

[Skip to content](#)

## Chat history

You said:

can you help me with my CS 6375 project?

ChatGPT said:

Absolutely — I'd be glad to help! 

Can you tell me a bit more about your CS 6375 project?

For example:

- What's the topic or goal of the project (e.g., machine learning model, data analysis, etc.)?
- What stage are you currently in (idea, implementation, debugging, report, etc.)?
- Are there any specific requirements or datasets given by the course?

Once I know those, I can tailor the help — whether it's code, explanations, or write-up support.

You said:

can you give me details about the MNIST dataset and how to use it in python?

ChatGPT said:

Definitely — let's go over everything you need to know about MNIST and how to use it in Python, step by step.



### What is the MNIST Dataset?

MNIST (Modified National Institute of Standards and Technology) is one of the most famous benchmark datasets in machine learning, particularly for image classification and computer vision tasks.

- Purpose: Handwritten digit recognition (0–9)
- Number of samples: 70,000 total
  - 60,000 training images
  - 10,000 testing images
- Image size: 28×28 pixels (grayscale)
- Number of classes: 10 (digits 0–9)

- Pixel values: Each pixel has an intensity from 0 (white) to 255 (black)
- 

## 📦 How to Load MNIST in Python

There are several easy ways to load it depending on your library of choice:

---

### 1 Using TensorFlow / Keras (most common)

```
from tensorflow.keras.datasets import mnist # Load dataset (x_train,  
y_train), (x_test, y_test) = mnist.load_data() # Normalize pixel values  
x_train = x_train / 255.0 x_test = x_test / 255.0 print("Training data  
shape:", x_train.shape) print("Test data shape:", x_test.shape)
```

Output:

```
Training data shape: (60000, 28, 28)  
Test data shape: (10000, 28, 28)
```

### ✓ Example: Train a Simple Neural Network

```
from tensorflow.keras.models import Sequential from  
tensorflow.keras.layers import Flatten, Dense model = Sequential([  
Flatten(input_shape=(28, 28)), Dense(128, activation='relu'), Dense(10,  
activation='softmax') ]) model.compile(optimizer='adam',  
loss='sparse_categorical_crossentropy', metrics=['accuracy'])  
model.fit(x_train, y_train, epochs=5, validation_split=0.1)  
model.evaluate(x_test, y_test)
```

---

### 2 Using PyTorch

```
import torch from torchvision import datasets, transforms # Transform to  
tensor & normalize transform = transforms.Compose([ transforms.ToTensor(),  
transforms.Normalize((0.5,), (0.5,)) ]) train_data =  
datasets.MNIST(root='./data', train=True, download=True,  
transform=transform) test_data = datasets.MNIST(root='./data',
```

```
train=False, download=True, transform=transform) train_loader =  
torch.utils.data.DataLoader(train_data, batch_size=64, shuffle=True)  
test_loader = torch.utils.data.DataLoader(test_data, batch_size=64,  
                                         shuffle=False)
```

### ✓ Example: View a batch

```
images, labels = next(iter(train_loader)) print(images.shape) #  
torch.Size([64, 1, 28, 28])
```

## 3 Using `sklearn.datasets` (small version)

Scikit-learn provides a small 8×8 pixel version of MNIST called the “digits” dataset:

```
from sklearn.datasets import load_digits digits = load_digits()  
print(digits.images.shape) # (1797, 8, 8) print(digits.target.shape) #  
(1797,)
```

## Visualizing Samples

Here's a quick way to visualize a few images:

```
import matplotlib.pyplot as plt plt.figure(figsize=(5,5)) for i in  
range(9): plt.subplot(3,3,i+1) plt.imshow(x_train[i], cmap='gray')  
plt.title(f"Label: {y_train[i]}") plt.axis('off') plt.show()
```

## ⚙ Common Preprocessing Steps

Step	Description
Normalization	Scale pixels to [0, 1] or [-1, 1]
Flattening	Convert 28×28 → 784 vector for fully connected layers
One-hot encoding	Convert labels 0–9 → one-hot vectors (for categorical loss)

---

Augmentation

Random rotations, shifts, etc. to improve generalization

---



## Example Use Cases

- Digit classification with CNNs
  - Dimensionality reduction (PCA, t-SNE)
  - Autoencoders / Generative models (VAE, GANs)
  - Benchmarking new ML models
- 

Would you like me to show a CNN example for MNIST (using TensorFlow or PyTorch)?

That's a very common assignment for CS 6375-type courses.

You said:

also, how parallelized are pytorch models? like for random search validation si that already running them in parallel, or at least the models are using all cores in parallel?

ChatGPT said:

Excellent question — and you're thinking about the *right* performance details 🤗

Let's break this down carefully, because PyTorch's “parallelism” can refer to two different layers of parallelism:

---



### 1 Intra-model parallelism (inside a single model)

This is the kind PyTorch gives you automatically, out of the box.

#### ✓ What happens:

When you train a single model (e.g. one run of `model.fit()` or your `for epoch in ...` loop):

- PyTorch automatically uses multi-threaded CPU kernels (via OpenMP and MKL) for matrix operations (GEMM, convolutions, etc.).
- If you use a GPU, PyTorch runs the computations in parallel on thousands of CUDA cores.

- So yes — even *one* model run already uses parallelism under the hood.

You can check or limit CPU threading:

```
import torch print(torch.get_num_threads()) # see how many CPU threads are  
being used torch.set_num_threads(8) # manually set (optional)
```



## 2 Inter-model parallelism (between different model runs)

This is what you're referring to with random search, grid search, or cross-validation.

### ✗ By default:

PyTorch does not automatically parallelize *different model runs*.

So if you're looping through random hyperparameter combinations like:

```
for params in param_list: train_and_evaluate_model(params)
```

→ each model runs sequentially unless you explicitly parallelize it.



## How to Parallelize Random Search / Validation

You can use Python tools to parallelize across multiple processes (each training a different model):

- ◆ **Option 1:** concurrent.futures.ProcessPoolExecutor

```
from concurrent.futures import ProcessPoolExecutor def  
train_and_validate(params): # your PyTorch model training & evaluation  
here return accuracy with ProcessPoolExecutor() as executor: results =  
list(executor.map(train_and_validate, param_list))
```

Each process runs independently and can use its own cores or GPU if available.

- ◆ **Option 2: Scikit-learn's RandomizedSearchCV with PyTorch wrapper**

If your PyTorch model is wrapped as an sklearn estimator (using skorch or similar), you can use:

```

from sklearn.model_selection import RandomizedSearchCV
search = RandomizedSearchCV(estimator=model, param_distributions=params, n_iter=10,
                            n_jobs=-1)
search.fit(X, y)

```

The `n_jobs=-1` makes sklearn run multiple trials in parallel, using all CPU cores.

---

## 3 GPU note

If you're using a single GPU, parallel model training is limited — GPUs don't multitask well.

Instead, you'd typically:

- Train one model at a time on GPU.
  - Or, if you have multiple GPUs, use DistributedDataParallel (DDP) for multiple simultaneous runs.
- 

## Summary

Layer	What it Means	Happens Automatically?	How to Control
Intra-model (within one model)	Uses multithreaded CPU ops or GPU kernels	 Yes	<code>torch.set_num_threads()</code>
Inter-model (multiple models / random search)	Run multiple training jobs	 No	<code>multiprocessing</code> , <code>concurrent.futures</code> , or <code>RandomizedSearchCV(n_jobs=-1)</code>
Multi-GPU	Parallelize across GPUs	 No	<code>torch.nn.DataParallel</code> or <code>DistributedDataParallel</code>

---

Would you like me to show a minimal working example of how to run random search on MNIST models in parallel using `ProcessPoolExecutor` or `RandomizedSearchCV`?

You said:

if ther's already intra model parallelism, is there any use in inter model parallelism?

ChatGPT said:

That's a *very sharp* question — and it shows you're thinking like someone who actually

understands system-level efficiency 🤝

Let's unpack it carefully, because the answer depends on *what your bottleneck is* and *what you're trying to optimize*.

---

## 🧠 TL;DR

Yes — inter-model parallelism still matters, even if each model already uses intra-model (threaded or GPU) parallelism.

They operate at different levels and serve different goals.

---

## ⚙️ 1 Intra-model parallelism

*Parallelizing the math inside one training run.*

### What it does:

- Speeds up matrix multiplications, convolutions, and gradient updates.
- Uses all CPU cores (or all GPU CUDA cores).
- Achieves maximum efficiency for one model.

### What it doesn't do:

- It doesn't make trying *multiple* models faster — each model still trains one after another.
- You're still serially running different hyperparameters, architectures, or cross-validation folds.

So: intra-model parallelism is about *training one model efficiently*, not about *testing many models efficiently*.

---

## 2 Inter-model parallelism

*Parallelizing across multiple training runs.*

### What it does:

- Lets you train multiple models at the same time — e.g. for:
  - Hyperparameter search (grid/random search)
  - Cross-validation
  - Model ensembling
- Each model runs independently in its own process.

### Example benefit:

Let's say each model takes 10 minutes to train on your 8-core CPU.

If you try 8 hyperparameter sets sequentially, that's 80 minutes total.

If you run 8 in parallel (one per process), you can finish in  $\approx$ 10 minutes (plus overhead).

So, even though each process is internally multithreaded, you can *share the CPU cores* across multiple models — or better yet, assign each model a subset of cores.

---

## Tradeoffs and Real-World Scenarios

Situation	Best approach
You have only 1 GPU	Run one model at a time (inter-model parallelism doesn't help unless CPU-based).
You have multiple GPUs	Run different models on different GPUs (inter-model parallelism = big win).

---

You have lots of CPU cores

Combine both: intra-model threading (for efficient math) and inter-model multiprocessing (for multiple model runs).

---

You're doing random search / CV

Inter-model parallelism almost always helps — it's embarrassingly parallel.

---



### 3 Practical tip: Managing threads

Sometimes, if you *don't* limit threads per process, your inter-model runs can oversubscribe the CPU, making everything *slower*.

A good pattern is:

```
import torch
torch.set_num_threads(2) # limit intra-model threads # then
                        launch multiple processes in parallel
```

That way, if you have 8 cores and run 4 models at once, each gets ~2 threads.

---



### In summary

Parallelism				
Type	Level	Purpose	Automatic?	Example
Intra-model	Within one model	Speed up one training run	<input checked="" type="checkbox"/> Yes	PyTorch threads, CUDA
Inter-model	Between runs	Try many hyperparameters/models faster	<input type="checkbox"/> No	ProcessPoolExecutor, Ray, sklearn n_jobs=-1

👉 Use both when:

- You're doing hyperparameter tuning or cross-validation,
  - And your system has many CPU cores or multiple GPUs.
-

If you'd like, I can show you an example where we train multiple MNIST models with different hyperparameters in parallel, while keeping each model multithreaded internally — so you can see exactly how to balance both layers. Would you like that?

You said:

one more thing, for methods of a class do you start it with public methodName?

ChatGPT said:

Good question — this depends on the programming language you're using.

Here's how it works in the most common cases:

---

## In Python

No — you don't start methods with `public`, `private`, etc.

Python doesn't have explicit access modifiers.

