

REPORT

Name – **Ravi Tarun Prasad Nimmalapudi**

Course - **CPSC 5960 03 PyTorch Lab Design**

❖ What is PyTorch?

- PyTorch is an open-source deep learning framework developed by Meta AI in 2016. It is designed to be flexible and modular, catering both to research needs for quick experimentations and to industry requirements and make deployments scalable. This basically simplifies building, training and deploying deep learning models and is widely used in areas such as computer vision, natural language processing and reinforcement learning.
- Now let's quickly go through what deep learning is before we dig into how PyTorch is so effective in this field. So, deep learning can be called a branch of machine learning that uses artificial neural networks with many layers to learn from huge loads of data. In simple words we can think of a deep learning model as a human brain loaded with a network of "neurons" that processes data in layers. For example, let us consider an image recognition task, like identifying whether a given image is of a dog or a cat. Now, considering the deep learning model, the first few or 'n' layers might learn to recognize basic shapes and edges in the images. As we move deep through layers, the network tries to learn more complex features like facial shapes etc. Eventually combining all the learned stuff to make the prediction, such as classifying the given image as a dog or a cat.
- So, one of the really coolest things about PyTorch is it provides pre-trained models and deep learning models that have already been trained on large datasets and can be used a really good starting point for similar tasks, saving us time and computational resources which makes lives easy. By using these pre-trained models, you can access transfer learning, that is we can simply adapt these models to our specific needs/tasks with minimal additional training. I believe this plays a very important role when we have a smaller dataset, and since the model is pre-trained it can bring in its own knowledge and with very little tuning for our specific task it can yield pretty good results. For instance, PyTorch's "torchvision" library offers a variety of pre-trained models for tasks like image classification, object detection etc. Some of the popular models include architectures like ResNet and VGG which already are available trained on massive ImageNet dataset, meaning they already know a lot of information about general image features.

❖ Why PyTorch?

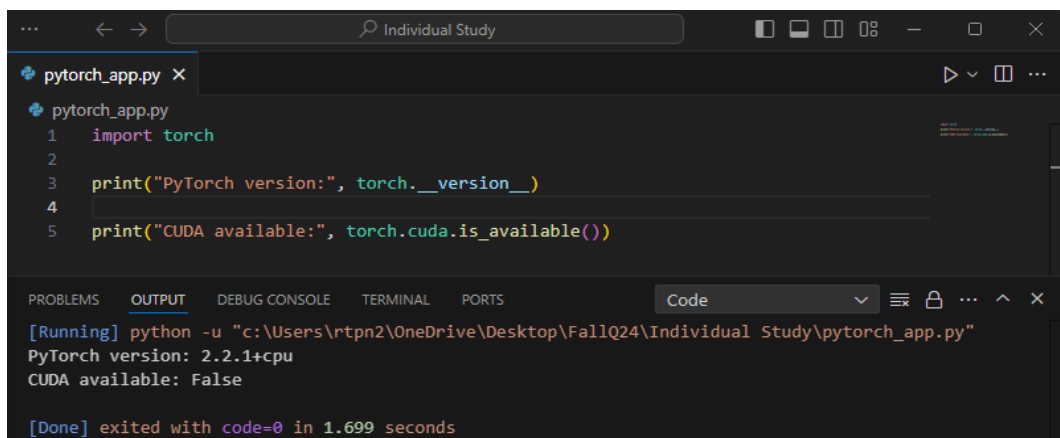
- PyTorch is a highly popular deep learning framework that has been widely adopted across both academia and industry. In September 2022, Meta transferred PyTorch to the PyTorch Foundation which is managed under the Linux Foundation. This emphasizes PyTorch's commitment to an open-source, vendor neutral ecosystem, with a mission to make AI tools and resources widely accessible to everyone.
- Leading businesses like Meta, Tesla, OpenAI, Uber, Airbnb, Microsoft, and Toyota had already used PyTorch extensively. Its popularity comes from a combination of technical advantages and its python-friendly developer focused design. Many developers prefer PyTorch due to its "eager execution" mode, where code is evaluated as it's run, allowing for easy inspection and debugging.
- Some of key features of PyTorch:
 - Eager Execution (Dynamic Computation Graphs):
You can create and alter models in real time with PyTorch's dynamic graph building capability, which facilitates experimentation and debugging.
 - Python Design:
For python developers, PyTorch is very user friendly, resulting in code that is clean, straightforward and simple to maintain.
 - Available pre-trained models and Transfer Learning:
Many pre-trained models are available in PyTorch, especially through torchvision, which enables users to refine models without starting from scratch.
 - Huge Community and Industry Support:
PyTorch has a strong ecosystem and is expanding its resources and capabilities with the backing of major tech businesses and input from top AI firms.
 - Strong Research and Academic Presence:
PyTorch's flexible design has made it the top choice in academia, resulting in faster availability of SotA models and innovations accessible to the broader community.
State-of-the-Art models represent the highest-performing, most advanced models in machine learning and deep learning. They achieve results on benchmark tasks such as image classification, natural language processing and speech recognition utilizing the latest techniques and architectures.

