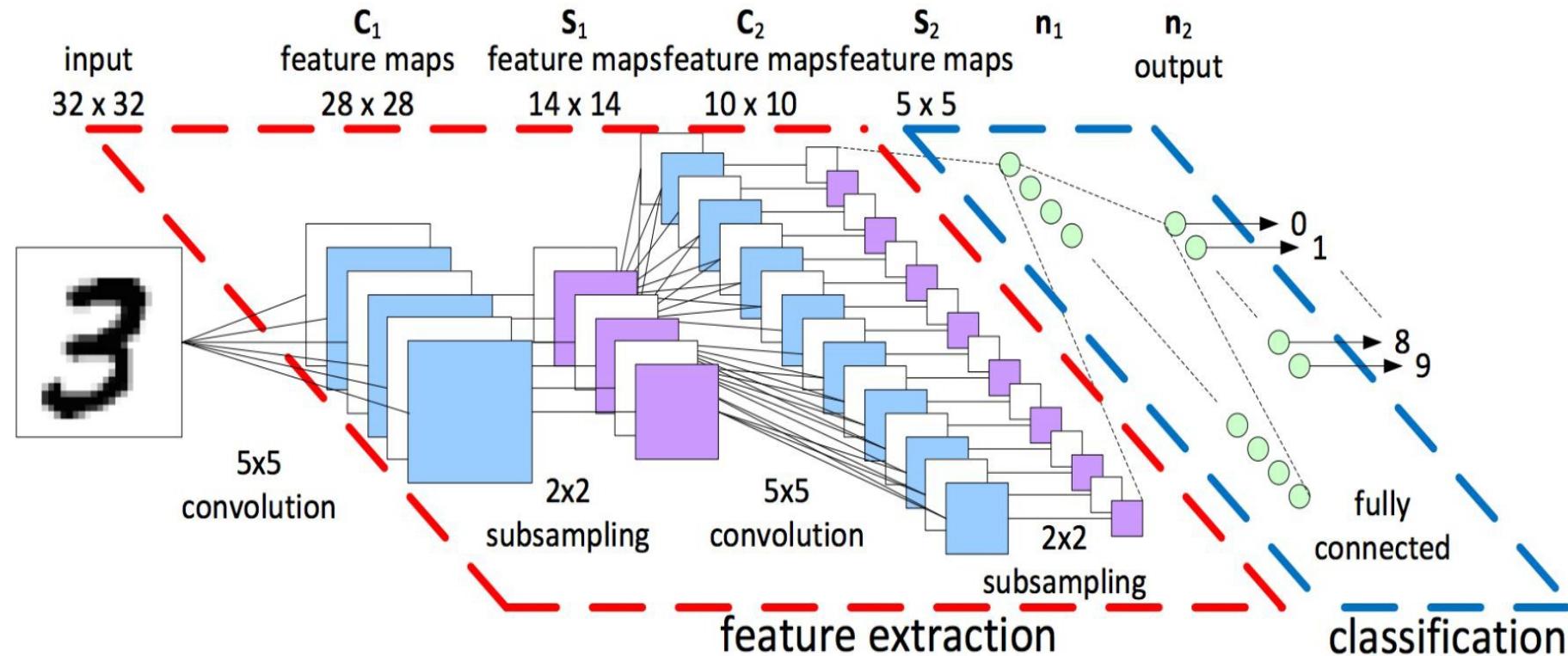


MNIST Dataset



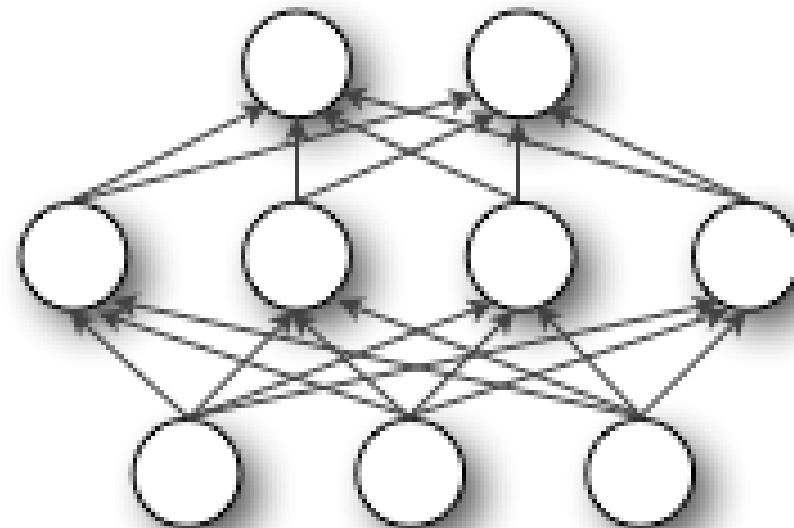


MNIST Example





MLP for MNIST (0-d features)



output layer

10

hidden layer

64

input layer

28*28



MLP for MNIST

In [3]:

```
class MLP(nn.Module):
    def __init__(self,n_class=10):
        super(MLP, self).__init__()

        self.fc = nn.Sequential(
            nn.Linear(28*28,64),
