Tricks

- **All functions that are followed by a _ are in-place, e.g: `uniform_(0, 1)`, `add_(x)`, etc**
    - In-place functions are usually **more efficient**
    - **+=** is also in-place
- **You can't use view() if you have a transposed matrix!!! (*requires contiguous memory*)**
- **`z = x.view(batch, -1)` -** this flattens the whole thing out
- Transposing is a special case of the permute function

Maths

`torch.empty(shape)` - uninitialized memory - whatever was in the memory at the time

`torch.rand(shape)` , **`torch.zeros(shape)`** , **`torch.eye(shape)`** , **`torch.ones(shape)`** , **torch.arange(s,e,s)**

`torch.tensor(arr, dtype=_, device=_, requires_grad=_)` - This tells us if the tensor needs gradients

torch.linspace(start, end, steps) - *arange* defines step size but linspace the number of values, and has ++options

`tensor_x.normal_(mean=0, std=1)` - normalizes data *(in place function)*

`tensor_x.uniform_(0,1)` - uniform distribution *(in place function)*

`torch.diag(torch.ones(3))` - creates a diagonal matrix from tensor ---------------->

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & 3\end{bmatrix}$$

Conversion (works no matter the device you're on)

`tensor:` `.bool()`, `.short(`*`int16`*`)`, `.long(`*int64*`)`, `.half(`*float16, only new gpus*`)`, `.float(`*float32*`)`, `.double(`*float64*`)`

`torch.from_numpy(np_arr)`

`np_arr = tensor.numpy()`

Tensor Math and Comparison

- You can use the default **+ -** operators or explicit functions **.add()**, etc
- **`torch.true_divide(x, y)`**
    - If the two shapes are the same, **element-wise division**
    - If **y** is an integer, it divides all elements by **y**
- **`x.pow(2)`** is the same as **`x ** 2`**
- **`z = x > 0`** returns a boolean array
- **Matrix Multiplication: `x1.mm(x2)`** or **`torch.mm(x1, x2)`** *(2x3 * 3x5 = 2x5)*
- **`z = x * y`** is **element-wise** multiplication
- **`torch.dot(x, y)`** dot product is for vectors, matrix multiplication for matrices
- **`x1.matrix_power(3)`**
- **`torch.bmm(x1, x2)`** - Batch Multiplication for two vectors of shape **(batch, x, y)**. Inside dimensions must match

Broadcasting

```
x1 = torch.rand((5, 5))
x2 = torch.rand((1, 5))
z = x1 - x2        z = x1 ** x2
```

If you do something like subtracting a vector from a matrix, **pyTorch** *(and NumPy)* broadcast the vector, i.e. copy it 5 identically 5 times to make the shapes match

Other useful things

`torch.any(x)` returns true for a boolean array if *any value* is **true**

`torch.all(x)` returns true for a boolean array if *all values* is **true**

Torch Indexing

`x[0] == x[0,:]`

Yaddy yadda same as numpy

Fancy Indexing

`x[list_of_indices]`

`x[(x < 2) | (x > 8)]`

Useful Operations

`torch.where(x > 5, x, x*2)` **-** condition, how to change true vals, and how to change false vals

`x.ndimension()` **-** length of shape

`x.numel()` **-** total number of elements in tensor

Reshaping Tensors

`x.arange(9)`

`x_3x3 = x.view(3, 3)` - this needs the tensor to be in contiguous memory, superior performance

- **This means you can't use view() if you have a transposed matrix!!!**

`x_3x3 = x.reshape(3, 3)` - this doesn't, if it isn't it just makes a copy, safer but performance loss

`z = x.view(-1)` **-** this flattens the whole thing out

`z = x.view(batch, -1)` **-** this flattens the whole thing out

`torch.cat((x1, x2), dim=0)`

z = x.permute(0, 2, 1) - If you want to switch the order of axes (*i.e. (batch, x, y) -> (batch, y, x)*)

- Transposing is a special case of the permute function

`x.unsqueeze(0)` - adds a dimension in the 0th spot

`x.squeeze(0)`

torch.nn.Embedding -