1 Preliminaries

In this section, code will be provided where applicable.

1.1 Data Manipulation

1.1.1 Getting Started

- A tensor represents a (possibly multi-dimensional) array of numerical values
  - With one axis, a tensor is called a *vector*
  - With two axes, a tensor is called a *matrix*
  - With $n > 2$ axes, we just call it a tensor and it is referred to as a k^{th} order tensor for k dimensions

1.1.2 Vectors

Here is how you can create a column vector in pytorch.

```
# Library import
import torch
# Assign x to an array with 12 floats
x = torch.arange([12, dtype=torch.float])
x

"""
Returns:

  tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11.])
"""
```

The tensor **x** contains 12 elements. We can determine the total number of elements in a tensor using the **numel** method.

```
x.numel()

"""
Returns:

  12
"""
```

We can also determine the shape of a tensor by calling a tensor's **shape** method.

```
x.shape()

"""
Returns:

  torch.Size([12])
"""
```

1.1.3 Matrices

We can change the shape of a tensor without altering its size or values by calling a tensor's **reshape** method. For example, we can reshape a vector **x** whose shape is (12,) to a matrix **X** whos shape is (3, 4). **X** retains all of the elements of **x**, but organizes them into a matrix.

```
X = x.reshape()
X

"""
Returns:

  tensor([[0., 1., 2., 3.],
    [4., 5., 6., 7.],
    [8., 9., 10. 11.]])
"""
```

Specifying each shape compoenent can be redundant, so we can work out one component and pytorch will infer the rest. To automatically infer a component, we can place a -1 to the component that will be inferred. In our example, instead of calling **x.reshape(3, 4)**, we can call **x.reshape(-1, 4)** or equivalently **x.reshape(3, -1)**

1.1.4 Additional Functions

We can call the **torch.zeros** function to create a tensor of zeros.

```
torch.zeros((2, 3))

"""
Returns:

  tensor([[0., 0., 0.],
    [0., 0., 0.]])
"""
```

Likewise, we can call the **torch.zeros** function to create a tensor of ones.

```
torch.ones((3, 2))

"""
Returns:

  tensor([[1., 1.],
    [1., 1.],
    [1., 1.]])
"""
```

1.1.5 Indexing and Slicing

For this section, we will use the matrix **X** from our previous example

```
X

"""
Returns:

  tensor([[0., 1., 2., 3.],
    [4., 5., 6., 7.],
    [8., 9., 10. 11.]])
"""
```

Similar to python lists, we can access tensor elements by indexing (starting with zero). To access an element based on its position relative to the end of the list, we can use negative indexing. We can access whole ranges of indices via slicing (e.g., **X[start:stop]**), where the returned value includes the first index **(start)** but not the last **(stop)**.

```
X[-1], X[1:3]

"""
Returns:

  tensor([8., 9., 10. 11.]),
  tensor([[4., 5., 6., 7.],
    [8., 9., 10. 11.]])
"""
```

You can also reassign elements of a matrix by specifying indices.

```
X[1, 2] = 17
X

"""
Returns:

  tensor([8., 9., 10. 11.]),
  tensor([[4., 5., 6., 7.],
    [8., 9., 10. 11.]])
"""
```

Similarly

```
X[:2, :] = 12
X

"""
Returns:

  tensor([12., 12., 12. 12.]),
    [12., 12., 12. 12.],
    [8., 9., 10. 11.]])
"""
```

1.1.6 Operations

We can calculate the exponential e^x using **torch.exp(x)**. Below are your common mathematical operations.

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y , x - y, x * y, x / y, x ** y

"""
Returns:

  (tensor([3., 4., 6., 10.]),
    tensor(-1., 0., 2/, 6.]),
    tensor([2., 4., 8., 16.]),
    tensor([0.5000, 1.0000, 2.0000, 4.0000]),
    tensor([1., 4., 16., 64.])
    )
"""
```

We can also concatenate multiple tensors along a specific axis

```
X = torch.arange(12, dtype=torch.float32).reshape((3, 4))
Y = torch.tensor([[2.0, 1, 4, 3],
  [1, 2, 3, 4],
  [4, 3, 2, 1]])
torch.cat((X, Y), dim=0) # Join by row, this is similar to a SQL Union

"""
Returns:

  tensor([0., 1., 2., 3.],
    [4., 5., 6., 7.],
    [8., 9., 10., 11.],
    [2., 1., 4., 3.],
    [1., 2., 3., 4.],
    [4., 3., 2., 1.]])
"""
```

