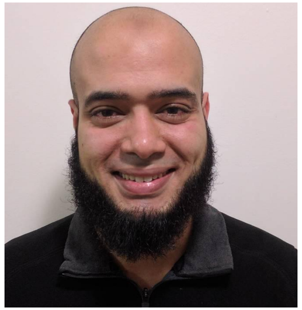
Machine Learning
PyTorch Basics

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / MSc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

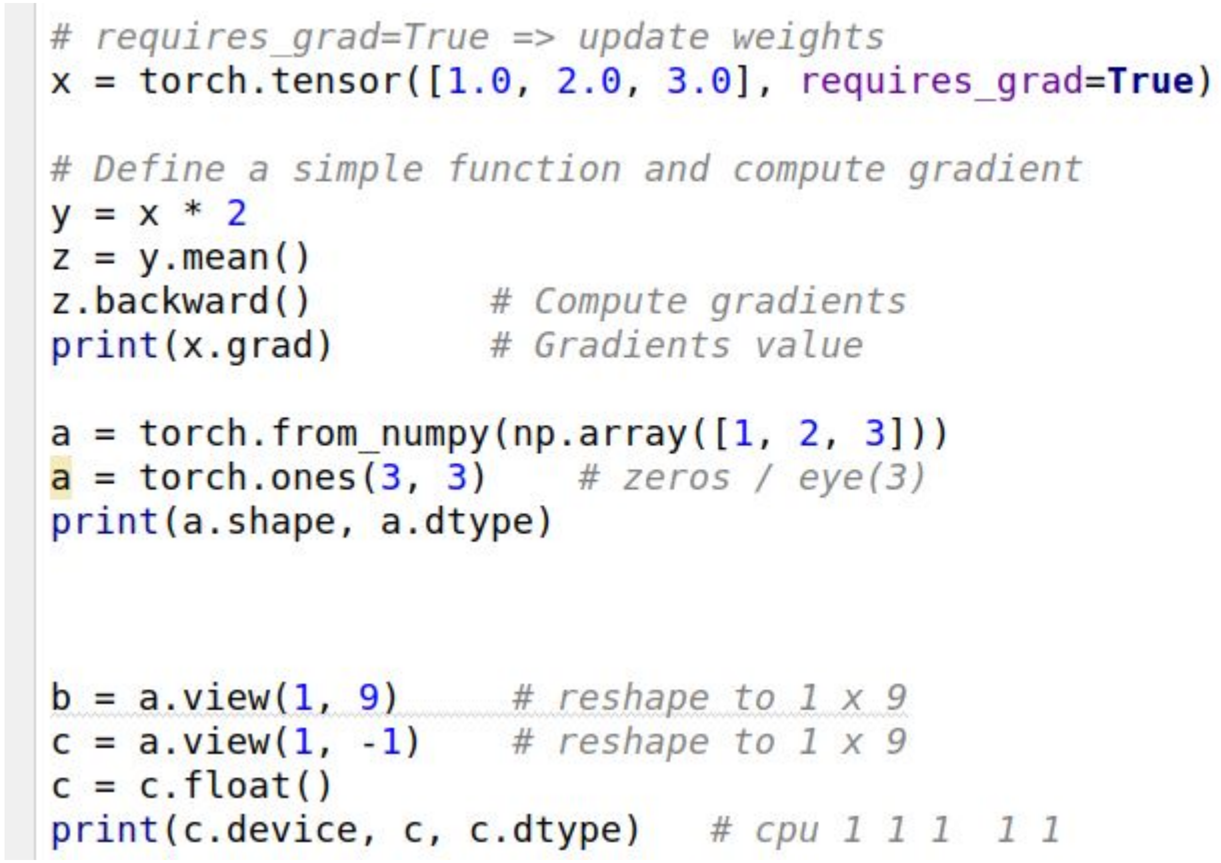


Tensor

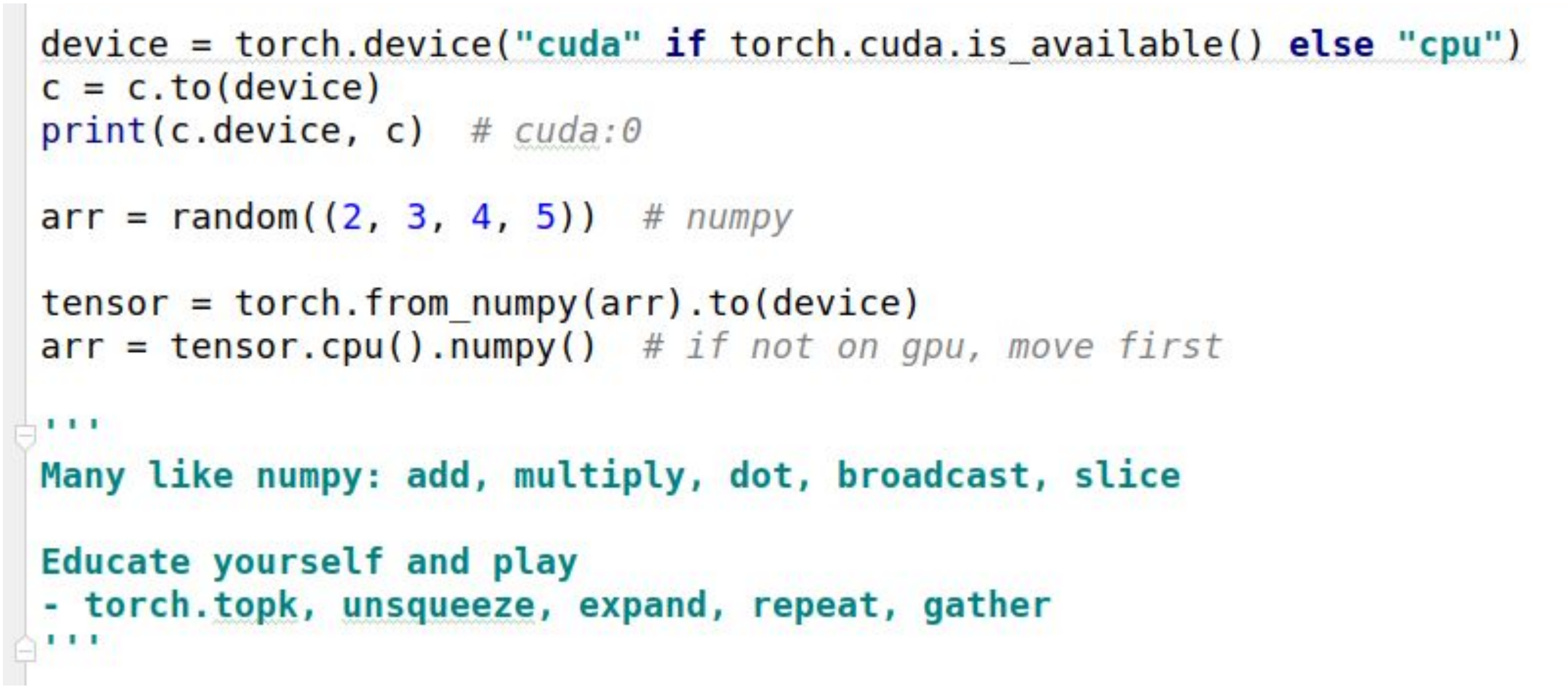
- A tensor is a multi-dimensional array of numbers that can run on GPU
    - Rank: 0 for Scalar / 1 for Vector / 2 for Matrix
    - Typically used for inputs (e.g. images), Outputs (e.g. heatmaps) and weights
    - Has many operations like **numpy**
- In PyTorch, a 4-D tensor is used: B * C * W * H
    - B is for the train/test **batch size**
    - C number of channels (e.g. 3 for an RGB image)
    - W x H: width and height of the 2D perspective
    - For example, training with 64 batches of images of sizes 600*800 ⇒ 64 * 3 * 600 * 800

PyTorch Autograd Engine

- The automatic differentiation engine in PyTorch (Autograd) automatically computes the gradients for tensor operations
  - As a result, we just write the feedforward only
  - This feature allowed huge growth in developing DNNs
- When you perform tensor operations with **requires_grad=True**, PyTorch keeps **track** of all the operations
- When you call .**backward**() on a tensor, Autograd computes gradients for each tensor involved in the creation of that tensor, provided they have requires_grad=True
  - Accumulating gradients across different mini-batches
- The computed gradients are populated in the **.grad attribute** of the tensor

