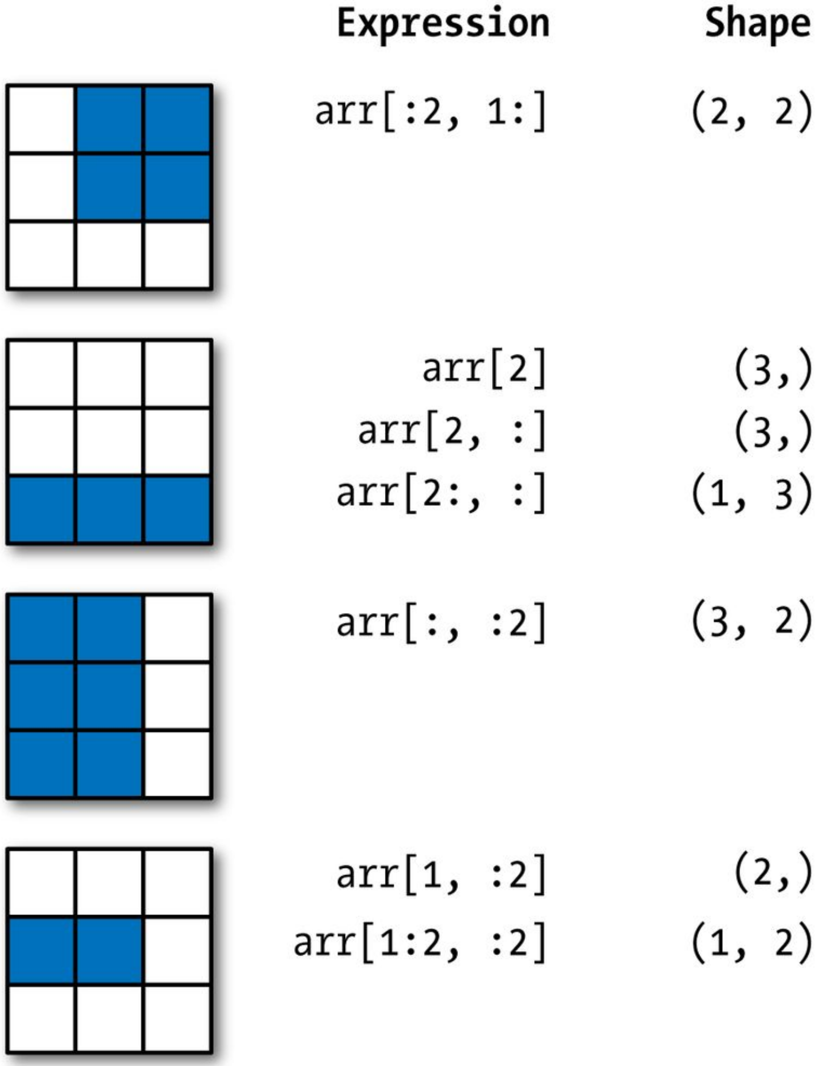
Introduction to PyTorch

Nikos Gkanatsios

Credits to Sai Shruthi Balaji

Preliminaries: NumPy

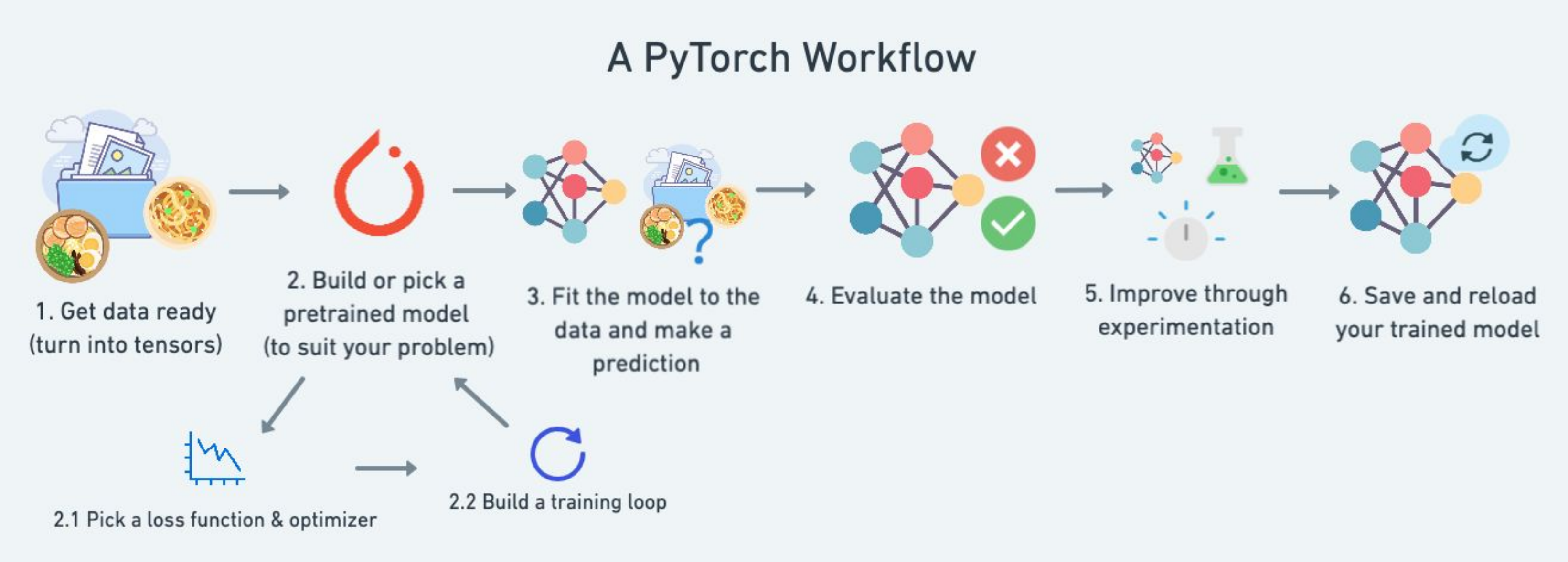
```
In [1]: import numpy as np

In [2]: a = np.array([1,2,3,4,5,6,7,8,9])

In [3]: a
Out[3]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In [4]: b = a.reshape((3,3))