❖ Installing PyTorch:

- To get started with PyTorch, we would need to install it on our system. There are multiple ways to install PyTorch, but I did set up a dedicated virtual environment specific to this application. This approach basically ensures a clean setup by isolating all dependencies. But as an alternative among those which I explored 'Google Colab' would be a great option which offers a free environment with GPU support as well.

A brief explanation of the setup:

- Firstly, create a virtual environment dedicated to PyTorch (in which we could include any other dependencies we would want to go further as per the application's need).
 - `python -m venv pytorchenv`
- Then we would activate the environment using:
 - `pytorchenv\Scripts\activate`
- Then by simply visiting the PyTorch website, we could find all the options available to install PyTorch onto our machine. Based on your selection, you will receive a command which you can run in our virtual environment created.
 - `pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118`
- Official website link : <https://pytorch.org>
- Now, let us verify that the PyTorch is installed by simply checking the version of the PyTorch and verifies if CUDA is available, which indicates if PyTorch can access the GPU (If we would have installed the CUDA-enabled version, we should specify at the time of installation itself)



```
... Individual Study
pytorch_app.py X
pytorch_app.py
1 import torch
2
3 print("PyTorch version:", torch.__version__)
4
5 print("CUDA available:", torch.cuda.is_available())

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Code
[Running] python -u "c:\Users\rtpn2\OneDrive\Desktop\FallQ24\Individual Study\pytorch_app.py"
PyTorch version: 2.2.1+cpu
CUDA available: False

[Done] exited with code=0 in 1.699 seconds
```

❖ Exploring PyTorch:

- **Tensors:**

Tensors are specialized data structures that are very similar to arrays and matrices. In PyTorch, we use tensors to encode inputs, outputs and model parameters. Unlike NumPy arrays, tensors can run on GPUs for faster computation and are optimized for automatic differentiation, making them essential for deep learning tasks. In deep learning, a 2D array is referred to as a matrix, and arrays with more than 2 dimensions are called tensors.

- **Why Tensors?**

To put it in simple words, these are essential in deep learning frameworks due to their efficiency and flexibility in handling large scale data, with unique advantages over NumPy arrays. While these are very similar to arrays, tensors can run on GPUs or any other accelerators, enabling faster computation speed which is very crucial for deep learning tasks. Therefore, their design allows for efficient data manipulation, batch processing, and seamless integration with PyTorch modules, which simplifies workflows and allows rapid experimentation with models, making tensors a powerful and preferred data structure in machine learning.

For example, when optimizing neural network weights, each weight is adjusted individually to see how it affects the overall loss, and because these updates are independent, they can be computed in parallel. Now, because GPUs have several cores built for high-speed computations, this parallelism is perfect for them. Tensors in PyTorch can leverage GPU acceleration, allowing these weight updates to be processed simultaneously across many cores. Tensor-based GPU computations are around 30 times quicker than CPU computations, and they are almost 60 times faster than NumPy array computations. This massive speed advantage makes tensors the preferred choice for deep learning tasks, where quick, efficient computation is critical.