You just define a method normally:

```
class MyClass: def my_method(self): print("Hello")
```

However, Python uses naming conventions to *signal* visibility:

Convention	Meaning	Example
<code>def my_method()</code>	Public	<code>obj.my_method()</code>
<code>def _helper()</code>	Protected (internal use)	<code>_</code> prefix is a <i>hint</i> to other developers
<code>def __secret()</code>	Private-ish (name mangling)	Interpreter renames it to <code>_MyClass__secret</code>

## Example:

```
class Example: def public_method(self): print("Public") def
    _internal_method(self): print("Intended for internal use") def
    __private_method(self): print("Name-mangled, not easily accessed") obj =
Example() obj.public_method() # Works obj._internal_method() # Works, but
                                # discouraged # obj.__private_method() # Fails
```

---

## In Java / C++ / C#

Yes — you do declare the access modifier.

```
class MyClass { public void myMethod() { } private void helperMethod() { }
```

Modifier	Meaning
<code>public</code>	Accessible from anywhere
<code>private</code>	Accessible only within the class
<code>protected</code>	Accessible within class and subclasses
(none)	Package-private (in Java)

## Summary

Language	Access keyword required?	Example
Python	 No (uses naming conventions)	<code>def my_method(self):</code>
Java / C++ / C#	 Yes	<code>public void myMethod() { }</code>

So if you're coding your CS 6375 project in Python, then no, you don't start methods  
with `public` — just `def`.

Would you like me to show how to structure a clean Python class with public and  
“private” helper methods (e.g., for a model training module)?

You said:

also you mentioned you can't parallelize GPUs, how many GPUs does google colab  
give?

ChatGPT said:

Excellent question — and this is a *really practical* one if you're planning to run parallel PyTorch or TensorFlow jobs in Google Colab.

Let's go over exactly how GPU access works there 

---

## 1 How many GPUs does Google Colab give you?

By default:

- Free Colab tier →  1 GPU at a time
- Colab Pro / Pro+ →  Still 1 GPU per runtime, but often a faster GPU and longer sessions

You cannot request or access multiple GPUs simultaneously in a single notebook runtime — even on paid tiers.

Colab's environment is sandboxed; each notebook gets 1 virtual machine (VM) that includes 1 GPU device (e.g., a Tesla T4, P100, V100, A100, or L4 depending on your tier and availability).

---

## 2 Checking your GPU in Colab

You can check what GPU you got with:

```
!nvidia-smi
```

Typical output (for example):

```
+-----+
| NVIDIA-SMI 530.30.02      Driver Version: 530.30.02      CUDA Version: 12.1
|
| GPU Name      Persistence-M| Bus-Id      Disp.A | Volatile Uncorr.
|               | Persistence |          |          |
| Fan Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute
|               |          |          |          |
+-----+
```

```
| =====+=====+=====+  
|     |  
|  0  Tesla T4      Off  | 00000000:00:04.0 Off |  
|     |  
+-----+-----+
```

You'll see only "GPU 0" — there's just one device.



### 3 What about running *multiple GPUs*?

You cannot in a single Colab runtime.

However, you could:

- Open multiple Colab notebooks (each with its own GPU) — but you'll quickly hit resource limits.
- Use Kaggle, Paperspace, RunPod, or Google Cloud (GCE) if you truly need multi-GPU or distributed setups.



### 4 Alternatives if you need more parallelism

Option	GPUs per runtime	Notes
Colab Free	1 (T4/K80, may vary)	Occasional disconnects, time-limited
Colab Pro	1 (T4, P100, L4, sometimes A100)	Priority access, longer sessions
Colab Pro+	1 (often A100 or L4)	Longest runtimes
Kaggle Notebooks	1 (T4 or P100)	Free, slightly slower
Google Cloud / AWS / Lambda Labs	Up to 8+ GPUs	You pay by hour; full control over parallelism

## Summary

Question	Answer
How many GPUs do you get in Google Colab?	1 per runtime
Can you use multiple GPUs at once?	 No (even in Pro/Pro+)
Can you run multiple Colabs at once?	 Yes, but limited by usage quotas
How to scale up?	Use distributed cloud compute (GCP, AWS, etc.)

If you tell me how much parallelism you actually need (e.g., number of models, runtime per model, GPU vs CPU), I can show you the most efficient way to structure it — whether it's batching, multiprocessing on CPU, or distributing over multiple GPUs.

You said:

can you give me details about the CIFAR dataset and how to use it in python?

ChatGPT said:

Absolutely  — the CIFAR dataset is another foundational computer vision dataset like MNIST, but more complex and realistic. Let's go over everything you need to know — from what it is, to how to use it in Python (TensorFlow or PyTorch).



## What is the CIFAR Dataset?

CIFAR stands for Canadian Institute For Advanced Research.

There are two main variants:

Dataset	#Classes	#Images	Image Size	Description
CIFAR-10	10	60,000 (50k train / 10k test)	32×32 RGB	General object categories (airplane, car, bird, etc.)

---

CIFAR-100	100	60,000 (50k train / 10k test)	32×32 RGB	Finer-grained classes (e.g. apple, aquarium fish, wolf)
-----------	-----	-------------------------------	-----------	---

Each image is small — 32×32 pixels with 3 color channels (RGB).

---

## CIFAR-10 Classes

Label	Class Name
0	airplane
1	automobile
2	bird
3	cat
4	deer
5	dog
6	frog
7	horse
8	ship
9	truck

---

## Using CIFAR in Python

You can load CIFAR-10 or CIFAR-100 directly from TensorFlow/Keras or PyTorch.

---



```

from tensorflow.keras.datasets import cifar10 import matplotlib.pyplot as
plt # Load data (x_train, y_train), (x_test, y_test) = cifar10.load_data()
# Normalize pixel values to [0,1] x_train, x_test = x_train / 255.0,
x_test / 255.0 print("Training data shape:", x_train.shape) print("Test
data shape:", x_test.shape)

```

### Output:

```

Training data shape: (50000, 32, 32, 3)
Test data shape: (10000, 32, 32, 3)

```

### Visualize a few samples:

```

plt.figure(figsize=(6,6)) for i in range(9): plt.subplot(3,3,i+1)
plt.imshow(x_train[i]) plt.title(y_train[i][0]) plt.axis('off') plt.show()

```

### Example: Simple CNN

```

from tensorflow.keras.models import Sequential from
tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
Dropout model = Sequential([ Conv2D(32, (3,3), activation='relu',
input_shape=(32,32,3)), MaxPooling2D((2,2)), Conv2D(64, (3,3),
activation='relu'), MaxPooling2D((2,2)), Flatten(), Dense(128,
activation='relu'), Dropout(0.5), Dense(10, activation='softmax') ])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy']) model.fit(x_train, y_train, epochs=10,
validation_split=0.1) model.evaluate(x_test, y_test)

```

## 2 PyTorch

```

import torch import torchvision import torchvision.transforms as
transforms # Define transforms (normalization is key) transform =
transforms.Compose([ transforms.ToTensor(), transforms.Normalize((0.5,
0.5, 0.5), (0.5, 0.5, 0.5)) ]) # Download dataset trainset =
torchvision.datasets.CIFAR10(root='./data', train=True, download=True,

```

```

transform=transform) testset = torchvision.datasets.CIFAR10(root='./data',
train=False, download=True, transform=transform) trainloader =
torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=64,
shuffle=False)

```

### Visualize a batch:

```

import matplotlib.pyplot as plt import numpy as np dataiter =
iter(trainloader) images, labels = next(dataiter) # Undo normalization
images = images / 2 + 0.5 npimg = images.numpy()
plt.imshow(np.transpose(npimg, (1, 2, 0))) plt.show()

```

### Example: Simple CNN model

```

import torch.nn as nn import torch.nn.functional as F class
Net(nn.Module): def __init__(self): super().__init__() self.conv1 =
nn.Conv2d(3, 32, 3, padding=1) self.pool = nn.MaxPool2d(2, 2) self.conv2 =
nn.Conv2d(32, 64, 3, padding=1) self.fc1 = nn.Linear(64 * 8 * 8, 128)
self.fc2 = nn.Linear(128, 10) def forward(self, x): x =
self.pool(F.relu(self.conv1(x))) x = self.pool(F.relu(self.conv2(x))) x =
torch.flatten(x, 1) x = F.relu(self.fc1(x)) x = self.fc2(x) return x net =
Net()

```

Then train and evaluate as usual with `torch.optim.Adam` or `SGD`.

## CIFAR-10 vs. MNIST

Feature	MNIST	CIFAR-10
Image size	28×28	32×32
Channels	1 (grayscale)	3 (RGB)
Classes	10 (digits)	10 (objects)

Difficulty	Easy	Moderate
Typical models	MLP, small CNN	CNN, ResNet, VGG, etc.

CIFAR-10 is often used as a benchmark for convolutional architectures like:

- ResNet (residual networks)
- VGGNet
- DenseNet
- EfficientNet

## ⚡ Summary

Aspect	CIFAR-10	CIFAR-100
Classes	10	100
Total images	60,000	60,000
Image size	32×32×3	32×32×3
Use case	Object classification	Fine-grained classification
Python access	<code>tf.keras.datasets.cifar10</code> or <code>torchvision.datasets.CIFAR10</code>	Similar for CIFAR-100

Would you like me to show how to add data augmentation (rotations, flips, color jitter) to CIFAR-10 for better generalization? That's often required to reach higher accuracy.

You said:

will i be able to reference these variables outside the if else statement?

```
if setType == "mnist":
    (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
else:
```

```
(xTrain, yTrain), (xTest, yTest) = cifar10.load_data()
```

ChatGPT said:

✓ Yes — you can reference those variables (`xTrain`, `yTrain`, `xTest`, `yTest`) outside the `if-else` block in Python.

Here's why:

---

## 🧠 Python's variable scoping rule

Python's if / else / for / while blocks do not create a new scope — only:

- functions,
- classes, and
- modules

create their own variable scopes.

So variables assigned inside an `if` or `else` are simply assigned in the current (enclosing) scope — usually the global or function scope.

---

## ✓ Example

```
setType = "mnist" if setType == "mnist": xTrain, yTrain = [1, 2, 3], [0, 1, 0] else: xTrain, yTrain = [4, 5, 6], [1, 1, 1] print(xTrain) # ✓ Works fine
```

Output:

```
[1, 2, 3]
```

---

## ⚠ Caution — If no branch runs

The only case you'd get an error is if no assignment occurs, e.g.:

```
if setType == "something": xTrain = ... # else block missing!
```

Then `xTrain` wouldn't exist if the condition isn't met → `NameError`.

So, as long as one branch *always* assigns the variables, you're safe.

---

✓ In your case:

```
if setType == "mnist": (xTrain, yTrain), (xTest, yTest) =  
    mnist.load_data() else: (xTrain, yTrain), (xTest, yTest) =  
    cifar10.load_data() # ✓ Accessible here print(xTrain.shape)
```

That will work perfectly — both branches assign the same variable names, so you can safely use them afterward.

---

Would you like me to show how to refactor this into a cleaner function (e.g.

`load_dataset(setType)`) so you can reuse it for MNIST, CIFAR-10, or CIFAR-100 easily?

You said:  
will this work fine?

```
from abc import ABC  
from tensorflow.keras.datasets import mnist, cifar10
```

```
class Model(ABC):  
    def retrieveDataset(self, validationIndex, setType):  
        if setType == "mnist":  
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()  
        else:  
            (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()  
  
        # normalizing them. Converting pixel values between 0 - 255 to 0 - 1  
        xTrain /= 255.0  
        xTest /= 255.0
```

```

xValid = xTrain[validationIndex:]
xTrain = xTrain[:validationIndex]

return (xTrain, xValid, yTrain, xTest, yTest)

# initialize method
def __init__(self):
    (self.mnistXTrain, self.mnistXValid, self.mnistYTrain, self.mnistXTest,
     self.mnistYTest) = self.retrieveDataset(50000, "mnist")
    (self.cifarXTrain, self.cifarXValid, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
        self.retrieveDataset(45000, "cifar")

```

ChatGPT said:

Nice — this is a clean start 

Your logic makes sense overall, but there are a few small but important details to fix so that your code runs correctly and cleanly.