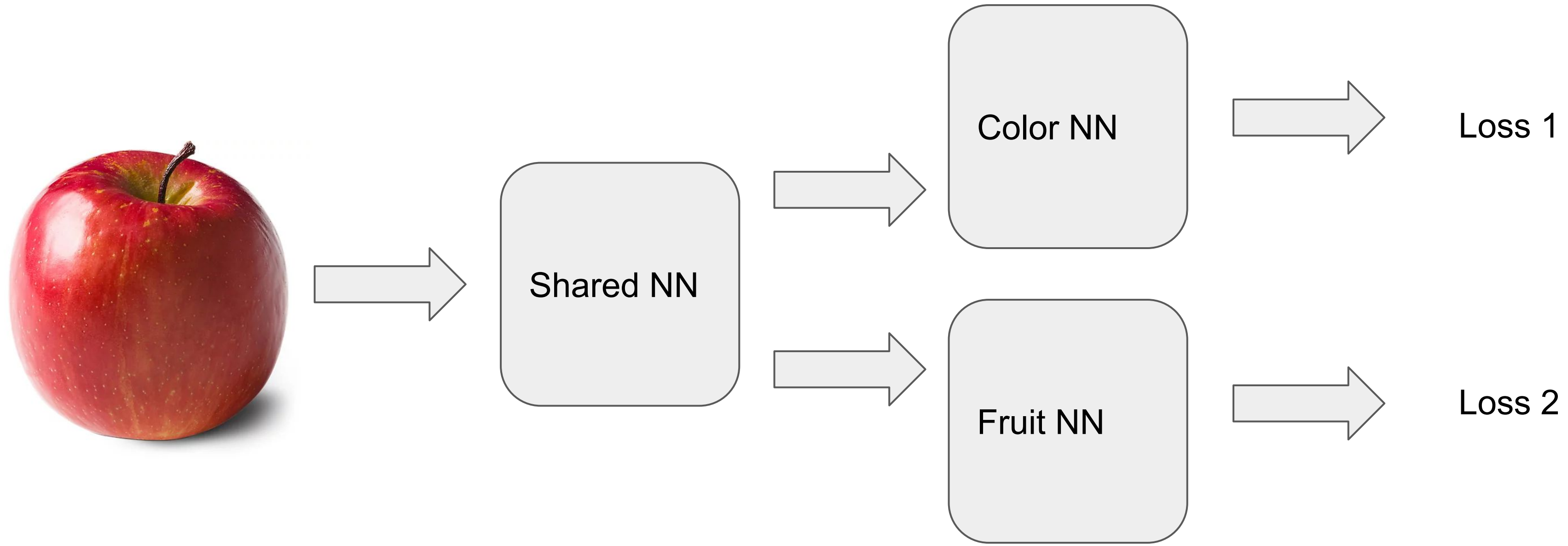


```python
# requires_grad=True => update weights
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)

# Define a simple function and compute gradient
y = x * 2
z = y.mean()
z.backward()          # Compute gradients
print(x.grad)         # Gradients value

a = torch.from_numpy(np.array([1, 2, 3]))
a = torch.ones(3, 3)    # zeros / eye(3)
print(a.shape, a.dtype)



b = a.view(1, 9)       # reshape to 1 x 9
c = a.view(1, -1)      # reshape to 1 x 9
c = c.float()
print(c.device, c, c.dtype)   # cpu 1 1 1  1 1
```



```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
c = c.to(device)
print(c.device, c)  # cuda:0

arr = random((2, 3, 4, 5))  # numpy

tensor = torch.from_numpy(arr).to(device)
arr = tensor.cpu().numpy()  # if not on gpu, move first

'''
Many like numpy: add, multiply, dot, broadcast, slice

Educate yourself and play
- torch.topk, unsqueeze, expand, repeat, gather
'''
```

Updates only with backward



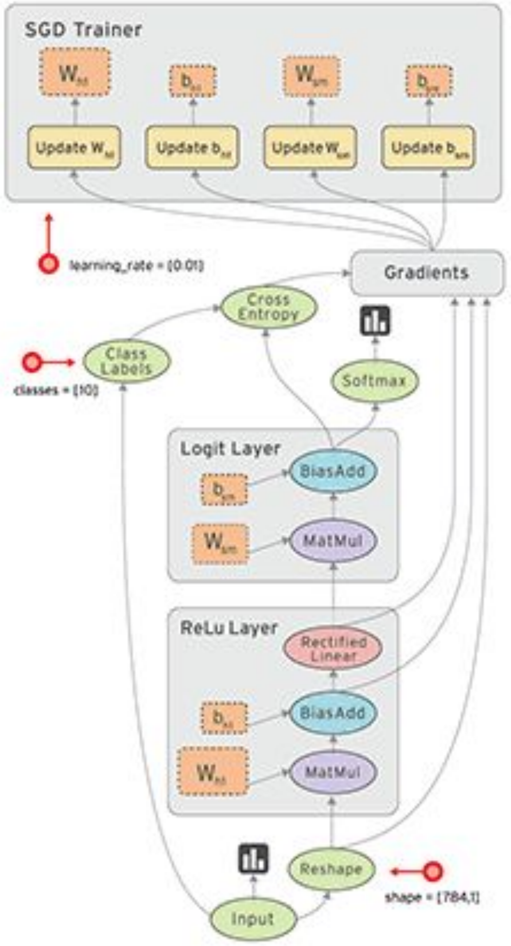
- We typically do backward on a loss output
- Each weight is updated based on backwards on losses
- For example, if loss 2 is not activated, then Fruit NN weights won’t get updated
- This is very useful for complex training of specific network branches

Computational Graph

- Frameworks build the graph in 2 different ways
    - **Static**: Defined once
        - Allows for powerful offline optimization ⇒ faster
        - However, not flexible for variable-sized data
        - **Tensorflow** use it + has eager execution mode (similar to dynamic graphs)
    - **Dynamic**: the graph is generated on-the-fly and can be different **at each iteration**.
        - **Building** the graph and **computing** the graph happen at the **same** time.
        - Useful for variable-length inputs in tasks like NLP and object detection
        - **PyTorch** takes this approach
- Tip: **in-place** operations sometimes cause problems

Graph Example

- Just extended flow compare to NNs



Img src

Relevant Materials

- Automatic Differentiation / Impl
- Computational graphs in PyTorch and TensorFlow
- The Fundamentals of Autograd

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."