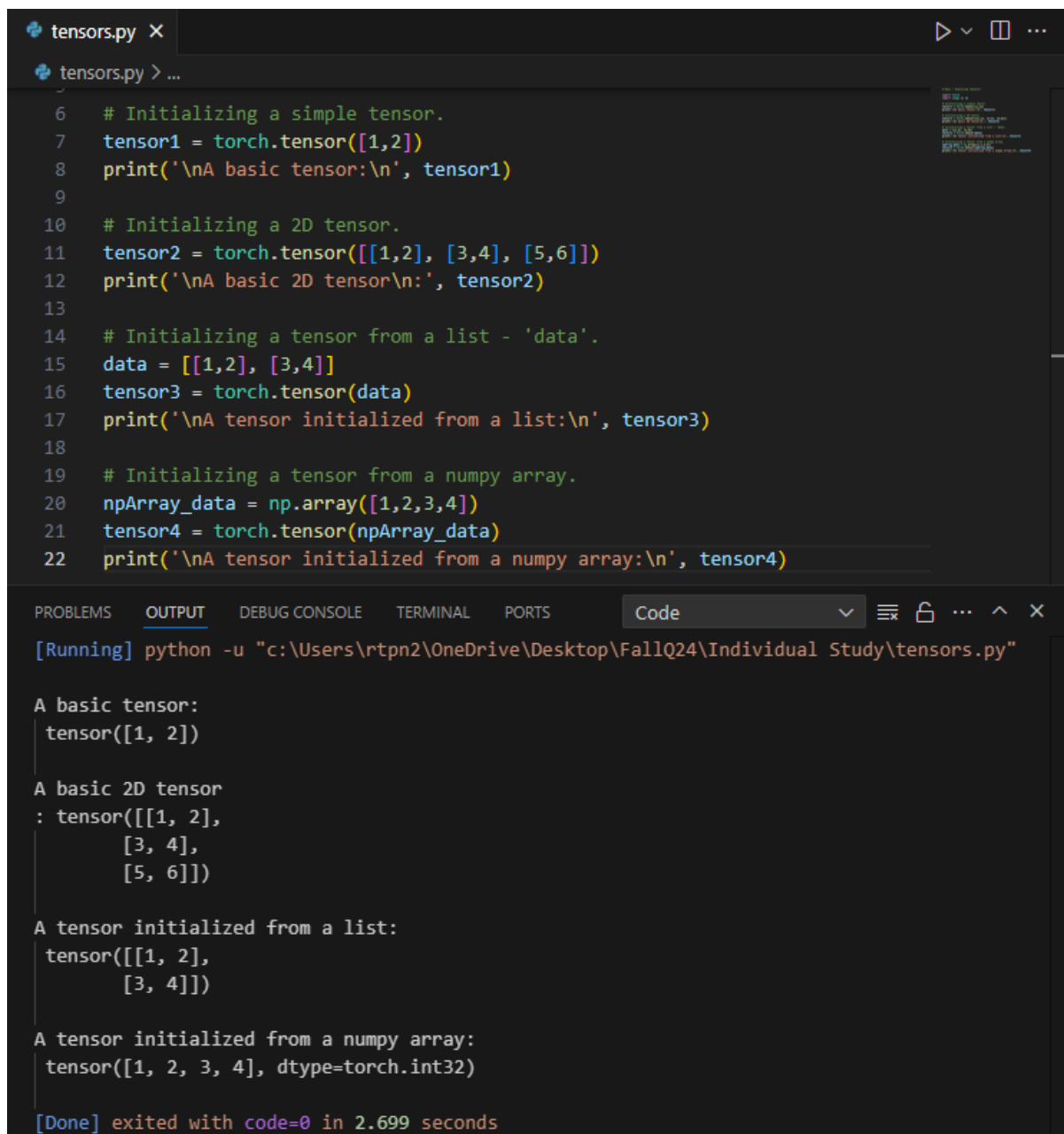
- **Exploring Tensors:**

To start, let us simply imagine them as containers for numbers which are very flexible. These containers are powerful because they can hold and manipulate large amounts of data quickly, especially when GPUs are involved for deep learning tasks. Tensors can hold data of various shapes and sizes, allowing us to perform mathematical operations easily across multiple elements.

- Initializing a basic tensor:

In PyTorch, tensors are created using `torch.tensor()`.

These tensors can be directly created from data and the datatype is automatically inferred or else assigned from a NumPy array.



```
tensors.py X
tensors.py > ...

6 # Initializing a simple tensor.
7 tensor1 = torch.tensor([1,2])
8 print('\nA basic tensor:\n', tensor1)
9
10 # Initializing a 2D tensor.
11 tensor2 = torch.tensor([[1,2], [3,4], [5,6]])
12 print('\nA basic 2D tensor\n:', tensor2)
13
14 # Initializing a tensor from a list - 'data'.