In [5]: b
Out[5]:
array([[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])

In [6]: b * 10 + 4
Out[6]:
array([[14, 24, 34],
[44, 54, 64],
[74, 84, 94]])
```



What is PyTorch?

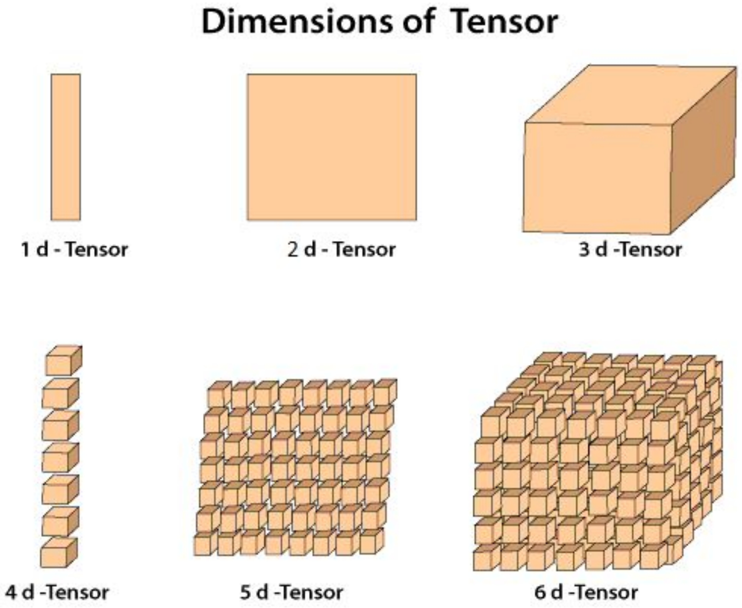
PyTorch

- An open-source machine learning framework that accelerates the path from research prototyping to production deployment.