Let's go through it line by line 

---

## The idea

You're:

1. Creating a base abstract class `Model`.
2. Writing a helper method `retrieveDataset` that loads MNIST or CIFAR-10, normalizes it, and splits off a validation set.
3. Storing each dataset's split as instance variables in `__init__`.

That's all good. 

---

## Issues and Fixes

1 `setType == "cifar"`

The correct identifier for CIFAR-10 is "cifar10", not "cifar".

Your code will currently fall into the `else` branch, which works, but it's clearer to be explicit:

```
if setType == "mnist": (xTrain, yTrain), (xTest, yTest) =  
    mnist.load_data() elif setType == "cifar10": (xTrain, yTrain), (xTest,  
    yTest) = cifar10.load_data() else: raise ValueError(f"Unknown dataset  
        type: {setType}")
```

This prevents silent errors or typos.

---

## 2 Label slicing — missing for `yTrain`

You sliced `xTrain` into `xTrain` and `xValid`, but you didn't split `yTrain`.

That means your validation set's images and labels will no longer match.

Fix:

```
xValid = xTrain[validationIndex:] yValid = yTrain[validationIndex:] xTrain  
= xTrain[:validationIndex] yTrain = yTrain[:validationIndex]
```

---

## 3 CIFAR label shape

For MNIST, labels are shaped `(n,)`, but for CIFAR-10, they come as `(n, 1)`.

You may want to flatten them for consistency:

```
yTrain = yTrain.reshape(-1) yValid = yValid.reshape(-1) yTest =  
yTest.reshape(-1)
```

Otherwise, you might get shape mismatches in training.

---

## 4 Validation index sanity check

If you ever pass a `validationIndex` larger than your training set (e.g., 60000 for MNIST), slicing will silently give you empty sets.

You might add a safety check:

```
if validationIndex > len(xTrain): raise ValueError(f"validationIndex {validationIndex} exceeds training size {len(xTrain)}")
```

---

## ✓ 5 Final fixed version

```
from abc import ABC from tensorflow.keras.datasets import mnist, cifar10
class Model(ABC): def retrieveDataset(self, validationIndex, setType): if
setType == "mnist": (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
elif setType == "cifar10": (xTrain, yTrain), (xTest, yTest) =
cifar10.load_data() else: raise ValueError(f"Unknown dataset type: {setType}")
# Normalize to [0,1] xTrain = xTrain.astype("float32") / 255.0
xTest = xTest.astype("float32") / 255.0 # Validation split if
validationIndex > len(xTrain): raise ValueError(f"validationIndex {validationIndex} exceeds training size {len(xTrain)}")
xValid = xTrain[validationIndex:]
yValid = yTrain[validationIndex:]
xTrain = xTrain[:validationIndex]
yTrain = yTrain[:validationIndex] # Flatten label
arrays (for consistency) yTrain = yTrain.reshape(-1)
yValid = yValid.reshape(-1)
yTest = yTest.reshape(-1)
return (xTrain, xValid,
yTrain, yValid, xTest, yTest)
def __init__(self):
(self.mnistXTrain, self.mnistXValid, self.mnistYTrain, self.mnistYValid, self.mnistXTest, self.mnistYTest) =
self.retrieveDataset(50000, "mnist")
(self.cifarXTrain, self.cifarXValid, self.cifarYTrain, self.cifarYValid, self.cifarXTest, self.cifarYTest) =
self.retrieveDataset(45000, "cifar10")
```

---

## ⊕ Improvements you might consider later

- Use `sklearn.model_selection.train_test_split()` for splitting (clearer, handles shuffling).
- Add an optional `shuffle=True` argument to randomize before validation split.

- Use class inheritance — e.g., make `MNISTModel` or `CIFARModel` extend `Model`.
- 

Would you like me to show how to extend this base class into a concrete model (say, a CNN that trains automatically once it's initialized)?

You said:

when you are passing in an image to a neural network, do you typically flatten it from (width, height, colors) to (numPixels, colors)?

ChatGPT said:

Excellent question — and the answer depends on what type of neural network you're

using 

---

### General rule of thumb

Network Type	Input shape	Flatten?	Why
Fully Connected (Dense) Network	1D vector (e.g., (784,) for MNIST)	 Yes	Dense layers expect a flat vector of features
Convolutional Neural Network (CNN)	3D tensor (height, width, channels)	 No	CNNs operate <i>spatially</i> , so they need the 2D structure intact
Recurrent Neural Network (RNN) (used for sequences)	2D/3D (time steps, features)	 (usually)	The time structure is preserved instead of flattened

---

### Example 1 — MNIST (grayscale, 28×28)

- Raw shape from dataset: (28, 28)
- For Dense network, flatten:
- `xTrain = xTrain.reshape(-1, 28 * 28)`  
→ shape becomes (num\_samples, 784)
- For CNN, add a channel dimension instead:

- `xTrain = xTrain.reshape(-1, 28, 28, 1)`  
→ shape becomes `(num_samples, 28, 28, 1)`
- 



## Example 2 — CIFAR-10 (RGB, 32×32×3)

- Raw shape: `(32, 32, 3)`
  - For Dense:
    - `xTrain = xTrain.reshape(-1, 32 * 32 * 3)`  
→ `(num_samples, 3072)`
  - For CNN:
    - You keep it as-is:
  - `# Already (num_samples, 32, 32, 3)`
- 



## Why you don't flatten for CNNs

Flattening would destroy spatial relationships:

- The pixel at `(5, 5)` and `(5, 6)` are neighbors — CNNs exploit that using filters.
  - If you flatten, the model just sees a long list of pixels, losing that local structure.
- 

## Summary

Dataset	Model Type	Input Shape
MNIST	Dense NN	<code>(num_samples, 784)</code>
MNIST	CNN	<code>(num_samples, 28, 28, 1)</code>
CIFAR-10	Dense NN	<code>(num_samples, 3072)</code>
CIFAR-10	CNN	<code>(num_samples, 32, 32, 3)</code>

---

Would you like me to show you how to automatically reshape your datasets in your existing `retrieveDataset()` method based on whether it's for a CNN or a dense network?

You said:

will this code work for the cifar since it has RGB instead of one mono color layer?

```
from abc import ABC
from tensorflow.keras.datasets import mnist, cifar10

class Model(ABC):
    def retrieveDataset(self, validationIndex, setType):
        if setType == "mnist":
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
        else:
            (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

        # normalizing them. Converting pixel values between 0 - 255 to 0 - 1
        xTrain /= 255.0
        xTest /= 255.0

        xValid = xTrain[validationIndex:]
        xTrain = xTrain[:validationIndex]

        yValid = yTrain[validationIndex:]
        yTrain = yTrain[:validationIndex]

    return (xTrain, yTrain, xValid, yValid, xTest, yTest)

    # initialize method
    def __init__(self):
        (self.mnistXTrain, self.mnistYTrain, self.mnistXValid, self.mnistYValid,
         self.mnistXTest, self.mnistYTest) = self.retrieveDataset(50000, "mnist")
```

```
(self.cifarXTrain, self.cifarYTrain, self.cifarXValid, self.cifarYValid, self.cifarXTest,  
self.cifarYTest) = self.retrieveDataset(45000, "cifar")
```

ChatGPT said:

✓ You're very close — and yes, that code will mostly work for both MNIST and CIFAR-10, but there's one key adjustment you'll want to make to handle the difference in color channels correctly.

Let's break it down 👇

---

### 🧠 The issue

- MNIST images are grayscale, shape: (28, 28)
- CIFAR-10 images are RGB, shape: (32, 32, 3)

Your code treats both the same way, which means:

- For CIFAR-10, it'll work fine (already 3D per image).
  - For MNIST, you'll likely get an error later when feeding into a CNN, because MNIST lacks the color/channel dimension.
- 

### ✓ Simple Fix

Add a line to expand MNIST's dimensions to include the channel axis (1 channel instead of 3):

```
if setType == "mnist": (xTrain, yTrain), (xTest, yTest) =  
    mnist.load_data() # Add channel dimension → (28, 28, 1) xTrain =  
    xTrain[..., None] xTest = xTest[..., None] else: (xTrain, yTrain), (xTest,  
    yTest) = cifar10.load_data()
```

That [...] syntax adds a final dimension.

---

### ✓ Final version

```

from abc import ABC from tensorflow.keras.datasets import mnist, cifar10
class Model(ABC): def retrieveDataset(self, validationIndex, setType): if
setType == "mnist": (xTrain, yTrain), (xTest, yTest) = mnist.load_data() #
Add color channel dimension xTrain = xTrain[..., None] xTest = xTest[...,
None] else: (xTrain, yTrain), (xTest, yTest) = cifar10.load_data() #
Normalize pixel values between 0 and 1 xTrain = xTrain.astype("float32") /
255.0 xTest = xTest.astype("float32") / 255.0 # Split validation set
xValid = xTrain[validationIndex:] yValid = yTrain[validationIndex:] xTrain
= xTrain[:validationIndex] yTrain = yTrain[:validationIndex] return
(xTrain, yTrain, xValid, yValid, xTest, yTest) def __init__(self):
(self.mnistXTrain, self.mnistYTrain, self.mnistXValid, self.mnistYValid,
self.mnistXTest, self.mnistYTest) = self.retrieveDataset(50000, "mnist")
(self.cifarXTrain, self.cifarYTrain, self.cifarXValid, self.cifarYValid,
self.cifarXTest, self.cifarYTest) = self.retrieveDataset(45000, "cifar")

```

---

## ✓ Output shapes

After this:

Dataset	Training shape	Validation shape	Test shape
MNIST	(50000, 28, 28, 1)	(10000, 28, 28, 1)	(10000, 28, 28, 1)
CIFAR-10	(45000, 32, 32, 3)	(5000, 32, 32, 3)	(10000, 32, 32, 3)

So you can now safely feed both into CNNs or dense networks (after flattening, if needed).

---

Would you like me to show a quick snippet that automatically flattens or keeps the data shaped correctly depending on whether you're training a CNN or MLP model?

You said:

will the division work correctly though? does it just take the deepest element and divide that?

ChatGPT said:

Ah — yes, this is an important detail about how NumPy / TensorFlow handle division.

Let's clarify.

---

## What happens here

```
xTrain /= 255.0
```

- `xTrain` is a NumPy array (or a TensorFlow tensor if you convert it).
- Python / NumPy / TensorFlow treat `/` as element-wise division.

That means every single pixel value is divided by `255.0` — not just the “deepest element” or the first element.

---

## Example

```
import numpy as np x = np.array([[0, 128, 255]]) x = x / 255.0 print(x)
```

Output:

```
[[0.0, 0.5019608, 1.0]]
```

 Every pixel is normalized individually.

---

## Important notes

1. Make sure `xTrain` and `xTest` are floating point arrays before division, otherwise integer division may cause rounding issues:

```
xTrain = xTrain.astype("float32") xTest = xTest.astype("float32") xTrain  
/= 255.0 xTest /= 255.0
```

2. Works for any shape, including `(num_samples, 28, 28, 1)` for MNIST or `(num_samples, 32, 32, 3)` for CIFAR — the division is applied element-wise across the whole array.
-

## ✓ Summary

- `xTrain /= 255.0` divides every pixel by 255.
  - You don't need to loop over images or channels.
  - Always cast to float first to avoid integer rounding.
- 

If you want, I can show a one-line robust normalization that works for both MNIST and CIFAR automatically, including ensuring correct dtype and shape for CNNs. Do you want me to do that?