15 data = [[1,2], [3,4]]
16 tensor3 = torch.tensor(data)
17 print('\nA tensor initialized from a list:\n', tensor3)
18
19 # Initializing a tensor from a numpy array.
20 npArray_data = np.array([1,2,3,4])
21 tensor4 = torch.tensor(npArray_data)
22 print('\nA tensor initialized from a numpy array:\n', tensor4)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Code

[Running] python -u "c:\Users\rtpn2\OneDrive\Desktop\FallQ24\Individual Study\tensors.py"

A basic tensor:
tensor([1, 2])

A basic 2D tensor
: tensor([[1, 2],
[3, 4],
[5, 6]])

A tensor initialized from a list:
tensor([[1, 2],
[3, 4]])

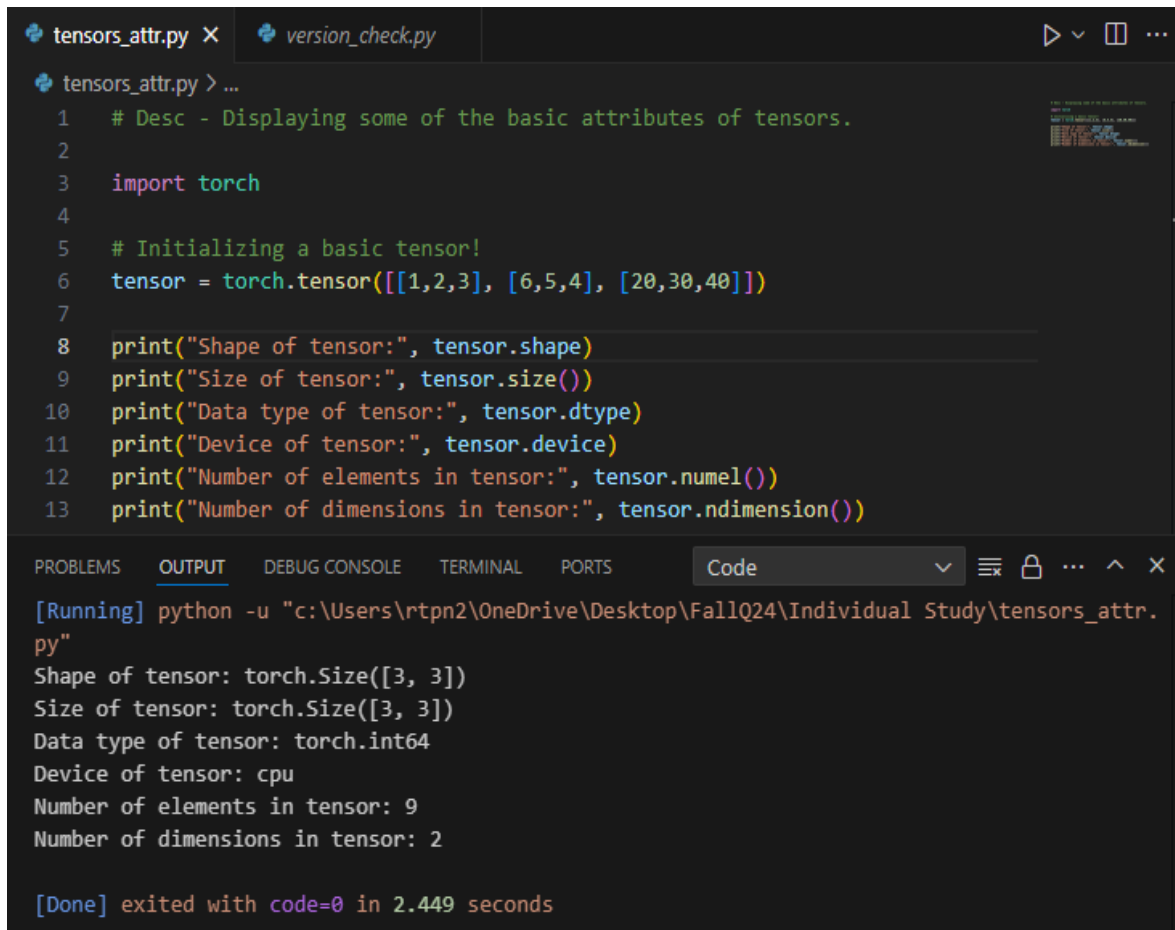
A tensor initialized from a numpy array:
tensor([1, 2, 3, 4], dtype=torch.int32)

[Done] exited with code=0 in 2.699 seconds

Also, we could specify a datatype while creating the tensor using `'dtype=torch.'datatype'`.

- Attributes of a tensor:

Tensor attributes describe their shape, datatype, size, device on which they are stored and many more.