            nn.ReLU(inplace=True),
            nn.Linear(64,n_class)
        )

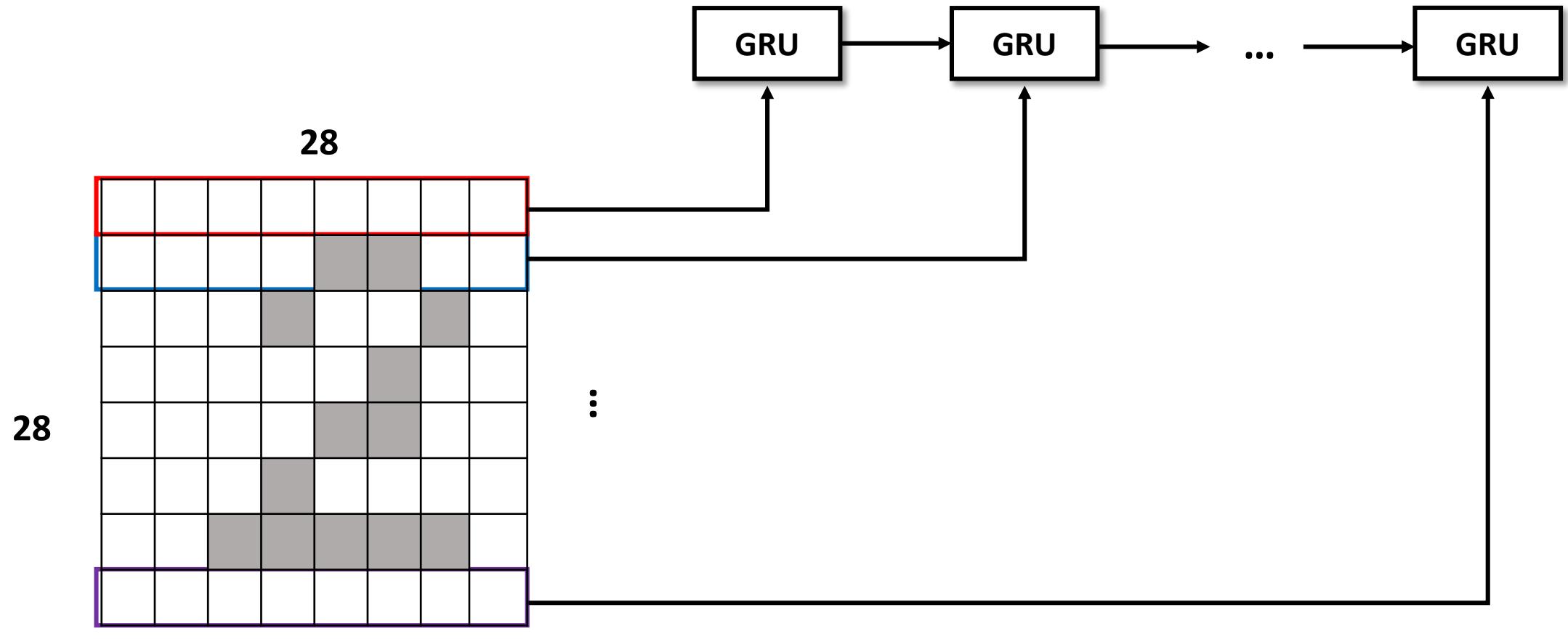
        """
        self.fc1 = nn.Linear(28*28,64)
        self.relu = nn.ReLU(inplace=True)
        self.fc2 = nn.Linear(64,n_class)
        """

    def forward(self, x):
        x = x.view(-1,28*28)      # x:(batch_size,1,28,28) => x:(batch_size,28*28)
        logits = self.fc(x)
        return logits
```