```
X = torch.arange(12, dtype=torch.float32).reshape((3, 4))
Y = torch.tensor([[2.0, 1, 4, 3],
  [1, 2, 3, 4],
  [4, 3, 2, 1]])
torch.cat((X, Y), dim=0) # Join by row, this is similar to a SQL Union

"""
Returns:

  tensor([0., 1., 2., 3., 2., 1., 4., 3.],
    [4., 5., 6., 7., 8., 9., 10., 11.],
    [1., 2., 3., 4., 4., 3., 2., 1.]])
"""
```

1.2 Linear Algebra

After storing data into tensors and preprocessing them with basic mathematical operations, we need to user linear algebra to build our models. In this section, we will provide an overview of some linear algebra topics.

1.2.1 Scalars and Vectors

A scalar is a tensor with only one element and are known as zero order tensors! Like all other tensors, we can apply basic operations to it.

```
x = torch.tensor(3.0)
y = torch.tensor(2.0)
x + y, x * y, x / y, x**y

"""
Returns:

  (tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
"""
```

When vectors represent examples from a dataset, their values hold some real-world significance. For example, suppose we are studying heart attack risk, each vector might represent a patient and its components might correspond to their most recent vital signs (cholesterol levels, minutes of exercise per day, etc.). Vectors are implemented as first order tensors. Mathematically, we write vectors as *column vectors*

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

```
x = torch.arange(3)
x

"""
Returns:

  tensor([0, 1, 2])
"""
```

1.2.2 Matrices

Matrices are second order tensors. They can be written as

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

Matrix $A \in \mathbb{R}^{m \times n}$ is represented by a second order tensor with shape (m, n). a_{ij} is the element belonging to the i^{th} row and the j^{th} column.

We can also take the transpose of a matrix

$$A = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}$$

```
A = torch.arange(6).reshape(3, 2)
A.T

"""
Returns:

  tensor([[0, 2, 4],
    [1, 3, 5]])
"""
```

1.2.3 Tensors

Tensors give us a generic way to describe extentions to n^{th} order arrays. This becomes useful when we starting working with images. Each image is organized as a third order tensor with axes corresponding to height, weight, color channel. A collection of images then becomes a a fourth order tensor where each image are indexed along

the first axis! High order tensors are constructed similarly to vectors and matrices by growing the number of shape components

```
"""
# This is a third order tensor.  Think of it as a collection of matrices
"""
torch.arange(24) = torch.arange(6).reshape(2, 3, 4)
```

1.2.4 Tensor Operations

We can apply element wise operations against tensor elements.

```
A = torch.ones((2, 2))
B = A.clone() # A deep copy of tensor A
A, A + B

"""
Returns:
  (tensor([[1, 1],
    [1, 1]]),
  tensor([[2, 2],
    [2, 2]]))
"""
```

The elementwise product of two matrices is called a **Hadamard Product** (denoted as $\odot$)

$$A \odot B = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}$$

```
C = A + B
A * C

"""
Returns:
  tensor([[2, 2],
    [2, 2]]),
"""
```

Adding or multiplying a scalar by any other tensor creates a result with the same shape as the original tensor. Each element in the resultant tensor is added or multipled by the scalar.

```
a = 2
X = torch.ones((2, 2))
a + X, (a * X).shape

"""
Returns:
  (tensor([[3, 3],
    [3, 3]]),
  torch.Size([2, 2]))
"""
```

1.2.5 Reduction

There are a lot of times where we want to calculate the sum of a tensor's elements. We express the sum of elements in a vector x of length n as

$$\sum_{i=1}^{n} x_i$$

```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()

"""
Returns:
  (tensor([0., 1., 2.]), tensor(3.))
"""
```

To express sums over elements of tensors of arbitrary shape, we simply sum over all of the axies. For example, the sum of elements of an $x \times n$ matrix A could be written as

$$\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij} \qquad \text{for row i and column j}$$

```
x = torch.ones((2, 2))
x.shape, x.sum()

"""
Returns:
  (torch.Size([2, 2]), tensor(8.))
"""
```

1.2.6 Dot Products
1.2.7 Matrix-Vector Products
1.2.8 Matrix-Matrix Products
1.2.9 Norms
1.3 Calculus
1.3.1 Differentiation
1.3.2 Partial Derivatives and Gradients
1.3.3 Notations and Common Functions
1.4 Probability and Statistics
1.4.1 Random Variables