```
tensors_attr.py X version_check.py
tensors_attr.py > ...
1 # Desc - Displaying some of the basic attributes of tensors.
2
3 import torch
4
5 # Initializing a basic tensor!
6 tensor = torch.tensor([[1,2,3], [6,5,4], [20,30,40]])
7
8 print("Shape of tensor:", tensor.shape)
9 print("Size of tensor:", tensor.size())
10 print("Data type of tensor:", tensor.dtype)
11 print("Device of tensor:", tensor.device)
12 print("Number of elements in tensor:", tensor.numel())
13 print("Number of dimensions in tensor:", tensor.ndim())
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Code

```
[Running] python -u "c:\Users\rtpn2\OneDrive\Desktop\FallQ24\Individual Study\tensors_attr.py"
Shape of tensor: torch.Size([3, 3])
Size of tensor: torch.Size([3, 3])
Data type of tensor: torch.int64
Device of tensor: cpu
Number of elements in tensor: 9
Number of dimensions in tensor: 2

[Done] exited with code=0 in 2.449 seconds
```

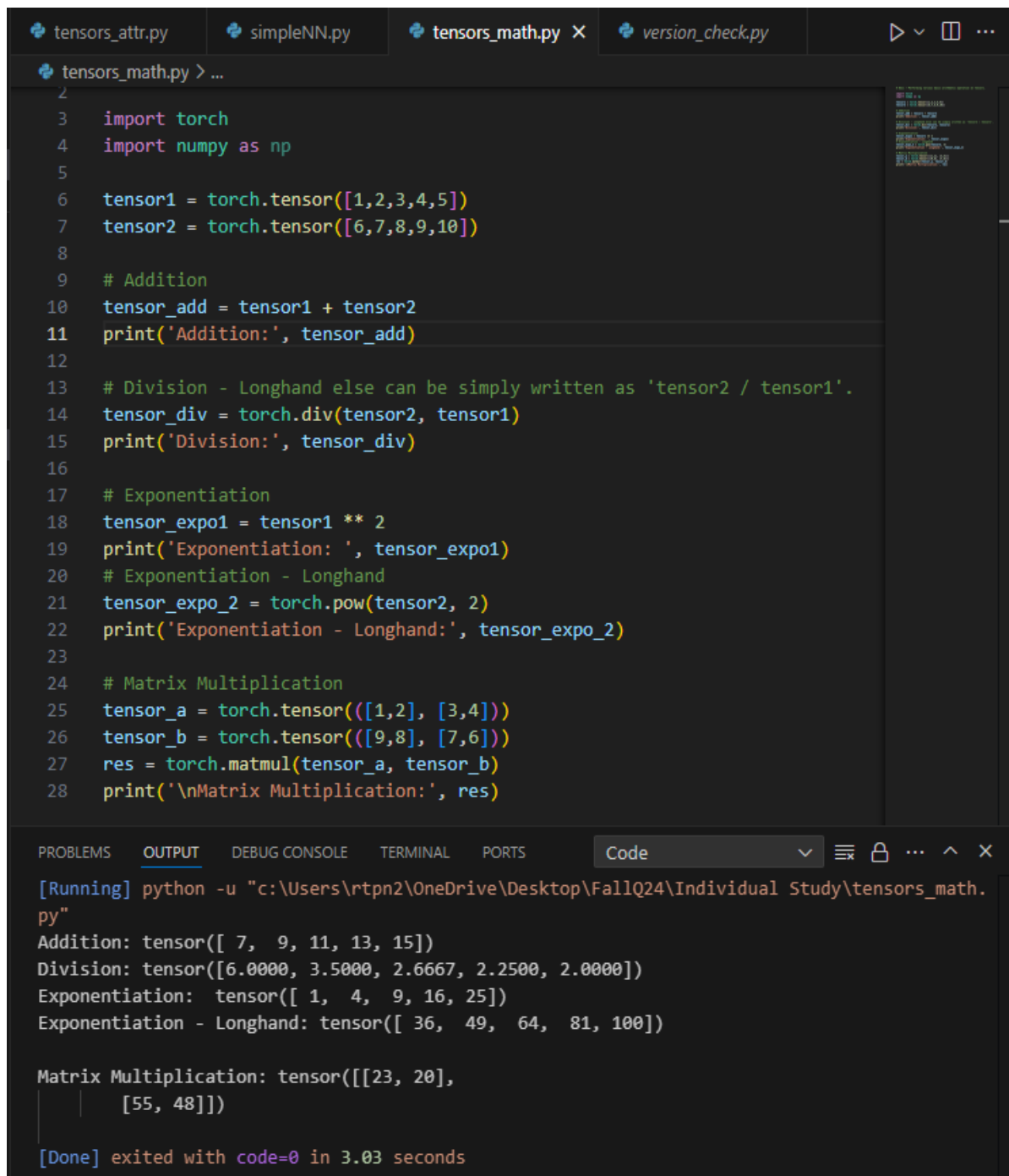
As we can see above, the shape and the size refer to the same concept, that is the dimensions of a tensor. Both describe the number of elements along each axis of the tensor. The only and the main difference is in how they are accessed, but eventually the information returned is 'same'.

Talking about the number of dimensions i.e. 'tensor.ndim()' – This tells us how many axes the tensor has (basically the rank). These ranks basically help define the type of operations and mathematical formulations that can be applied to a tensor.

However, it is essential to ensure that all elements in a tensor have the same shape and data type for consistency and to avoid errors when performing operations.

- Arithmetic operations on tensors:

In PyTorch, tensors support a variety of arithmetic operations, making it easy to manipulate data directly in tensor form. These operations include basic operations like element wise addition, subtraction, multiplication etc. Also, allows more complex operations like matrix multiplications and exponentiation.



The screenshot shows a code editor with four tabs: `tensors_attr.py`, `simpleNN.py`, `tensors_math.py` (active), and `version_check.py`. The `tensors_math.py` file contains the following Python code:

```
2
3 import torch
4 import numpy as np
5
6 tensor1 = torch.tensor([1,2,3,4,5])
7 tensor2 = torch.tensor([6,7,8,9,10])
8
9 # Addition
10 tensor_add = tensor1 + tensor2
11 print('Addition:', tensor_add)
12
13 # Division - Longhand else can be simply written as 'tensor2 / tensor1'.
14 tensor_div = torch.div(tensor2, tensor1)
15 print('Division:', tensor_div)
16
17 # Exponentiation
18 tensor_expo1 = tensor1 ** 2
19 print('Exponentiation: ', tensor_expo1)
20 # Exponentiation - Longhand
21 tensor_expo_2 = torch.pow(tensor2, 2)
22 print('Exponentiation - Longhand:', tensor_expo_2)
23
24 # Matrix Multiplication
25 tensor_a = torch.tensor([[1,2], [3,4]])
26 tensor_b = torch.tensor([[9,8], [7,6]])
27 res = torch.matmul(tensor_a, tensor_b)
28 print('\nMatrix Multiplication:', res)
```

The bottom panel of the editor shows the output of the code execution:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  Code
[Running] python -u "c:\Users\rtprn2\OneDrive\Desktop\FallQ24\Individual Study\tensors_math.py"
Addition: tensor([ 7,  9, 11, 13, 15])
Division: tensor([6.0000, 3.5000, 2.6667, 2.2500, 2.0000])
Exponentiation: tensor([ 1,  4,  9, 16, 25])
Exponentiation - Longhand: tensor([ 36,  49,  64,  81, 100])