You said:  
how do you use MLP with pytorch?

ChatGPT said:

Absolutely — I can walk you through using an MLP (Multilayer Perceptron) in PyTorch from start to finish, including dataset handling, model definition, training, and evaluation.

I'll keep it clear and practical.

---

## 1 Import Libraries

```
import torch import torch.nn as nn import torch.optim as optim from  
torch.utils.data import DataLoader, TensorDataset
```

---

## 2 Prepare the dataset

Assume you're using MNIST or CIFAR-10 and have them loaded as NumPy arrays:

```
import numpy as np from tensorflow.keras.datasets import mnist # Load  
MNIST (xTrain, yTrain), (xTest, yTest) = mnist.load_data() # Flatten  
images for MLP: (num_samples, 28*28) xTrain = xTrain.reshape(-1,  
28*28).astype("float32") / 255.0 xTest = xTest.reshape(-1,  
28*28).astype("float32") / 255.0 # Convert labels to torch tensors yTrain  
= torch.from_numpy(yTrain).long().squeeze() yTest =  
torch.from_numpy(yTest).long().squeeze() # Convert images to torch tensors  
xTrain = torch.from_numpy(xTrain) xTest = torch.from_numpy(xTest) # Create
```

```
PyTorch datasets and loaders train_dataset = TensorDataset(xTrain, yTrain)
test_dataset = TensorDataset(xTest, yTest) train_loader =
DataLoader(train_dataset, batch_size=64, shuffle=True) test_loader =
DataLoader(test_dataset, batch_size=64, shuffle=False)
```

✓ Note: We flatten the images for MLP input. For CNNs, you'd keep the 2D shape with channels.

---

## 3 Define the MLP Model

```
class MLP(nn.Module): def __init__(self, input_size, hidden_sizes,
output_size): super(MLP, self).__init__() layers = [] in_size = input_size
# Hidden layers for h in hidden_sizes: layers.append(nn.Linear(in_size,
h)) layers.append(nn.ReLU()) in_size = h # Output layer
layers.append(nn.Linear(in_size, output_size)) self.net =
nn.Sequential(*layers) def forward(self, x): return self.net(x)
```

Example instantiation for MNIST:

```
input_size = 28*28 hidden_sizes = [256, 128] # two hidden layers
output_size = 10 # 10 classes model = MLP(input_size, hidden_sizes,
output_size)
```

---

## 4 Set Loss and Optimizer

```
criterion = nn.CrossEntropyLoss() # for multi-class classification
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

---

## 5 Training Loop

```
num_epochs = 5 for epoch in range(num_epochs): model.train() running_loss =
0.0 for images, labels in train_loader: # Forward pass outputs =
model(images) loss = criterion(outputs, labels) # Backward pass
optimizer.zero_grad() loss.backward() optimizer.step() running_loss +=
```

```
loss.item() print(f"Epoch [{epoch+1}/{num_epochs}], Loss:  
{running_loss/len(train_loader):.4f}")
```

---

## 6 Evaluation

```
model.eval() correct = 0 total = 0 with torch.no_grad(): for images,  
    labels in test_loader: outputs = model(images) _, predicted =  
        torch.max(outputs, 1) total += labels.size(0) correct += (predicted ==  
labels).sum().item() print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

---

### ⚡ Notes

1. Flatten images for MLP; CNNs keep 2D+channel shape.
  2. CrossEntropyLoss expects class indices (0 to `num_classes-1`) — no one-hot encoding needed.
  3. Batch size, hidden layers, and learning rate are tunable hyperparameters.
  4. Can be extended to CIFAR-10 by flattening `(32*32*3)` per image.
- 

If you want, I can also show a full example using CIFAR-10 with PyTorch MLP, including normalization and RGB handling — which is slightly trickier than MNIST.

Do you want me to do that?

You said:

how can i use `torch.utils.data.random_split`?

ChatGPT said:

Excellent — `torch.utils.data.random_split()` is a very handy function for splitting a dataset into training, validation, and test subsets directly in PyTorch.

Let's break it down clearly 

---

## 🧠 Purpose

```
torch.utils.data.random_split(dataset, lengths, generator=None)
```

splits a dataset (any PyTorch `Dataset` object) into non-overlapping new datasets of the specified lengths.

---

## ✓ Basic Example

```
from torch.utils.data import random_split, DataLoader, TensorDataset
import torch # Example dataset x = torch.randn(1000, 28*28) # 1000
samples, 784 features y = torch.randint(0, 10, (1000,)) # 1000 labels
dataset = TensorDataset(x, y) # Split into 80% train, 20% validation
train_size = int(0.8 * len(dataset)) valid_size = len(dataset) -
train_size train_dataset, valid_dataset = random_split(dataset,
[train_size, valid_size]) print(len(train_dataset), len(valid_dataset)) #
e.g. 800, 200 # Create DataLoaders train_loader =
DataLoader(train_dataset, batch_size=64, shuffle=True) valid_loader =
DataLoader(valid_dataset, batch_size=64)
```

✓ Each subset is now a valid PyTorch `Dataset`, so you can plug it directly into `DataLoader`.

---

## ⚙️ Deterministic Split (for reproducibility)

You can make the split reproducible by setting a random seed:

```
generator = torch.Generator().manual_seed(42) train_dataset, valid_dataset
= random_split(dataset, [train_size, valid_size], generator=generator)
```

This ensures you get the same split every run.

---

## ✳️ Example with CIFAR-10 or MNIST

If you're using the built-in datasets:

```
from torchvision import datasets, transforms from torch.utils.data import
DataLoader, random_split transform =
```

```

transforms.Compose([transforms.ToTensor()]) # Load CIFAR-10 training set
(50,000 images) full_train_dataset = datasets.CIFAR10(root='./data',
train=True, transform=transform, download=True) # Split into
train/validation train_size = 45000 val_size = 5000 generator =
torch.Generator().manual_seed(42) train_dataset, val_dataset =
random_split(full_train_dataset, [train_size, val_size],
generator=generator) train_loader = DataLoader(train_dataset,
batch_size=64, shuffle=True) val_loader = DataLoader(val_dataset,
batch_size=64, shuffle=False)

```

Then use `train_loader` and `val_loader` in your training loop.

## ⚡ Summary

Step	Description
1	Wrap your tensors or dataset in a <code>TensorDataset</code> or <code>Dataset</code>
2	Use <code>random_split(dataset, [len1, len2, ...])</code>
3	Optionally use <code>generator=torch.Generator().manual_seed(SEED)</code> for reproducibility
4	Create <code>DataLoader</code> s for each subset

Would you like me to show how to split a dataset into train/validation/test (3 subsets) in one go using `random_split` too?

You said:  
how do you typically do random search in python?

ChatGPT said:

Excellent question — doing random search for hyperparameter tuning in Python can be done a few different ways, depending on whether you want to roll your own simple version or use a library like scikit-learn or Ray Tune.

Let's go step-by-step 

---

## What is Random Search?

Instead of testing every possible combination of hyperparameters (grid search), you:

- randomly sample combinations of hyperparameters,
- train/evaluate a model for each,
- and keep the best-performing one.

It's much faster and often just as good, especially when some parameters don't matter much.

---



## Option 1 — Simple Manual Random Search (for PyTorch, TensorFlow, etc.)

Here's a general template:

```
import random import numpy as np # Define hyperparameter search space
param_grid = { "learning_rate": [1e-4, 1e-3, 1e-2, 1e-1], "batch_size": [32, 64, 128], "num_layers": [1, 2, 3], "hidden_size": [64, 128, 256, 512] }
# Function to sample random combination def sample_params(grid): return {key: random.choice(values) for key, values in grid.items()} # Function to evaluate a model def evaluate_model(params): # Example: plug params into training function print(f"Training with params: {params}") # Replace with your actual model training + validation accuracy accuracy =
np.random.uniform(0.7, 0.99) return accuracy # Run random search best_acc = 0 best_params = None for i in range(10): # e.g. try 10 random combinations params = sample_params(param_grid) acc =
```

```
evaluate_model(params) if acc > best_acc: best_acc = acc best_params = params print(f"\nBest params: {best_params}, Accuracy: {best_acc:.4f}")
```

✓ This works for any model framework — PyTorch, TensorFlow, XGBoost, etc.

You just plug your actual training and validation logic into `evaluate_model()`.

---

## ⚙️ Option 2 — Using `RandomizedSearchCV` (**Scikit-learn**)

If your model is a scikit-learn estimator (like logistic regression, SVM, etc.):

```
from sklearn.model_selection import RandomizedSearchCV from
sklearn.ensemble import RandomForestClassifier from sklearn.datasets
import load_iris # Example data X, y = load_iris(return_X_y=True) # Model
model = RandomForestClassifier() # Define parameter distributions
param_distributions = { "n_estimators": [10, 50, 100, 200], "max_depth": [None, 5, 10, 20], "min_samples_split": [2, 5, 10] } # Random search
search = RandomizedSearchCV(model, param_distributions, n_iter=10, cv=3,
random_state=42) search.fit(X, y) print("Best parameters:", search.best_params_)
print("Best score:", search.best_score_)
```

✓ Automatically handles cross-validation and random sampling.

---

## ⚡ Option 3 — Parallel Random Search (for faster runs)

If training is slow, you can use multiprocessing or libraries like Ray Tune or Optuna to evaluate models in parallel.

Simple multiprocessing version:

```
import concurrent.futures def evaluate_model(params): # your training +
validation logic return params, np.random.uniform(0.7, 0.99) param_list =
[sample_params(param_grid) for _ in range(10)] with
concurrent.futures.ProcessPoolExecutor() as executor: results =
list(executor.map(evaluate_model, param_list)) best_params, best_acc =
```

```
max(results, key=lambda x: x[1]) print(f"Best params: {best_params},  
Accuracy: {best_acc:.4f}")
```

## Summary

Approach	When to Use	Pros
Manual random search	PyTorch / custom models	Full control, simple
RandomizedSearchCV	scikit-learn models	Easy CV + built-in
Parallel / Ray Tune / Optuna	Expensive models	Runs fast across cores/GPUs

Would you like me to show you a version of this random search that works directly with your PyTorch MLP training function (so you can test learning rates, batch sizes, etc.)?

You said:

can i just take any batch size and be fine without having to tune it?

ChatGPT said:

Excellent question — and it's one that even experienced ML practitioners debate.

The short answer is:

👉 You *can* usually pick a reasonable batch size and be fine, but it's not always optimal — and it can affect your model's training dynamics quite a bit depending on the situation.

Let's go over this clearly 👇

## What batch size actually does

Batch size controls how many samples your model processes before doing one gradient update.

Batch Size	Description	Pros	Cons
Small (e.g., 16–64)	Few samples per update	More noisy updates → may generalize better	Slower training (more updates)
Medium (e.g., 128–256)	Common default	Good balance between noise and speed	Usually fine
Large (e.g., 512–2048+)	Many samples per update	More stable gradients, faster per epoch	May overfit, require smaller LR

## Practical guideline (what most do)

Model type	Typical batch size
MLP on MNIST	64 or 128
CNN on CIFAR-10	128 or 256
Large models (ResNet, Transformers)	256–2048+ (with scaling tricks)

For most academic or course projects (like your CS 6375 work):

- 64 or 128 is a solid default.
- You rarely need to tune it aggressively.

## Relationship with learning rate

There *is* an interaction:

- Larger batch → more stable gradients → you can often increase learning rate a bit.
- Smaller batch → noisier gradients → use a smaller learning rate.

