Intro to PyTorch

PyTorch vs Scikit-Learn

- sklearn/scikit-learn is a general machine learning library with utilities for data-processing, model training and evaluation
- PyTorch has additional support for large neural networks – “deep learning library”
  - Tensors – sklearn uses numpy arrays as the underlying math library which allows for “vectorized” computation (https://en.wikipedia.org/wiki/Single_instruction,_multiple_data). Similar to Numpy arrays, PyTorch uses “Tensors” which also perform vectorized operations but also have gpu support
  - PyTorch Autograd – An automatic differentiation module which computes gradients numerically, efficiently
  - “Deep Learning” utilities – activation functions, loss functions, large scale optimizers

Pytorch Tensors (link)

- Used to represent data, model parameters etc. Just like sklearn functions take numpy-arrays as input, pytorch functions take tensors as input
- Can be stored/processed in GPUs
- Has support for automatic-differentiation i.e you can get the gradient/jacobian of some function at the input tensor you gave (efficiently)
- Creating tensors:
  - From python arrays like

```
x = torch.tensor([1,2,3], dtype=torch.float)        #create a tensor of floats from python list
```

  - From numpy arrays

```
a = np.array([1,2,3], dtype=float)     #numpy array of floats
x = torch.tensor(a)                    #torch tensor from numpy array (new array)
x = torch.from_numpy(a)                #torch tensor from numpy array (sharing memory locations)
```

Pytorch Tensors (link)

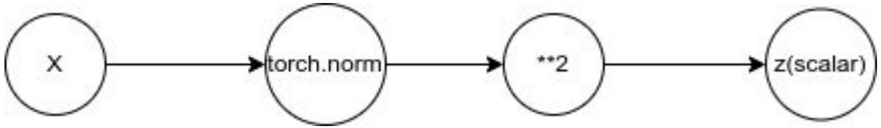
- You can do operations like in numpy arrays – add, subtract, multiply, dot-product, resize, concat etc
- You can move tensors to GPU (where array operations can be executed faster)

```
if torch.cuda.is_available():
	device = torch.device("cuda:0")
#we are moving a tensor to the first GPU (indexed by 0)
a = torch.rand(3, dtype=torch.float).to(device)
```

Pytorch Autograd (link)

- An automatic differentiation engine builds the computation-graph and can calculate gradients efficiently (using the backpropagation algorithm)
- During the “forward-pass”, the computational graph is built efficiently keeping in mind which leaves need gradient
- In the “backward pass”, the gradient of final root node w.r.t the leaves is computed

```
x = torch.rand(3, requires_grad=True)
y = torch.norm(x, 2)          #calculate l2 norm
z = y*y                       #z has square of euclidean norm
z.backward()                  #command to perform backward-pass
print(x.grad)                 #x.grad prints the gradient of z w.r.t x (a vector)
```



Pytorch Autograd (link)

```
In [225]: x = torch.rand(3, requires_grad=True)

In [226]: y = torch.norm(x, 2)

In [227]: y.retain_grad()

In [228]: z = y*y

In [229]: z.backward()

In [230]: z.grad_fn
Out[230]: <MulBackward0 at 0x7fe3f75e2250>

In [231]: y.grad_fn
Out[231]: <NormBackward1 at 0x7fe3f763eb90>

In [232]: x.grad
Out[232]: tensor([1.8328, 1.4116, 0.1263])

In [233]: x
Out[233]: tensor([0.9164, 0.7058, 0.0632], requires_grad=True)

In [234]: x.grad == 2*x
Out[234]: tensor([True, True, True])

In [235]:
```

This is the strength of pytorch, it allows you to construct computational graphs of your choice and efficiently compute gradients for training.

By default **nn.Module** parameters have the requires_grad option set to true