Matrix Multiplication: tensor([[23, 20],
                               [55, 48]])
[Done] exited with code=0 in 3.03 seconds
```

❖ **Building a simple neural network model:**

- **What are neural networks?**

A neural network can be simply defined as a machine learning program, or model, that makes decisions in a manner somewhat similar to a human brain. These use processes that simulate the behavior of biological neurons, allowing the network to learn (identify patterns, weigh options, and make decisions). By analyzing the given training data, a neural network can improve its accuracy over time, making it an effective and essential tool in fields like artificial intelligence and computer science in general.

- **How do these work?**

These work by processing data through layers of interconnected nodes or can be called artificial neurons. The network starts with an input layer that receives raw data, which is then passed through one or more hidden layers where computations take place. The network's nodes each apply a mathematical function to the input they receive, modifying the output according to its weight and comparing it to a threshold value. The node becomes activated and transmits its output to the following layer if the result is over the threshold; else, it stays inactive. This process continues until the data reaches the output layer, where the network produces a prediction or classification.

- **Building the neural network:**

Now, we will be building a simple neural network model for binary classification task.

- Data:

We will be generating synthetic data for this simple demonstration.

Synthetic data is the data that is artificially created and nearly identical to real-world data but does not come from actual observations or measurements. In the context of machine learning, this data is often used for testing and training models.

Using the scikit-learn 'make_classification' technique, we would be producing this data.

- Model:

The model we are building is a simple neural network for a classification task. It would have 3 layers in total consisting of an input layer, a hidden layer and an output layer.

The model would eventually consist of the first layer with 10 features as input then the hidden layer with 50 neurons and each of the neurons will apply the ReLU activation function to this input and finally an output layer with 2 neurons, corresponding to the two binary classes.

So, speaking about activation functions and why particularly 'ReLU' for this problem, an activation function in a neural network is a mathematical function that introduces non-linearity into the model. Putting in simple terms without these functions we are simply limiting the ability of a neural network to learn complex patterns.

Also, in this specific case of binary classification problem, ReLU is a good choice because it can effectively capture complex patterns in the data and also it involves a simple threshold operation.

- Loss Function and Optimizer:

Firstly, the loss function or cost function measures how well (or poorly) the model is performing. Whereas optimizer can be defined as an algorithm that keeps on adjusting the model's parameters (weights, biases) to minimize the loss.

In our case, we are using 'CrossEntropyLoss' because it is specifically designed to work with class probabilities which aligns with the structure of our model and also penalizes incorrect predictions more effectively than simpler loss functions like mean squared error etc.

One of the reasons for choosing 'Adam' optimizer is its adaptiveness. Adam adapts with the learning rate for each parameter individually, making it particularly effective with noisy data, which is common with synthetic data.

- Training and Evaluation:

Now, we train the model for 100 epochs. In each epoch, we compute the loss, backpropagate the gradients, and update the weights using the optimizer. The progress of the training is displayed by printing the loss at regular intervals.

After training the model, we evaluate it on the test data. We simply calculate the accuracy of the model by comparing the predicted labels with the true labels.

- **A brief walkthrough of the code:**

- Importing required libraries:

```
# Importing required libraries.

import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import pandas as pd
import warnings
warnings.filterwarnings("ignore")
```

✓ 25.6s Python

The above code block imports all the required libraries for creating, training, and visualizing a machine learning model specifically a neural network using PyTorch, along with data handling and preprocessing functions from scikit-learn and pandas.

‘torch’: This is the core PyTorch library, used for manipulating tensors and building neural networks.

‘torch.nn’: Includes classes such as layers and activation functions that help in the construction and training of neural networks.

‘torch.optim’ includes optimizers for training neural networks.

‘make_classification’: For generating synthetic data.

‘train_test_split’: For splitting the data into training and test sets.

‘StandardScaler’: For standardizing features. Basically, helps in normalizing data which eventually improves model performance and stability.

‘matplotlib.pyplot’: For creating visualizations.

- Gathering Data:

```
# Generating synthetic data.  
  