But unless you're doing state-of-the-art optimization, it's fine to just pick a standard learning rate like `0.001` for Adam and move on.

---

## Rule of thumb

Start with batch size = 64 or 128.

If training is slow → increase it.

If your model gets stuck or performs poorly → try smaller (like 32).

---

## Example

```
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)  
test_loader = DataLoader(test_dataset, batch_size=128)
```

You can keep that same value for both MNIST and CIFAR unless you run out of GPU memory.

---

## Summary

Aspect	Recommendation
Default choice	batch_size = 64 or 128
Don't need to tune aggressively	 True for most cases
Matters most	For large-scale or performance-critical work
Main effect	Changes training noise/speed trade-off

---

So yes — you can absolutely pick a standard batch size (like 128) and be totally fine for your project.

Would you like me to explain *what happens internally* in PyTorch when you change the batch size (i.e., how it affects gradient averaging and convergence)?

You said:

for MLP, does the input have to be a 2D array? cause for CIFAR, we also have to consider the RGB values

ChatGPT said:

Excellent question — and this is one of the key differences between using an MLP and a CNN on image data like CIFAR-10.

Let's break it down carefully 

---

## MLP input format

Yes — an MLP expects a 2D input tensor of shape:

(batch\_size, num\_features)

Each row represents one sample, flattened into a single vector of features.

So if you're using CIFAR-10, which has images of shape

(3, 32, 32) # (channels, height, width)

then you need to flatten each image into

(3 \* 32 \* 32) = 3072 features.

---

## Example (Flattening CIFAR for an MLP)

```
import torch from torchvision import datasets, transforms # CIFAR-10 has
3x32x32 images transform = transforms.Compose([ transforms.ToTensor(), #
converts to (C,H,W) with floats in [0,1] transforms.Lambda(x:
    x.view(-1)) # flatten to 3072-dim vector ]) train_dataset =
datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform) x, y = train_dataset[0] print(x.shape) #
torch.Size([3072])
```

Now each image is a 1D vector of 3072 pixels, where RGB channels are concatenated.

---

## Why flattening is necessary for MLPs

An MLP doesn't "see" spatial structure — it just treats every pixel value as an independent feature.

So for an image, flattening removes all the 2D spatial information.

This is why MLPs are weaker for vision tasks — they ignore locality and structure that CNNs handle well.

---

## CNNs vs MLPs (for context)

Model	Input shape	Keeps spatial info?	Typical use
MLP	(batch, num_features)	 No	Tabular data, small experiments
CNN	(batch, channels, height, width)	 Yes	Images, video frames

---

## MLP model example for CIFAR-10

```
import torch.nn as nn
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(3*32*32, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        )
    def forward(self, x):
        x = x.view(x.size(0), -1) # flatten batch dynamically
        return self.model(x)
```

## Summary:

- MLPs require input shape (batch\_size, num\_features).

- For CIFAR, you must flatten `(3, 32, 32) → (3072)`.
  - The RGB channels are just part of that flattened vector.
  - Works fine, but not as effective as a CNN for image data.
- 

Would you like me to show you how to compare that MLP's performance to a CNN on CIFAR-10 (same training loop, just different model architecture)? It's a great learning contrast.

You said:

so MLP always expected 1D (excluding the batch size)?

How about CNN?

ChatGPT said:

Exactly right  — you've got it.

Let's break this down clearly:

---

## MLP (Multilayer Perceptron)

### Input shape:

`(batch_size, num_features)`

That is:

- `batch_size` → number of samples per batch
- `num_features` → flattened input (so it's 1D per sample)

 So yes — MLPs always expect 1D input vectors per example, not counting the batch dimension.

If your data is an image, like `(3, 32, 32)`, you must flatten it:

```
x = x.view(x.size(0), -1)
```

which gives shape `(batch_size, 3072)` for CIFAR-10.

---

## CNN (Convolutional Neural Network)

### Input shape:

(batch\_size, channels, height, width)

So for CIFAR-10:

(batch\_size, 3, 32, 32)

and for MNIST:

(batch\_size, 1, 28, 28)

Unlike MLPs, you do NOT flatten images for CNNs.

Instead, you preserve the spatial structure — the `height` and `width` — because convolution layers exploit local patterns (edges, textures, etc.).

---

### Example: CNN expecting 4D input

```
import torch.nn as nn
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1), # (3,32,32) -> (32,32,32)
            nn.ReLU(),
            nn.MaxPool2d(2, 2), # (32,32,32) -> (32,16,16)
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2) # (64,16,16) -> (64,8,8)
        )
        self.fc_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64*8*8, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        )
    def forward(self, x):
        x = self.conv_layers(x)
        x = self.fc_layers(x)
        return x
```

Here the flattening only happens after the convolution and pooling layers — before the final fully connected layers.

---

## Summary

Model	Expected Input (per batch)	Flatten before model?	Keeps spatial structure?
MLP	(batch_size, num_features)	Yes	No
CNN	(batch_size, channels, height, width)	No	Yes

If you want, I can show you a small example where both an MLP and a CNN are trained on CIFAR-10 side-by-side to see the accuracy difference — it's dramatic and really illustrates why CNNs are better for images. Want me to?

You said:

how can i get the shape of an n-dimension array?

ChatGPT said:

Good question — that depends on what kind of array or tensor you're using, but it's very simple in all major libraries:

## 1. NumPy arrays

If you're using `numpy.ndarray`:

```
import numpy as np a = np.zeros((3, 32, 32)) print(a.shape) # (3, 32, 32)
```

`a.shape` returns a tuple of dimensions (number of elements along each axis).

You can access individual dimensions:

```
print(a.shape[0]) # 3 print(a.shape[1]) # 32
```

## 2. PyTorch tensors

If you're using a `torch.Tensor`:

```
import torch x = torch.randn(10, 3, 32, 32) print(x.shape) #  
torch.Size([10, 3, 32, 32]) print(x.size()) # same thing print(x.shape[1])  
# 3 (the channel dimension)
```

✓ `x.shape` and `x.size()` are equivalent.

✓ You can also use `x.ndim` to get the number of dimensions.

Example:

```
print(x.ndim) # 4
```



### 3. TensorFlow / Keras tensors

For `tf.Tensor` objects:

```
import tensorflow as tf x = tf.random.normal((10, 3, 32, 32))  
print(x.shape) # (10, 3, 32, 32) print(x.shape.rank()) # 4 (number of  
dimensions)
```

## 🔍 Summary

Library	Syntax	Example Output
NumPy	<code>arr.shape</code>	<code>(3, 32, 32)</code>
PyTorch	<code>tensor.shape</code> or <code>tensor.size()</code>	<code>torch.Size([10, 3, 32, 32])</code>
TensorFlow	<code>tensor.shape</code>	<code>(10, 3, 32, 32)</code>

Would you like me to show how to dynamically flatten an N-dimensional tensor in PyTorch (so it works for any input size automatically)?

You said:

when i call `mnist.load_data`, what data format is in?

ChatGPT said:

Excellent question — this is a common gotcha when working with Keras datasets like

```
mnist.load_data().
```

Let's go step-by-step 

---

## What `mnist.load_data()` returns

```
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

It returns two tuples — one for training and one for testing:

- `x_train`: images
  - `y_train`: labels
  - `x_test`: images
  - `y_test`: labels
- 

## Data shapes and types

Variable	Shape	Type	Description
<code>x_train</code>	(60000, 28, 28)	uint8	60k grayscale images (28×28)
<code>y_train</code>	(60000,)	uint8	Labels 0–9
<code>x_test</code>	(10000, 28, 28)	uint8	10k grayscale images
<code>y_test</code>	(10000,)	uint8	Labels 0–9

Each pixel in `x_train` or `x_test` is an integer from 0 to 255.

---

## Important details

### 1 Grayscale — not RGB

Each image is 2D (28×28), not `(28, 28, 1)`.

If your model expects a channel dimension (like CNNs), you'll need to add it manually:

```
x_train = x_train[..., None] # becomes (60000, 28, 28, 1) x_test =  
x_test[..., None]
```

---

## 2 Normalization (common preprocessing step)

Pixel values range from 0–255. It's standard to normalize to `[0, 1]` floats:

```
x_train = x_train.astype("float32") / 255.0 x_test =  
x_test.astype("float32") / 255.0
```

---

## 3 Label format

`y_train` and `y_test` are integers (0–9).

If you're using a model with softmax output, you can:

- Leave them as integers and use sparse categorical crossentropy, or
- Convert to one-hot vectors:  

```
from tensorflow.keras.utils import to_categorical y_train =  
to_categorical(y_train, 10) y_test = to_categorical(y_test, 10)
```

---

## ✓ In summary

After loading:

```
x_train.shape = (60000, 28, 28)  
y_train.shape = (60000,)  
x_test.shape = (10000, 28, 28)  
y_test.shape = (10000,)
```

Usually you'll do:

```
x_train = x_train.astype("float32") / 255.0 x_test =  
x_test.astype("float32") / 255.0 x_train = x_train[..., None] # add  
channel for CNNs x_test = x_test[..., None]
```

---

Would you like me to show the equivalent info for `cifar10.load_data()` too (the shapes are a bit different since it's RGB)?

You said:

like are they numpy arrays returned?

ChatGPT said:

✓ Exactly — yes, both MNIST and CIFAR datasets loaded via

```
tensorflow.keras.datasets return NumPy arrays.
```

---

## Example — MNIST

```
from tensorflow.keras.datasets import mnist (x_train, y_train), (x_test,  
y_test) = mnist.load_data() print(type(x_train)) # <class 'numpy.ndarray'>  
print(x_train.shape) # (60000, 28, 28) print(type(y_train)) # <class  
'numpy.ndarray'> print(y_train.shape) # (60000,)
```

So they're plain NumPy arrays containing image data and labels.

---

## CIFAR-10 too

```
from tensorflow.keras.datasets import cifar10 (x_train, y_train), (x_test,  
y_test) = cifar10.load_data() print(type(x_train)) # <class  
'numpy.ndarray'> print(x_train.shape) # (50000, 32, 32, 3)
```

Again, all NumPy arrays — the only difference is the shape:

- CIFAR includes the RGB channel (3),
  - MNIST is grayscale (no channel dimension).
- 

## So in short:

Dataset	Function	Data type	Typical shape
MNIST	<code>mnist.load_data()</code>	<code>numpy.ndarray</code>	<code>(60000, 28, 28)</code>
CIFAR-10	<code>cifar10.load_data()</code>	<code>numpy.ndarray</code>	<code>(50000, 32, 32, 3)</code>

You can directly use them with TensorFlow, PyTorch (after conversion with `torch.tensor()`), or plain NumPy processing.

Would you like me to show the quick way to convert those NumPy arrays into PyTorch datasets and dataloaders?

You said:

whatttt? so is it that the reutnred np arrays are integers?

---

```

UFuncTypeError           Traceback (most recent call last)
Cell In[1], line 16
    12     xTest /= 255.0
    14     print(xTrain.shape)
--> 16 (mnistXTrain, mnistYTrain, mnistXValid, mnistYValid, mnistXTest, mnistYTest) =
        retrieveDataset("mnist", 50000)
    17 (cifarXTrain, cifarYTrain, cifarXValid, cifarYValid, cifarXTest, cifarYTest) =
        retrieveDataset("cifar", 45000)