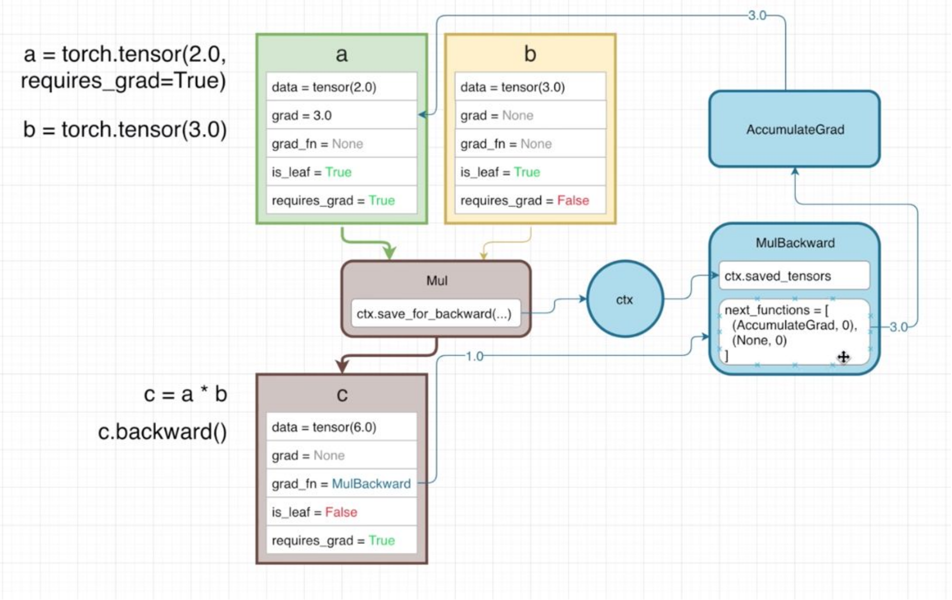
Tensors: Backbone of PyTorch

- Multi-dimensional array, same as numpy array
- Biggest difference: Tensors can be run on CPU/GPU



Tensors Operations

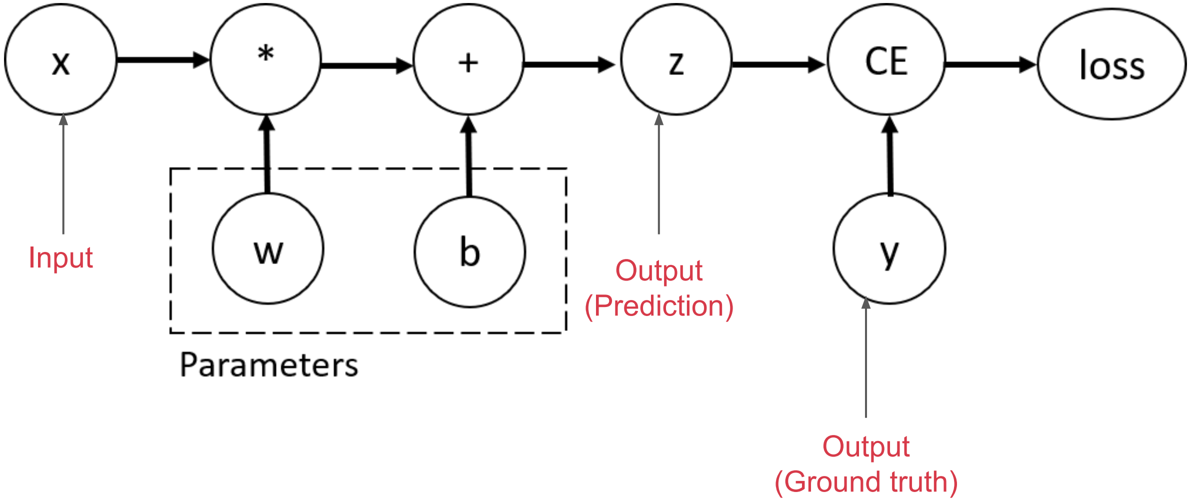
1. Directly stores input data, weights, etc
2. Holds grad, grad_fn (requires_grad)
3. detach()
4. clone()
5. numpy()
6. tensor.device
7. to(device) -> CPU or GPU / which GPU? (cuda:0)
8. tensor.dtype
9. to(dtype)
10. Between Tensors: similar operations like numpy sucl as:
    - Add, Matmul, Subtract, Concat, etc.



Autograd

- This class is PyTorch's automatic differentiation engine that powers neural network training.

```
import torch

x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

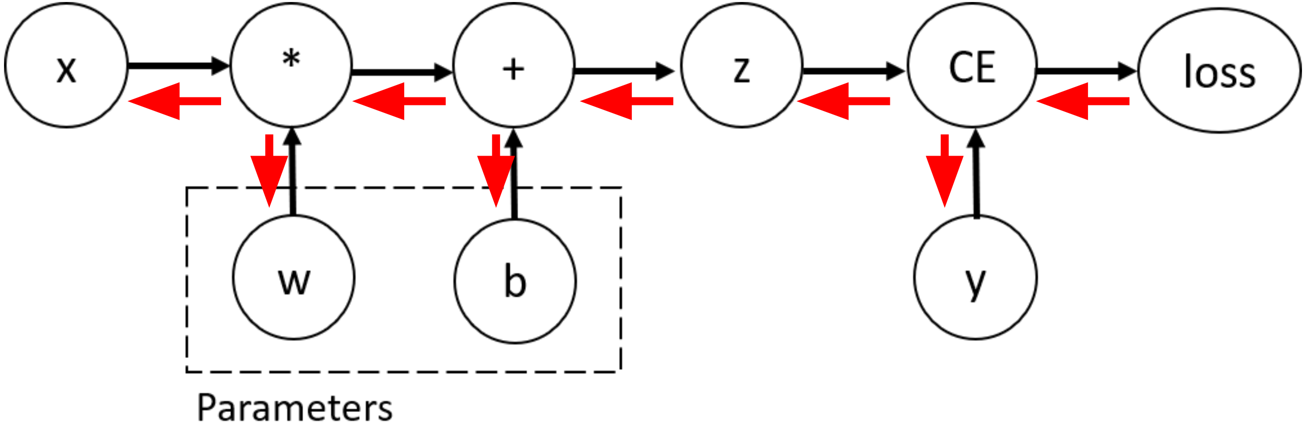


Autograd