Last Test Acc: 95.8%



RNN for MNIST (1-d features)





RNN for MNIST

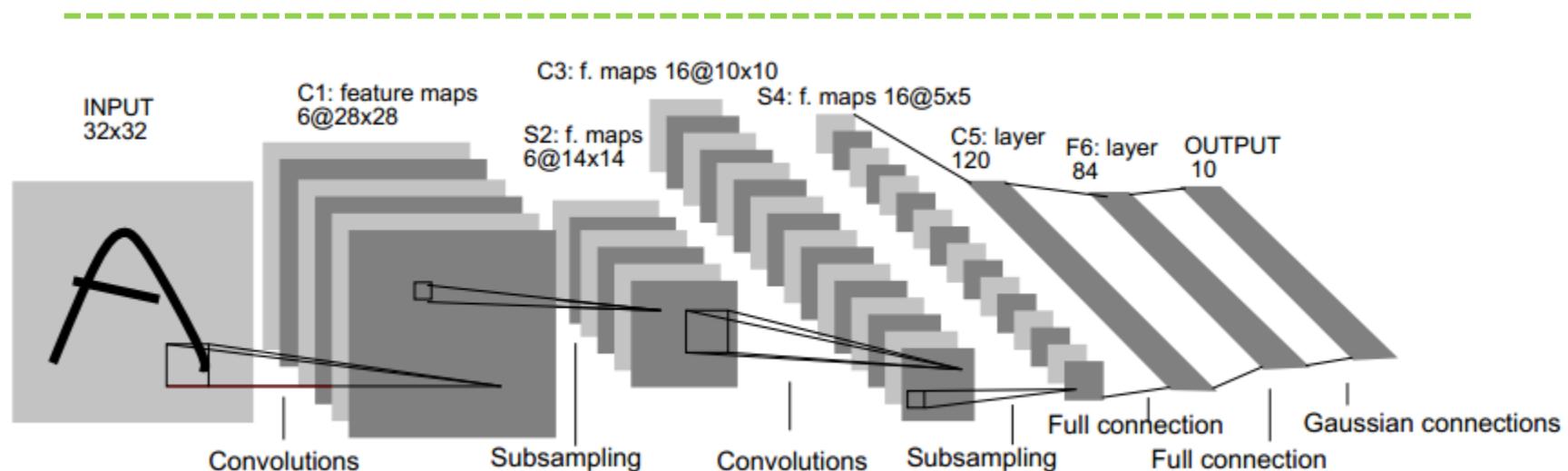
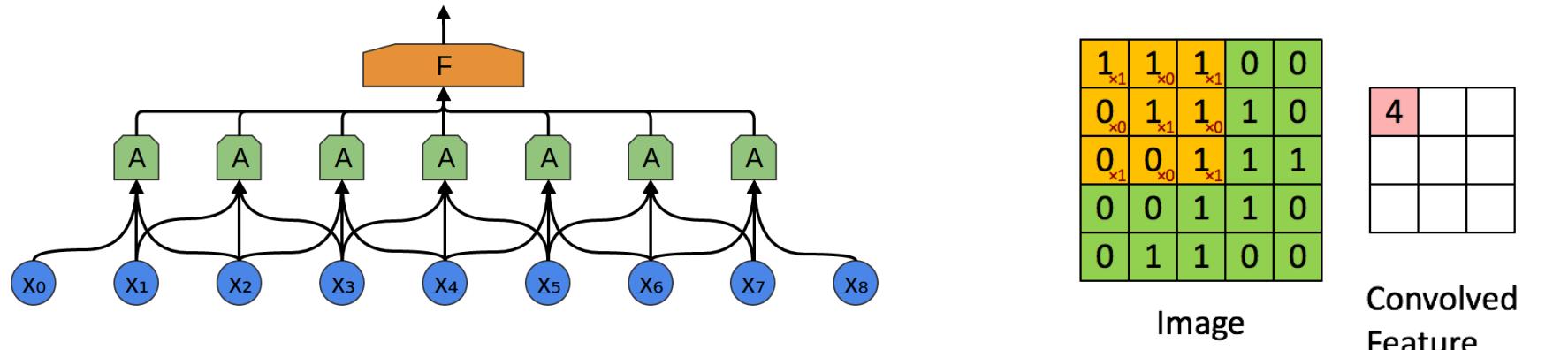
```
class RNN(nn.Module):
    def __init__(self, input_size=28, hidden_size=64, n_class=10):
        super(RNN, self).__init__()
        self.RNN = nn.GRU(
            input_size = input_size,
            hidden_size = hidden_size,
            batch_first = True
        )
        self.fc = nn.Linear(hidden_size, n_class)

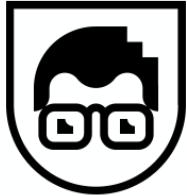
    def forward(self, x):
        x = x.squeeze()          # x:(batch_size,1,28,28) => x:(batch_size,28,28)
        out, _ = self.RNN(x)     # x:(batch_size,28,28) => out:(batch_size,28,hidden_size)
        # get last hidden
        out = out[:, -1, :]      # out:(batch_size,hidden_size)
        logits = self.fc(out)
        return logits
```