Cell In[1], line 11
    8     (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()
    10 # normalizing them. Converting pixel values between 0 - 255 to 0 - 1
        --> 11 xTrain /= 255.0
        12 xTest /= 255.0
        14 print(xTrain.shape)

```

UFuncTypeError: Cannot cast ufunc 'divide' output from dtype('float64') to dtype('uint8')  
with casting rule 'same\_kind'

ChatGPT said:

Ah — excellent catch! 😊

That's a classic NumPy type mismatch error — and you understood it exactly right:

Yes — the arrays returned by both `mnist.load_data()` and `cifar10.load_data()` are integers (`dtype=uint8`) with pixel values between 0–255.

---

## What's happening

When you do:

```
xTrain /= 255.0
```

NumPy tries to perform in-place division (`xTrain = xTrain / 255.0`),  
but `xTrain` has type `uint8`, which cannot store floating-point results.

So NumPy refuses to cast floats (from the division) back to `uint8`.

---

## The fix

You need to convert to `float32` before dividing:

```
xTrain = xTrain.astype('float32') / 255.0 xTest = xTest.astype('float32')  
/ 255.0
```

This changes the `dtype` to `float` and allows elementwise division to work safely.

---

## Example

```
from tensorflow.keras.datasets import mnist (xTrain, yTrain), (xTest,  
yTest) = mnist.load_data() print(xTrain.dtype) # uint8 xTrain =  
xTrain.astype("float32") / 255.0 print(xTrain.dtype) # float32
```

 Works fine — and you'll now get values between 0 and 1.

---

## Summary

Dataset	Returned dtype	Range	Fix
<code>mnist.load_data()</code>	<code>uint8</code>	0–255	<code>.astype('float32') / 255.0</code>
<code>cifar10.load_data()</code>	<code>uint8</code>	0–255	<code>.astype('float32') / 255.0</code>

So just change your code to this:

```
xTrain = xTrain.astype('float32') / 255.0 xTest = xTest.astype('float32')  
/ 255.0
```

and it'll work perfectly.

You said:

so after printing the shape, the last dimension for cifar is the colors. Would i have to swap it so that it comes before the width and height dimensions to pass into a CNN?

ChatGPT said:

 Excellent — that's exactly the right question to ask when moving between TensorFlow and PyTorch, because they use different image dimension orders.

Let's go step by step 

---

## TensorFlow / Keras format

By default, CIFAR-10 (and all `tensorflow.keras.datasets`) use the shape:

```
(batch_size, height, width, channels)
```

For CIFAR-10:

```
(50000, 32, 32, 3)
```

 TensorFlow CNN layers expect this format (`channels_last`).

So if you're using Keras / TensorFlow, you can pass it *as is*.

---

## 🔥 PyTorch format

PyTorch uses a different convention:

```
(batch_size, channels, height, width)
```

For CIFAR-10:

```
(50000, 3, 32, 32)
```

PyTorch CNNs (`nn.Conv2d`) expect this channels-first layout.

---

## ⚙️ Converting from TensorFlow format → PyTorch format

If you load CIFAR with `tensorflow.keras.datasets`, you'll get:

```
x_train.shape # (50000, 32, 32, 3)
```

You can reorder the axes (dimensions) using NumPy's `transpose`:

```
x_train = x_train.transpose(0, 3, 1, 2) x_test = x_test.transpose(0, 3, 1, 2)
```

Now:

```
x_train.shape # (50000, 3, 32, 32)
```

✓ Perfect for PyTorch's `Conv2d`.

---

## 🧠 Summary

Framework	Expected shape	Channel position	Need transpose?
TensorFlow / Keras	(batch, height, width, channels)	last	✗ No

---

PyTorch	(batch, channels, height, width)	first	<input checked="" type="checkbox"/> Yes
---------	----------------------------------	-------	---

---

## 🔍 Example (CIFAR ready for PyTorch CNN)

```
from tensorflow.keras.datasets import cifar10 import torch (x_train,  
y_train), (x_test, y_test) = cifar10.load_data() # Convert to float and  
normalize x_train = x_train.astype('float32') / 255.0 x_test =  
x_test.astype('float32') / 255.0 # Transpose to PyTorch format x_train =  
x_train.transpose(0, 3, 1, 2) x_test = x_test.transpose(0, 3, 1, 2) #  
Convert to tensors x_train = torch.tensor(x_train) y_train =  
torch.tensor(y_train.squeeze())
```

---

So in short:

Yes — if you're using PyTorch, you do need to swap the color (channel) dimension to come before height and width.

If you're using TensorFlow/Keras, you don't.

You said:

for a dataloader consisting of batches, do they have to have tensors as input?

ChatGPT said:

Excellent question 🤝 — this is one of those key details that really clarifies how PyTorch's `DataLoader` works under the hood.

Let's break it down clearly 👇



## Short answer

Yes — PyTorch `DataLoader` expects tensors, *eventually*.

But you don't have to give it tensors directly — it can convert or collate other formats (like NumPy arrays) *as long as you wrap them properly*.

---

## How it actually works

PyTorch's `Dataloader` works on a `Dataset` object.

A `Dataset` is any class implementing:

```
__len__(self) __getitem__(self, index)
```

The `Dataloader` repeatedly calls your dataset's `__getitem__` to get individual samples, then stacks them into batches.

---



### What `__getitem__` should return

Each item returned should typically be:

```
(image_tensor, label_tensor)
```

But they don't have to start as tensors — you can convert them *inside* the dataset definition.

---



### Example 1 — raw NumPy → converted inside dataset

```
from torch.utils.data import Dataset, DataLoader import torch import numpy
as np class MyDataset(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __len__(self):
        return len(self.x)
    def __getitem__(self, idx):
        # convert from numpy to torch tensor here
        return torch.tensor(self.x[idx], dtype=torch.float32), torch.tensor(self.y[idx],
            dtype=torch.long)
# create dataset from numpy arrays
x = np.random.rand(100, 3, 32, 32)
y = np.random.randint(0, 10, size=(100,))
dataset = MyDataset(x, y)
loader = DataLoader(dataset, batch_size=16, shuffle=True)
for xb, yb in loader:
    print(xb.shape, yb.shape)
    break
```



Works fine — even though `x` and `y` were originally NumPy arrays.

The conversion happens inside `__getitem__`.

---



### Example 2 — already tensors

If your data is already `torch.Tensor`, you can skip conversion:

```
dataset = torch.utils.data.TensorDataset(x_tensor, y_tensor) loader =  
    DataLoader(dataset, batch_size=32, shuffle=True)
```

## DataLoader collate step

Internally, `DataLoader` uses a “collate function” to stack samples into batches.

That means it can automatically handle:

- Tensors
- Lists of tensors
- Dictionaries containing tensors

as long as everything in a batch has the same shape.

## What won't work

If your dataset returns NumPy arrays directly, the default collate function will often fail  
(since it can't batch them into tensors automatically).

That's why you typically:

- Convert them manually (as above), or
- Use a `collate_fn` that handles NumPy.

## Summary

Input type	Works with DataLoader?	Notes
<code>torch.Tensor</code>		Ideal case
<code>np.ndarray</code>		Convert to tensor in <code>getitem</code>
list / dict of tensors		DataLoader can collate automatically

---

list / dict of numpy arrays



Needs custom collate or  
conversion

---

So:

- ✓ You don't need to start with tensors, but by the time your `Dataset` returns samples to the `DataLoader`, they *must be tensors*.

Would you like me to show a compact version that works for both MNIST *and* CIFAR-10, automatically converting them into a PyTorch-ready `DataLoader`?

You said:

how do you typically flatten np arrays in python?

ChatGPT said:

Great question — flattening NumPy arrays is super common, especially when preparing data for MLPs or reshaping image data.

There are a few ways to do it — here's a breakdown of the most common and reliable ones

---

## 1. Use `.reshape(-1)` — the most common way

```
import numpy as np a = np.array([[1, 2, 3], [4, 5, 6]]) flat =  
a.reshape(-1) print(flat) # [1 2 3 4 5 6] print(flat.shape) # (6,)
```

- ✓ `-1` tells NumPy to infer the correct size automatically.

✓ Works for any number of dimensions.

⚙ It returns a *view* when possible (no copy unless needed).

---

## 2. Use `.ravel()` — similar to `.reshape(-1)`

```
flat = a.ravel()
```

 Returns a *flattened view* of the array if possible (no memory copy).

 If the array is not contiguous, it might return a copy.

Use this when you want efficiency and don't need to worry about whether it's a view or a copy.

---



### 3. Use `.flatten()` — always returns a copy

```
flat = a.flatten()
```

 Returns a new array (copy).

 Slightly less efficient than `.reshape(-1)` or `.ravel()` if the array is large.

Use this if you explicitly need a copy and want to be sure changes to `flat` won't affect the original array.

---



### 4. Flatten multiple images

If you have, say, CIFAR-10 data `(50000, 32, 32, 3)` and want to flatten each image into a 1D vector:

```
x_flat = x.reshape(x.shape[0], -1) print(x_flat.shape) # (50000, 3072)
```

This flattens each image into a row of 3072 values.

 Perfect for MLP input:

`(batch_size, num_features)`



## Summary

Method	Returns copy?	Typical use
<code>.reshape(-1)</code>	 (view if possible)	Most flexible, clean

---

.ravel()	✗ (view if possible)	Fastest, efficient
.flatten()	✓ Always copy	Safe, independent data

---

✓ Typical for ML preprocessing:

```
x_train = x_train.reshape(x_train.shape[0], -1).astype('float32') / 255.0
```

That's the standard pipeline for flattening MNIST or CIFAR into vectors for an MLP.

Would you like me to show how to wrap that flattening automatically into a PyTorch

Dataset class for your MLP setup?

You said:

if i know which dimension to flatten along, how can i do that?

ChatGPT said:

Good question — flattening along a specific dimension (not the whole array) is a very common case when working with tensors or multi-dimensional NumPy arrays.

Let's go step-by-step 

---

## Example array

```
import numpy as np a = np.random.rand(2, 3, 4, 5) # shape = (2, 3, 4, 5)
```

Let's say you want to flatten along certain axes (e.g., combine the last two into one).

---

### ✓ 1. Use `reshape()` and control which dimensions are merged

If you want to flatten axes 2 and 3:

```
a_flat = a.reshape(a.shape[0], a.shape[1], -1) print(a_flat.shape) # (2, 3, 20)
```

 Here,  $4 * 5 = 20$ , so the last two dimensions are flattened together.

Basically, you decide where to put `-1`.

---

## 2. Use `np.reshape` more flexibly

If you want to flatten, say, the middle dimensions:

```
a_flat = a.reshape(a.shape[0], -1, a.shape[-1]) print(a_flat.shape) # (2,  
12, 5)
```

Here, dimensions  $3 * 4$  merged into 12.

---

## 3. If you know which specific axes to flatten (general case)

You can do it with a small helper:

```
def flatten_axes(x, start_axis, end_axis): new_shape =  
x.shape[:start_axis] + (-1,) + x.shape[end_axis+1:] return  
x.reshape(new_shape)
```

Example — flatten axes 1 through 2:

```
b = flatten_axes(a, 1, 2) print(b.shape) # (2, 12, 5)
```

---

## 4. If you're using PyTorch

It has a built-in for this:

```
import torch x = torch.randn(2, 3, 4, 5) y = torch.flatten(x, start_dim=1,  
end_dim=2) print(y.shape) # torch.Size([2, 12, 5])
```

Equivalent idea, but cleaner syntax.

---

## Summary Table

Goal	NumPy	PyTorch
------	-------	---------

---

Flatten everything	<code>x.reshape(-1)</code>	<code>torch.flatten(x)</code>
Flatten specific axes	<code>x.reshape(..., -1, ...)</code> or helper	<code>torch.flatten(x, start_dim, end_dim)</code>

---

Would you like me to show how you'd flatten CIFAR images from `(N, 32, 32, 3) → (N, 1024, 3)` (flatten spatial dims but keep channels separate)?

You said:

how do you use .reshape?

ChatGPT said:

Perfect — `.reshape()` is one of the most useful NumPy (and PyTorch) methods for controlling array shapes 

---

## Basic Idea