```
loss.backward()
print(w.grad)
print(b.grad)
```

Out:

```
tensor([[0.0659, 0.0797, 0.2611],
        [0.0659, 0.0797, 0.2611],
        [0.0659, 0.0797, 0.2611],
        [0.0659, 0.0797, 0.2611],
        [0.0659, 0.0797, 0.2611]])
tensor([0.0659, 0.0797, 0.2611])
```

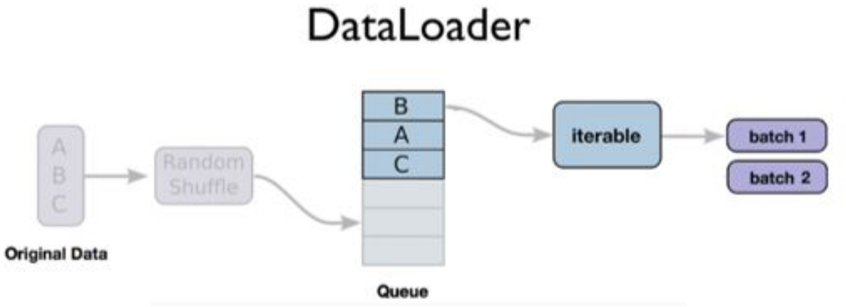


Parts of a Framework

- Data loading
- Model definition: Forward Pass
- Loss
- Backward Pass
- Optimization
- Validation

Data Loading

- Load data, turn into tensors, and batch
- Common image datasets available in torchvision.datasets
- DataLoader can handle shuffling and batching
- Writing your own DataSet class:
  - __init__
  - __len__
  - __getitem__



Defining a model: nn.Module

- Model class extends nn.Module and has the following methods:
  - __init__
  - forward

- nn.Module advantages:
  - Module reuse
  - Easy chaining of multiple steps
  - Intermediate states are held in compute graph

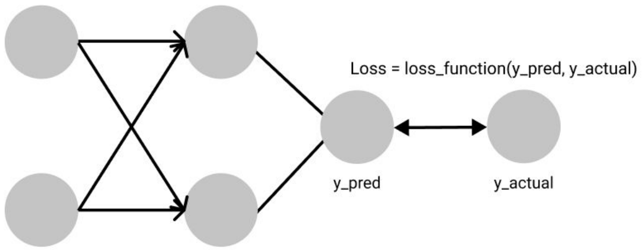
```
class TinyModel(torch.nn.Module):

    def __init__(self):
        super(TinyModel, self).__init__()

        self.linear1 = torch.nn.Linear(100, 200)
        self.activation = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(200, 10)
        self.softmax = torch.nn.Softmax()

    def forward(self, x):
        x = self.linear1(x)
        x = self.activation(x)
        x = self.linear2(x)
        x = self.softmax(x)
        return x
```

nn.Module: Useful Facts

1. module.parameters() -> gathers all nn.Parameter in the module
2. parameters() -> parameters to “update/optimize”
3. state_dict() -> dictionary of all parameters and buffers -> torch.save
4. load_state_dict(state_dict) -> used to load pretrained weights
5. train() - trainable weights
6. eval() - frozen dropout/norm layers
7. to(device) - train on GPU

Loss function

- Measures how well the model is doing
- Key for training:
  - Used to compute gradients
  - Used in back propagation (loss.backward)
- Different loss functions for different problems:
  - Regression: L1 Loss, L2 Loss
  - Classification: Cross Entropy Loss, NLL Loss, BCE Loss
  - Ranking: Margin Ranking Loss, Triplet Margin Loss
  - KL Divergence Loss: Difference between distributions



Optimization: torch.optim

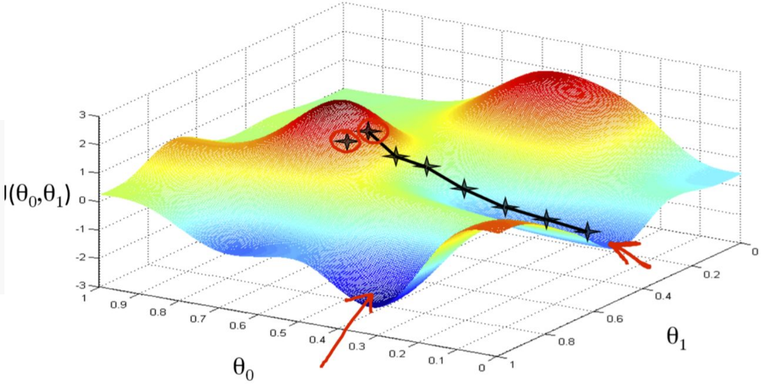
- Pass parameters, define learning rate and other optional params.