Last Test Acc: 97.7%



CNN for MNIST (2-d features)





CNN for MNIST

```
class LeNet(nn.Module):
    def __init__(self, n_class=10):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(
            in_channels=1,
            out_channels=20,
            kernel_size=5
        )
        self.conv2 = nn.Conv2d(
            in_channels=20,
            out_channels=50,
            kernel_size=5
        )
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, n_class)
    def forward(self, x):
        x = F.relu(self.conv1(x))      # x:[batch_size,1,28,28] => x:[batch_size,20, 24, 24]
        x = F.max_pool2d(x, 2, 2)     # x:[batch_size,20,24,24] => x:[batch_size,20, 12, 12]
        x = F.relu(self.conv2(x))      # x:[batch_size,20,12,12] => x:[batch_size,50, 8, 8]
        x = F.max_pool2d(x, 2, 2)     # x:[batch_size,50,8,8] => x:[batch_size,50, 4, 4]
        x = x.view(-1, 4*4*50)        # x:[batch_size,50,4,4] => x:[batch_size,50*4*4]
        x = F.relu(self.fc1(x))       # x:[batch_size,50*4*4] => x:[batch_size,500]
        x = self.fc2(x)               # x:[batch_size,500] => x:[batch_size,10]
        return x
```

Last Test Acc: 99.2%

Data Loading, Loss & Optimizers



Data Loading

Batch (batch size)

```
# Training cycle
for epoch in range(training_epochs):
    # Loop over all batches
    for i in range(total_batch):
```



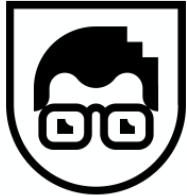
In the neural network terminology:



288

- one **epoch** = one forward pass and one backward pass of *all* the training examples
- **batch size** = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.
- number of **iterations** = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.



Dataloader

The `Dataset` retrieves our dataset's features and labels one sample at a time. While training a model, we typically want to pass samples in “minibatches”, reshuffle the data at every epoch to reduce model overfitting, and use Python’s `multiprocessing` to speed up data retrieval.

`DataLoader` is an iterable that abstracts this complexity for us in an easy API.