```
array.reshape(new_shape)
```

It returns a new view of the array (if possible) with the shape you specify, without changing the data — it just changes how it's *interpreted* in memory.

---

## Example 1 — Flatten everything

```
import numpy as np a = np.array([[1, 2, 3], [4, 5, 6]]) print(a.shape) # (2, 3) b = a.reshape(-1) # -1 = "figure out automatically" print(b) # [1 2 3 4 5 6] print(b.shape) # (6,)
```

Here, `-1` tells NumPy to infer that dimension automatically based on the total number of elements.

---

## Example 2 — Change shape but keep total elements same

```
a = np.arange(12) # [0, 1, 2, ..., 11] b = a.reshape(3, 4) # 3 rows, 4 columns print(b) # [[ 0  1  2  3] # [ 4  5  6  7] # [ 8  9 10 11]]
```

---

### ✓ Example 3 — Reshape multidimensional arrays

```
a = np.random.rand(2, 3, 4) # shape (2, 3, 4) b = a.reshape(2, 12) # merged last two dims print(b.shape) # (2, 12)
```

Total elements stayed the same:  $2 \times 3 \times 4 = 24 \rightarrow 2 \times 12 = 24$ .

---

### ✓ Example 4 — Keep one dimension flexible

You can put `-1` anywhere:

```
a = np.random.rand(4, 4, 4) # (4, 4, 4) b = a.reshape(4, -1, 2) # NumPy infers the middle print(b.shape) # (4, 8, 2)
```

Because  $4 \times 4 \times 4 = 64$ , and  $4 \times ? \times 2 = 64 \rightarrow ? = 8$ .

---

## ⚠ Rules / Gotchas

1. The total number of elements must not change (product of dimensions before = after).
  2. `.reshape()` doesn't modify the original array unless you do:
  3. `a = a.reshape(...)`
  4. It returns a view (shared memory) if possible — so modifying the reshaped array can also modify the original.
- 

Would you like me to show how to reshape CIFAR from `(50000, 32, 32, 3) → (50000, 3072)` to feed into an MLP?

You said:

is it that flatten always just reduces the dimension count by 1?

ChatGPT said:

Excellent question — and that's a really common misconception 

The short answer is:

👉 No — `.flatten()` (or `.reshape(-1)`) doesn't *always* reduce the dimension count by

1.

It actually reduces the array to *1D* — always exactly one dimension, regardless of how many you started with.

---

### ✳️ Example 1 — 2D → 1D

```
import numpy as np a = np.array([[1, 2, 3], [4, 5, 6]]) print(a.shape) #  
                                (2, 3) print(a.flatten().shape) # (6,)
```

2D → 1D ✓

---

### ✳️ Example 2 — 3D → 1D

```
b = np.zeros((4, 32, 32, 3)) print(b.shape) # (4, 32, 32, 3)  
print(b.flatten().shape) # (12288,)
```

4D → 1D ✓

---

### ✳️ Example 3 — N-D → 1D

No matter how many dimensions (`N`), `.flatten()` always gives you

`(total_elements,).`

If you want to reduce by only one dimension (say 4D → 3D), you need to use

`.reshape()` instead — specifying which axes to merge.

---

### ⚙️ Example: Merge last two dims only

```
a = np.random.rand(10, 4, 5) b = a.reshape(10, -1) # merges 4 and 5 → 20  
print(b.shape) # (10, 20)
```

So:

- `.flatten()` → always flattens *everything* into 1D
  - `.reshape()` → gives you control over *how much* to flatten
- 

Would you like me to show how to partially flatten CIFAR (e.g., keep batches, flatten each image)?

You said:

let's say i have 3 dimensions, and i only want to flatten along the 2nd dimension, like say if i have a bunch of samples, and each one is a monotone pic, how i'd flatten th pic

ChatGPT said:

Perfect — that's a great, very practical question 🌟

Let's go step by step.

---

### ✖ Suppose you have this setup:

You have a batch of grayscale (monotone) images — so each sample is 2D:

`shape = (num_samples, height, width)`

Example:

```
import numpy as np
x = np.random.rand(100, 28, 28) # 100 images of size
                                 28x28
print(x.shape) # (100, 28, 28)
```

Now you want to flatten only the image, not the batch —

so each `(28, 28)` becomes `(784, )`.

---

### ✓ Use `.reshape()` and keep the batch dimension intact

```
x_flat = x.reshape(x.shape[0], -1)
print(x_flat.shape) # (100, 784)
```

That merges all dimensions *after the first one* ( $\text{height} \times \text{width}$ ) into one.

---

### ⚙ General Rule

If you have an array with shape:

```
(batch, d1, d2, ..., dn)
```

and you want to flatten *everything except the batch dimension*, use:

```
x = x.reshape(x.shape[0], -1)
```

This works no matter how many extra dimensions you have (e.g., RGB → (batch, 32,

```
32, 3) → (batch, 3072)).
```

---

### If you really only want to flatten a **specific dimension** (not all after it)

Say shape = (A, B, C, D) and you only want to merge B and C, not D:

```
x = np.random.rand(2, 3, 4, 5) x_flat = x.reshape(2, 12, 5) # merged  
3*4=12 print(x_flat.shape) # (2, 12, 5)
```

---

So for your grayscale example, yes — you'd do:

```
x = x.reshape(x.shape[0], -1)
```

Would you like me to show how that same flattening works for CIFAR (RGB), where you have one extra color dimension?

You said:  
so will this work?

```
self.mnistXTrain = self.mnistXTrain.reshape(self.mnistXTrain.shape[0],  
                                             self.mnistXTrain.shape[1] * self.mnistXTrain.shape[2])
```

ChatGPT said:

 Yes — that will work perfectly for MNIST, because MNIST images are grayscale (shape (num\_samples, 28, 28)), and your reshape correctly flattens each 2D image into 1D.

Here's the reasoning:

---

### Suppose:

```
self.mnistXTrain.shape == (60000, 28, 28)
```

Your code:

```
self.mnistXTrain = self.mnistXTrain.reshape( self.mnistXTrain.shape[0],  
                                             self.mnistXTrain.shape[1] * self.mnistXTrain.shape[2] )
```

becomes:

```
(60000, 28 * 28) → (60000, 784)
```

 Exactly what you want for an MLP (each image flattened into a 784-length vector).

---

### Just one note:

If you later use CIFAR (which has shape `(num_samples, 32, 32, 3)`), this exact line won't work, because you're only multiplying the first two spatial dims ( $32 \times 32$ ) but not including the color channels.

For CIFAR, you'd need:

```
self.cifarXTrain = self.cifarXTrain.reshape( self.cifarXTrain.shape[0],  
                                             self.cifarXTrain.shape[1] * self.cifarXTrain.shape[2] *  
                                             self.cifarXTrain.shape[3] ) # (50000, 32*32*3) = (50000, 3072)
```

---

### Summary:

- Your line is 100% correct for MNIST.
- For CIFAR, include the extra `* self.cifarXTrain.shape[3]`.

Would you like me to show how to generalize it so the same function works for both datasets automatically?

You said:

how would i say add a dimension?

ChatGPT said:

Excellent question  — adding dimensions is just as common as flattening them, especially when preparing data for neural networks.

Here's how you can do it in NumPy (or PyTorch):

---

### 1. Using `np.expand_dims()`

This is the most explicit and readable way.

```
import numpy as np x = np.random.rand(28, 28) print(x.shape) # (28, 28) #
Add a new dimension at the front x = np.expand_dims(x, axis=0)
print(x.shape) # (1, 28, 28)
```

You can insert the new axis anywhere:

- `axis=0` → before the first dimension
- `axis=-1` → after the last dimension

Example:

```
x = np.expand_dims(x, axis=-1) print(x.shape) # (28, 28, 1)
```

---

### 2. Using slicing syntax: `np.newaxis`

This is shorthand for `expand_dims`.

```
x = np.random.rand(28, 28) x = x[np.newaxis, :, :] # add dim at front →
(1, 28, 28) x = x[:, :, :, np.newaxis] # add dim at end → (1, 28, 28, 1)
```

That's often what you'll see in deep learning code.

---

### 3. Using `.reshape()`

You can also reshape explicitly if you know the final shape.

```
x = np.random.rand(28, 28) x = x.reshape(1, 28, 28)
```

That works because you're telling it "I want one image with 28×28 pixels."

---

## When you need this

- MNIST grayscale: you often need to add a dummy "channel" dimension so it becomes `(batch, 28, 28, 1)` or `(batch, 1, 28, 28)` depending on framework.
  - `x = np.expand_dims(x, axis=-1)`
  - CNNs require that channel dimension even for monochrome images.
- 

Would you like me to show the typical way to add that extra channel dimension for MNIST before passing into a CNN?

You said:

so, will this work for a cnn for the mnist?

```
self.mnistXTest = np.expand_dims(self.mnistXTest, axis = 1)
```

ChatGPT said:

 You're very close — that line will *work*, but it depends on which framework convention your CNN is using:

---

## In PyTorch

PyTorch uses channel-first format:

```
(batch, channels, height, width)
```

So yes — your line:

```
self.mnistXTest = np.expand_dims(self.mnistXTest, axis=1)
```

✓ is exactly correct for PyTorch.

If your original shape is `(10000, 28, 28)`, it becomes `(10000, 1, 28, 28)` → perfect for a grayscale CNN.

---

## ✗ In TensorFlow / Keras

TensorFlow uses channel-last format:

`(batch, height, width, channels)`

So for that, you'd need:

```
self.mnistXTest = np.expand_dims(self.mnistXTest, axis=-1)  
→ (10000, 28, 28, 1)
```

---

## ✓ TL;DR

Framework	Channel format	Correct <code>expand_dims</code>
PyTorch	<code>(N, C, H, W)</code>	<code>axis=1</code> ✓
TensorFlow/Keras	<code>(N, H, W, C)</code>	<code>axis=-1</code> ✓

---

Would you like me to show how to make that line automatically adapt depending on whether you're using PyTorch or TensorFlow?

You said:

how do you swap dimensions?

ChatGPT said:

Swapping dimensions is super common when you need to convert between channel-first and channel-last formats, or just reorder axes in general.

Here's how to do it in NumPy and PyTorch.

---

## 1. NumPy: transpose

```
import numpy as np x = np.random.rand(100, 28, 28, 1) # (batch, height, width, channels) # Swap to (batch, channels, height, width) x_swapped = x.transpose(0, 3, 1, 2) print(x_swapped.shape) # (100, 1, 28, 28)
```

 `transpose(*axes)` takes the axes in the order you want.

- Original axes: `(0, 1, 2, 3)`
  - New order: `(0, 3, 1, 2)`
- 

## Quick rule of thumb

Goal	NumPy code
HWC → CHW	<code>x.transpose(0, 3, 1, 2)</code>
CHW → HWC	<code>x.transpose(0, 2, 3, 1)</code>

---

## 2. PyTorch: permute

PyTorch's `tensor.permute()` works similarly:

```
import torch x = torch.randn(100, 28, 28, 1) # (batch, H, W, C) x_swapped = x.permute(0, 3, 1, 2) # → (batch, C, H, W) print(x_swapped.shape) # torch.Size([100, 1, 28, 28])
```

- `permute(*dims)` lets you reorder any axes.
  - Unlike `view` or `reshape`, `permute` doesn't flatten anything — it just changes the dimension order.
- 

## Notes

1. `transpose` (NumPy) and `permute` (PyTorch) are cheap operations — usually just a view, not a copy.

2. Use them anytime you need channel-first vs channel-last conversion.
- 

If you want, I can show a mini helper function that will automatically swap your MNIST or CIFAR arrays into the correct format depending on whether you're using PyTorch or TensorFlow.

You said:

if say i need 3 models of an MLP, would it be better to make a mlp class that then has 3 instances of "class MLP(nn.Module)", or make one MLP class that imports nn.module and create the models a separate way?

ChatGPT said:

That's a *really* good question — and it touches on both code organization and PyTorch architecture design.

Let's break it down clearly 

---

## Your Two Options

### A Option 1 — Wrapper Class that Holds 3 `MLP` Models

You make one class (say, `ModelManager`) that contains *three instances* of your `MLP(nn.Module)` class.