```
optim.SGD([
                {'params': model.base.parameters()},
                {'params': model.classifier.parameters(), 'lr': 1e-3}
            ], lr=1e-2, momentum=0.9)
```

- Set gradients as zero and take a step.

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

- Common optimizers: Adam, SGD, RMSProp



Validation

- Download test dataset, run it periodically to look at accuracy/loss
- Set model.eval() to deactivate dropout and norm layers from updating
- Run with torch.no_grad to deactivate gradients
- Use an evaluation metric.
    - Eg: mAP for classification
    - Eg: IOU for bounding box prediction

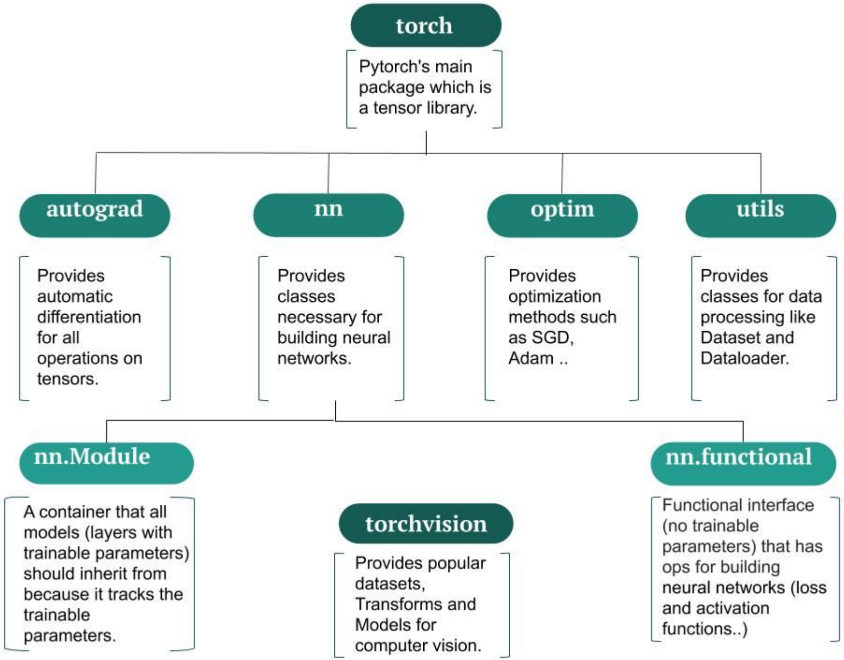
Running on the GPU

https://pytorch.org/docs/stable/notes/cuda.html

1. Move your dataset and your model to the GPU
2. Simple: .cuda()
3. Can also create the device and do .to(device)

Summary

1. Load data
   a. torchvision is sometimes convenient
   b. DataLoader for batching, shuffling
2. Define net
   a. Convenient to implement nn.Module, to get parameters, zero_grad, etc easily
3. Define loss
   a. Pytorch provides a lot of these, eg in torch.nn.functional
4. Backward automatically computes gradients
   a. Can manually clip, etc if desired
5. Optimizer to update the given parameters
   a. torch.optim



References

- https://pytorch.org/tutorials/beginner/blitz/
- https://pytorch.org/tutorials/beginner/basics/intro.html
- https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html
- https://github.com/yunjey/pytorch-tutorial
- https://pytorch.org/tutorials/beginner/ptcheat.html
- https://web.cs.ucdavis.edu/~yjlee/teaching/ecs289g-winter2018/Pytorch_Tutorial.pdf
- https://www.kaggle.com/code/residentmario/pytorch-autograd-explained/notebook
- https://neptune.ai/blog/pytorch-loss-functions