X, y = make_classification(n_samples=1000, n_features=10, n_classes=2, random_state=42)
```

✓ 0.0s Python

'n_samples=1000': Indicates 1,000 data samples (basically rows) to be generated.

'n_features=10': Indicates each sample should have 10 features (columns).

'n_classes=2': Defining that there should be two classes (binary classification). Each of the samples generated will belong to one of these two classes.

'random_state=42': Controlling randomness.

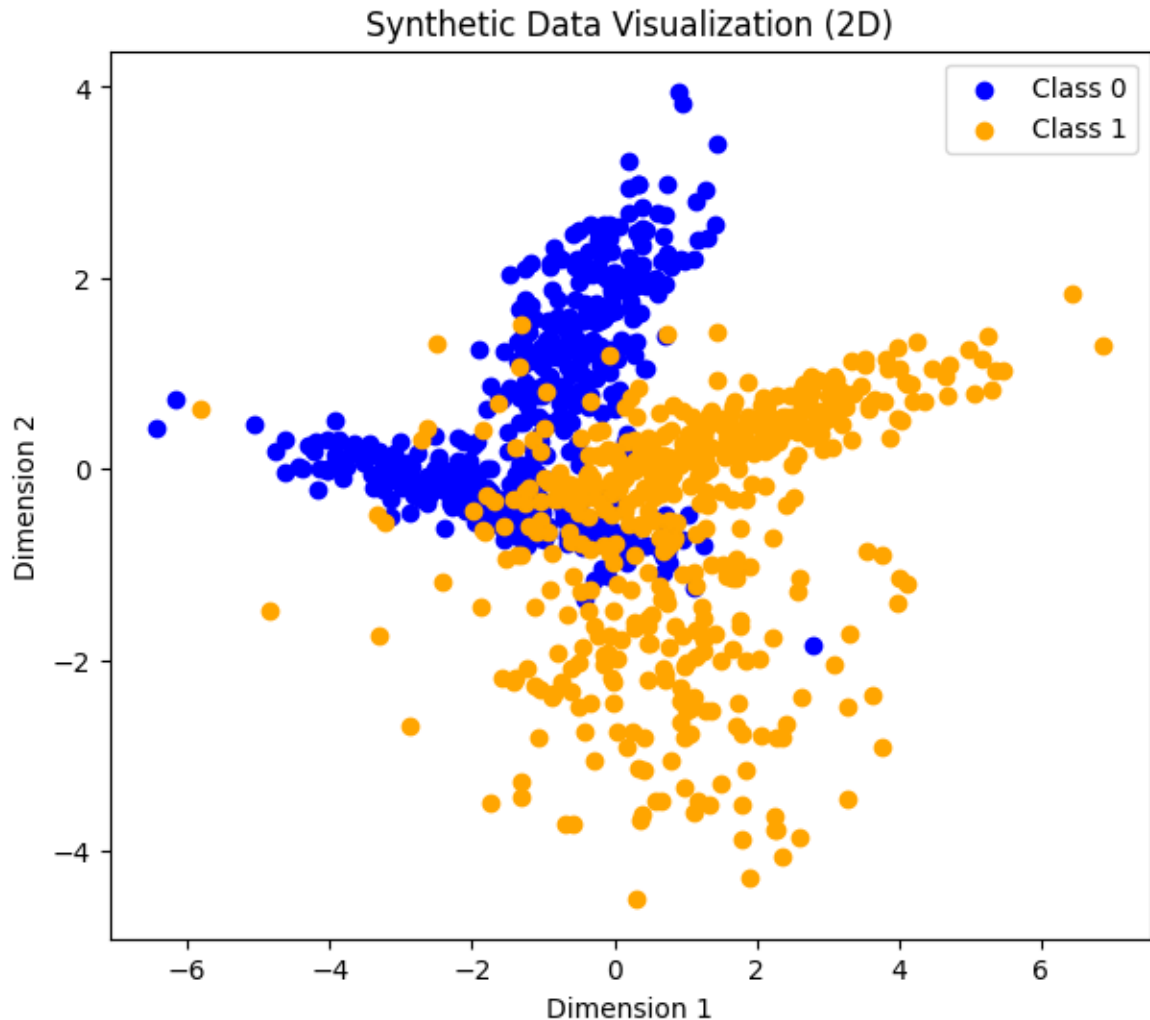
- Visualizing generated data:

```
# Visualizing the synthetic data.  
  
# Reducing the features to 2 dimensions for easy visualization.  
pca = PCA(n_components=2)  
X_reduced = pca.fit_transform(X)  
  
# Plotting a scatter plot.  
plt.figure(figsize=(7, 6))  
plt.scatter(X_reduced[y == 0, 0], X_reduced[y == 0, 1], color="blue", label="Class 0")  
plt.scatter(X_reduced[y == 1, 0], X_reduced[y == 1, 1], color="orange", label="Class 1")  
plt.xlabel("Dimension 1")  
plt.ylabel("Dimension 2")  
plt.title("Synthetic Data Visualization (2D)")  
plt.legend()  
plt.show()
```

✓ 0.3s Python

For visualizing the data, reducing the dimensionality. Dimensionality reduction is the process of transforming a dataset with many features into a smaller number of dimensions. The main goal is to **'simplify the data'**. Fewer dimensions make it easier to analyze, visualize and understand.

'PCA' is one of the most popular methods for dimensionality reduction. It basically finds the 'principal components' of the data – where data varies the most.



The above plot is a two-dimensional visualization of synthetic data generated for binary classification task with the data points reduced to two dimensions using 'PCA'.

Each dot represents a single data sample in the dataset projected onto a two-dimensional plane.

The points in the color 'blue' represent samples from 'Class 0' and the points in the color 'orange' represent samples from 'Class 1'.

There is some degree of separation between the two classes. However, there is also an overlap between the two classes, where both blue and orange points are mixed. This overlap suggests that the data may not be linearly separable might be pretty challenging for a simple classifier to achieve perfect accuracy.

- Splitting into train and test data:

```
# Splitting into train and test data.  
  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

✓ 0.0s

Python

Splitting the existing dataset into training data and testing data.

'test_size=0.25': Specifies 25% of the data to be included in the test set and the remaining 75% will be used for training.

- Standardizing the features:

```
# Standardizing the features to improve stability.  
  
scaler = StandardScaler()  
X_train = scaler.fit_transform(X_train)  
X_test = scaler.transform(X_test)
```

✓ 0.0s

Why standardize?

➔ Because standardizing helps improve convergence and reduces bias and brings stability in training.