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```





Iterate through Dataloader

We have loaded that dataset into the `Dataloader` and can iterate through the dataset as needed. Each iteration below returns a batch of `train_features` and `train_labels` (containing ``batch_size=64 features and labels respectively). Because we specified `shuffle=True`, after we iterate over all batches the data is shuffled (for finer-grained control over the data loading order, take a look at [Samplers](#)).

```
# Display image and label.
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```

Loss Function

Common loss functions include `nn.MSELoss` (Mean Square Error) for regression tasks, and `nn.NLLLoss` (Negative Log Likelihood) for classification. `nn.CrossEntropyLoss` combines `nn.LogSoftmax` and `nn.NLLLoss`.

We pass our model's output logits to `nn.CrossEntropyLoss`, which will normalize the logits and compute the prediction error.

```
# Initialize the loss function
loss_fn = nn.CrossEntropyLoss()
```

Optimizer

We initialize the optimizer by registering the model's parameters that need to be trained, and passing in the learning rate hyperparameter.

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Inside the training loop, optimization happens in three steps:

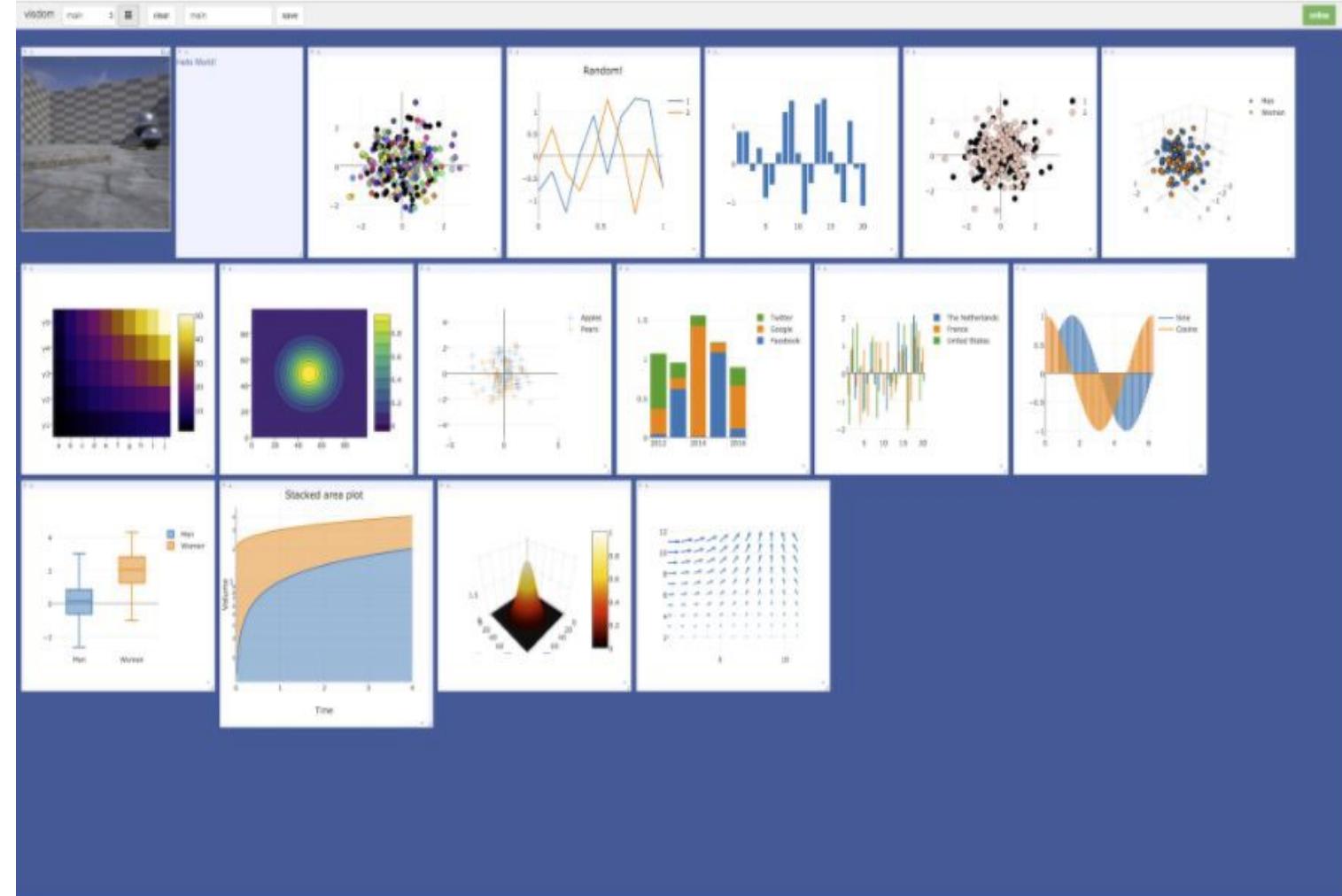
- Call `optimizer.zero_grad()` to reset the gradients of model parameters. Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration.
- Backpropagate the prediction loss with a call to `loss.backwards()`. PyTorch deposits the gradients of the loss w.r.t. each parameter.
- Once we have our gradients, we call `optimizer.step()` to adjust the parameters by the gradients collected in the backward pass.

Visualisation



Visualisation

Visdom

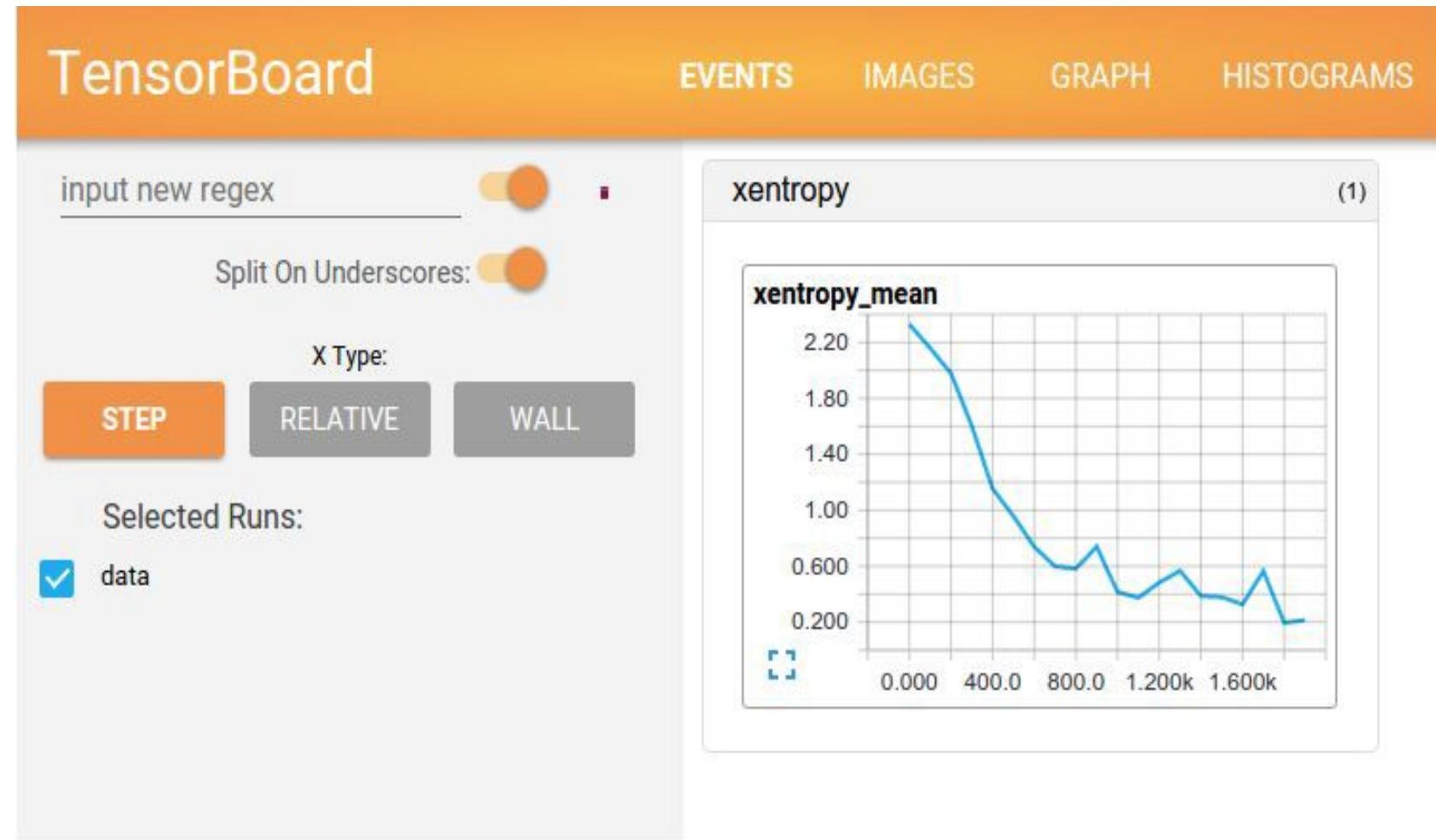


<https://github.com/facebookresearch/visdom>



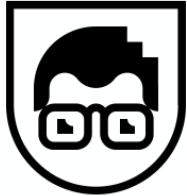
Visualisation

TensorBoard



https://www.tensorflow.org/get_started/summaries_and_tensorboard

Save & Load



Saving and Loading Model Weights

PyTorch models store the learned parameters in an internal state dictionary, called `state_dict`. These can be persisted via the `torch.save` method:

```
model = models.vgg16(pretrained=True)
torch.save(model.state_dict(), 'model_weights.pth')
```

To load model weights, you need to create an instance of the same model first, and then load the parameters using `load_state_dict()` method.

```
model = models.vgg16() # we do not specify pretrained=True, i.e. do not load default weights
model.load_state_dict(torch.load('model_weights.pth'))
model.eval()
```



Saving and Loading Models with Shapes

When loading model weights, we needed to instantiate the model class first, because the class defines the structure of a network. We might want to save the structure of this class together with the model, in which case we can pass `model` (and not `model.state_dict()`) to the saving function:

```
torch.save(model, 'model.pth')
```

We can then load the model like this:

```
model = torch.load('model.pth')
```



Conclusion

- PyTorch	Scientific computing package based on python
- Tensor	Fundamental datatype or most basic element in PyTorch equivalent to numpy.ndarray
- FloatTensor	Float Tensor datatype
- Autograd	Automatic differentiation of arbitrary scalar valued functions
- Variable	Wrapper around Tensor that contains three attributes of data, grad and grad_fn
- Computational Graph	Type of graph that can be used to represent mathematical expressions
- Torch.nn	PyTorch Class to help you create and train neural network
- Torch.nn.functional	PyTorch Class to Create and train net that uses functions
- GPU	Graphical interface to accelerate multiple computations simultaneously
- GRU	Gating mechanism in RNN
- Dataloader	Iterable that combines a dataset and a sampler over the given dataset
- Batch size	Number of training examples one forward/backward pass
- Iterations	Number of passes
- Epoch	Method of evaluating how well an algorithm is modeling the dataset
- Loss Function	Method of evaluating how well the algorithm is modeling the dataset
- Optimizer	Minimizer of the loss function
- Visdom	Visualisation tool that generates rich visualizations of live data
- TensorBoard	Visualisation tool for machine learning experimentation



Resources

- Official resources
 - Documentation <http://pytorch.org/docs/master/>
 - Tutorials <http://pytorch.org/tutorials/index.html>
 - Example projects <https://github.com/pytorch/examples>
- Github code
 - fairseq-py
 - OpenNMT-py
- Open course & tutorial used in this presentation
 - **Stanford CS231N 2017**, Lecture 8 “Deep Learning Software”
 - <https://pytorch.org/tutorials/beginner/basics/intro.html>