- Converting them into PyTorch tensors:

```
# Converting into PyTorch tensors so can be used effectively in further model building.  
  
X_train = torch.tensor(X_train, dtype=torch.float32)  
X_test = torch.tensor(X_test, dtype=torch.float32)  
y_train = torch.tensor(y_train, dtype=torch.long)  
y_test = torch.tensor(y_test, dtype=torch.long)
```

✓ 0.0s

Python

Converting the NumPy arrays into PyTorch tensors with specified datatypes, the input test features to float(as we are working with continuous data) and the test ones to long.

- Defining the network:

```
# Defining the class.

class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.layer1 = nn.Linear(10, 50)
        self.layer2 = nn.Linear(50, 2)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        return self.layer2(x)
```

✓ 0.0s

As mentioned above, it is a simple model consisting of 3 layers including one input layer, one hidden layer and one output layer.

The 'SimpleNN' (our network) inherited from 'nn.Module' which is the base class for all PyTorch neural network modules.

The '`__init__()`' method defines the layers of the network.

The '`forward()`' method defines the forward flow of the data through the network.

- Initializing the model:

```
# Creating an instance of the model.

model = SimpleNN()
print(f"Model structure: {model}\n\n")
```

✓ 0.0s

```
Model structure: SimpleNN(
  (layer1): Linear(in_features=10, out_features=50, bias=True)
  (layer2): Linear(in_features=50, out_features=2, bias=True)
)
```

We basically did create an instance of our model which sets up the layers to accept the input data and allows us to use the model for training and also making predictions on unseen data later.

- Loss Function and Optimizer:

```
#Loss and Optimizer.  
  
criterion = nn.CrossEntropyLoss()  
optimizer = optim.Adam(model.parameters(), lr=0.01)  
✓ 3.2s
```

As we already talked about what are loss functions and why particularly 'CrossEntropyLoss()' for our problem statement. Now, let us talk about how it works.

So, the model outputs raw scores for each class and the criterion here, the 'CrossEntropyLoss' computes how far these raw scores are from the actual true labels. Given this the optimizer adjusts the model's weights to reduce this gap.

- Training the model:

```
# Training.  
  
for epoch in range(1, 101):  
    optimizer.zero_grad()  
    outputs = model(X_train)  
    loss = loss_criterion(outputs, y_train)  
    loss.backward()  
    optimizer.step()  
  
    if epoch % 10 == 0 or epoch == 100:  
        print(f"Epoch [{epoch}/100], Loss: {loss.item():.4f}")  
✓ 0.4s
```

Epoch [10/100], Loss: 0.3363
Epoch [20/100], Loss: 0.3013
Epoch [30/100], Loss: 0.2822
Epoch [40/100], Loss: 0.2666
Epoch [50/100], Loss: 0.2549
Epoch [60/100], Loss: 0.2424
Epoch [70/100], Loss: 0.2287
Epoch [80/100], Loss: 0.2126
Epoch [90/100], Loss: 0.1939
Epoch [100/100], Loss: 0.1737

In the above code block, we loop our model through 100 epochs for it to be trained. One 'epoch' is training our model on the entire dataset once. Also, this is important because choosing the correct number of epochs depends on how well the model learns.

The 'optimizer.zero_grad()' step above clears the gradients from the previous iterations making sure they don't accumulate because in PyTorch, gradients are accumulated across epochs, and we should clear them before every iteration or epoch in order to avoid noise or unintentional updates.

The next two steps involve calculating the loss and optimizing to minimize the loss. As we can see the output, every 10 epochs we could see the loss being reduced. This is called 'Loss Tracking' and could be useful in complex tasks we could stop early if the loss stops improving over number of epochs saving us time and resources.

- Evaluating the model:

```
# Evaluating.

with torch.no_grad():
    outputs = model(X_test)
    i, predictions = torch.max(outputs, 1)
    accuracy = (predictions == y_test).float().mean()
✓ 0.0s

# Testing.

print(f"Accuracy: {accuracy:.2f}")
✓ 0.0s

Accuracy: 0.84
```

In this phase, we are simply evaluating our trained model and testing it on unseen data. This is done by running the test data through the model to generate predictions, which are then compared with the labels.

Then calculating the accuracy by checking how many of these predictions made match the actual test labels and then finding the percentage of correct predictions.

As we can see the model is making correct predictions 84% of the time on new data which it hasn't seen before. While this is a promising result, there is plenty of room for improvement by tuning hyperparameters, adjusting the model architecture etc.

❖ References:

- <https://pytorch.org/>
- <https://pytorch.org/tutorials/beginner/basics/>
- <https://www.nvidia.com/en-us/glossary/pytorch/>
- <https://trainingportal.linuxfoundation.org/>
- <https://www.educative.io/answers/what-are-tensors>
- <https://medium.com/writeasilearn/why-do-we-need-tensor-objects-over-numpy-arrays-for-building-neural-network-6448a6f60e40>
- <https://machinelearningmastery.com/develop-your-first-neural-network-with-pytorch-step-by-step/>
- <https://www.ibm.com/topics/neural-networks>
- <https://github.com/pytorch/pytorch>