```
import torch.nn as nn
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size), nn.ReLU(), nn.Linear(hidden_size, output_size))
    def forward(self, x):
        return self.net(x)

class ModelManager:
    def __init__(self):
        self.model1 = MLP(784, 128, 10)
        self.model2 = MLP(784, 256, 10)
        self.model3 = MLP(784, 512, 10)
```

 Pros:

- Very clean separation — `MLP` focuses purely on architecture.
- `ModelManager` can handle experiments, training loops, etc.

- Easy to manage multiple networks, run them in parallel, etc.
    - 💡 Recommended if you plan to compare or ensemble multiple models.
- 

## B Option 2 — One Big Class that Inherits `nn.Module` and Defines 3

### Inside

You could also define a single class that *itself* is an `nn.Module`, and it holds three submodules:

```
class TripleMLP(nn.Module): def __init__(self, input_size, hidden_size, output_size): super().__init__() self.mlp1 = MLP(input_size, hidden_size, output_size) self.mlp2 = MLP(input_size, hidden_size, output_size) self.mlp3 = MLP(input_size, hidden_size, output_size) def forward(self, x): # example: average their outputs out1 = self.mlp1(x) out2 = self.mlp2(x) out3 = self.mlp3(x) return (out1 + out2 + out3) / 3
```

#### ✓ Pros:

- Keeps everything in a single trainable `nn.Module`.
- Nice if the models are *meant to interact or ensemble* internally.

#### ⚠ Cons:

- Less flexible if you want to train or validate them separately.
  - Harder to scale if you want variable numbers of MLPs later.
- 

## 🧠 Guideline

Use Case	Recommended Design
You want to train/test models independently	<input checked="" type="checkbox"/> Separate <code>MLP</code> class + external manager (Option A)
You want to train them jointly / ensemble as one	<input checked="" type="checkbox"/> Single <code>nn.Module</code> containing submodules (Option B)

---

## Practical Tip

Even for random search or hyperparameter optimization, it's often better to:

```
models = [MLP(784, h, 10) for h in [128, 256, 512]]
```

This keeps your code flexible and clean.

---

Would you like me to show a short example of how to train and compare all three models in parallel (on CPU or GPU) using that first approach?

You said:

```
class MLP(nn.Module): def __init__(self, input_size, hidden_sizes, output_size): super(MLP, self).__init__()
```

also here, how does the super parameter work?

ChatGPT said:

Ah — perfect, let's break down what `super()` is doing here.

---

## Context

Your code looks like this:

```
import torch.nn as nn class MLP(nn.Module): def __init__(self, input_size, hidden_sizes, output_size): super(MLP, self).__init__() # your layers here
```

---

## What `super()` does

1. `super()` gives you access to the parent class.

In this case, `MLP` inherits from `nn.Module`.

So `super(MLP, self)` refers to the `nn.Module` part of your object.

2. Calling `__init__()` on the parent initializes it.

PyTorch's `nn.Module` has its own internal bookkeeping for layers, parameters, children modules, etc.

Without calling it, things like `self.parameters()` and `self.to(device)` won't work correctly.

---

#### ◆ Why `super(MLP, self).__init__()` works

- `MLP` is the current class
  - `self` is the instance
  - This calls `nn.Module.__init__(self)` safely — initializing the parent part of the object.
- 

#### ◆ Modern Python syntax

In Python 3, you can simplify this:

```
class MLP(nn.Module): def __init__(self, input_size, hidden_sizes,  
output_size): super().__init__() # automatically knows the parent
```

✓ Exactly equivalent and more concise.

---

#### ◆ Analogy

Think of it like this:

MLP └─ nn.Module

`super().__init__()` says:

"Hey, initialize the `nn.Module` part of me before I start adding my own layers."

---

So in short:

- Purpose: Initialize the parent class (`nn.Module`) so PyTorch can track your layers and parameters.
  - Python 3 tip: Just use `super().__init__()` — no need to repeat the class name and `self`.
-

If you want, I can also explain why forgetting `super().__init__()` causes weird PyTorch errors, which trips up a lot of beginners.

You said:

i mean do you have to do `super(MLP, self)?` why not like `super().__init__()`?

ChatGPT said:

Ah — got it! You're asking about the difference between the old style `super(MLP, self).__init__()` and the modern Python 3 style `super().__init__()`.

Let's break it down clearly 

---

## 1 Old-style: `super(MLP, self).__init__()`

```
class MLP(nn.Module): def __init__(self, ...): super(MLP, self).__init__()
```

- You explicitly tell Python:  
“I want the parent of `MLP` for this instance `self`.”
  - Required in Python 2 because `super()` didn't automatically know the current class or instance.
  - Works in Python 3 too, but a bit verbose.
- 

## 2 Modern Python 3 style: `super().__init__()`

```
class MLP(nn.Module): def __init__(self, ...): super().__init__()
```

- Python 3 automatically figures out:
    - the current class (`MLP`)
    - the current instance (`self`)
  - Cleaner, less code, and exactly equivalent.
  -  Recommended in modern PyTorch code.
- 

## Key takeaway

Style	Python version	Notes
<code>super(MLP, self).__init__()</code>	2 & 3	Explicit, old style

---

```
super().__init__()
```

3+

Cleaner, automatic, recommended

---

## ◆ Why it works

`super().__init__()` calls the `__init__()` of the first parent class in the method resolution order (MRO).

In `MLP(nn.Module)`, the MRO is:

`MLP → nn.Module → object`

So `super().__init__()` initializes `nn.Module`.

---

✓ Bottom line:

- In Python 3, you don't need `super(MLP, self)` — just use `super().__init__()`.
  - Old style exists mostly for backward compatibility with Python 2.
- 

If you want, I can also explain why skipping `super().__init__()` entirely causes PyTorch layers not to register — that's another common gotcha.

You said:

`self.net = nn.Sequential(*layers)`

also, here how does doing `*layers` work?

ChatGPT said:

Ah, perfect — that's a really useful Python trick combined with PyTorch. Let's break it down.

---

## 1 The \* operator in Python

The `*` operator is called “argument unpacking”.

- If you have a list or tuple of items, `*` lets you pass each item as a separate argument to a function.

### Example:

```
def add(a, b, c): return a + b + c
numbers = [1, 2, 3]
print(add(*numbers)) # same as add(1, 2, 3)
```

 Output: 6

---

## 2 Using it in PyTorch: `nn.Sequential(*layers)`

Suppose you have:

```
import torch.nn as nn
layers = [nn.Linear(784, 128), nn.ReLU(),
          nn.Linear(128, 10)]
net = nn.Sequential(*layers)
```

- `nn.Sequential()` expects each layer as a separate argument:  
`nn.Sequential(nn.Linear(784, 128), nn.ReLU(), nn.Linear(128, 10))`
- If you just did `nn.Sequential(layers)` without `*`, it would treat the entire list as a single module, which won't work.

 Using `*layers` unpacks the list so each element becomes a separate argument to

`nn.Sequential`.

---

## 3 Why it's convenient

1. You can programmatically build a list of layers (e.g., for variable hidden sizes).
2. Then pass them all to `nn.Sequential` without writing them manually.

### Example:

```
hidden_sizes = [128, 256, 512]
layers = []
input_size = 784
for h in hidden_sizes:
    layers.append(nn.Linear(input_size, h))
    layers.append(nn.ReLU())
    input_size = h
layers.append(nn.Linear(input_size, 10))
net = nn.Sequential(*layers)
```

- Now you have an MLP with any number of hidden layers, dynamically created.

---

✓ Bottom line:

- `*layers` unpacks a list into separate arguments.
  - Needed in `nn.Sequential` because it doesn't take a single list — it wants each layer individually.
- 

If you want, I can show a full example of an MLP class that dynamically builds any number of layers using `*layers`, ready for MNIST or CIFAR.

You said:

what's the difference between a single asterisk and a double asterisk?

ChatGPT said:

Ah — yes! In Python, `*` and `**` are related but do slightly different things. They're both about unpacking arguments, but in different ways. Let's break it down clearly:

---

## 1 Single asterisk: `*` — positional unpacking

✓ Uses:

1. Unpack a list or tuple into positional arguments

```
def add(a, b, c): return a + b + c
nums = [1, 2, 3]
print(add(*nums)) # same as add(1, 2, 3)
```

2. Collect extra positional arguments in a function

```
def foo(a, *args):
    print(a)
    print(args)
foo(1, 2, 3, 4) # Output: # 1 #
                  (2, 3, 4)
```

Here, `*args` collects any remaining positional arguments as a tuple.

---

## 2 Double asterisk: `**` — keyword/unpacking of dictionaries

✓ Uses:

## 1. Unpack a dictionary into keyword arguments

```
def greet(name, age): print(f"{name} is {age} years old") person = {"name": "Alice", "age": 30} greet(**person) # same as greet(name="Alice", age=30)
```

## 2. Collect extra keyword arguments in a function

```
def foo(a, **kwargs): print(a) print(kwargs) foo(1, b=2, c=3) # Output: # 1 # {'b': 2, 'c': 3}
```

Here, `**kwargs` collects any extra keyword arguments as a dict.

## 3 Summary Table

Symbol	In a call	In a function	Example
*	Unpacks a list/tuple into positional args	Collects extra positional args	<code>f(*[1,2,3])</code>
**	Unpacks a dict into keyword args	Collects extra keyword args	<code>f(**{'x':1,'y':2})</code>

## ⚡ Example combining both

```
def foo(a, b, c, d): print(a, b, c, d) args = [1, 2] kwargs = {'c': 3, 'd': 4} foo(*args, **kwargs) # prints: 1 2 3 4
```

- `*args` → fills `a` and `b`
- `**kwargs` → fills `c` and `d`

If you want, I can also show how this applies to PyTorch MLPs, for example using

`*layers` vs `**kwargs` in a class constructor.

You said:

```
layers.append(nn.Linear(in_size, output_size))
```

also, for a linear layer is output size just the number of perceptrons on that layer?

ChatGPT said:

Exactly right 

In PyTorch, when you write something like:

```
layers.append(nn.Linear(in_size, output_size))
```

- `in_size` → number of input features going into the layer
- `output_size` → number of output features (or neurons/perceptrons) in that layer

So yes — the `output_size` here is the number of perceptrons (units) in that layer.

Each perceptron (or neuron) in a fully connected layer takes all `in_size` inputs, multiplies them by a weight vector of length `in_size`, adds a bias, and outputs a single value. So if you have `output_size = 128`, that means you have 128 perceptrons, each producing one output — giving a final output shape of `(batch_size, 128)`.

---

### Example:

```
fcl = nn.Linear(784, 128) # 784 inputs (e.g., flattened 28x28 MNIST), 128  
output neurons fc2 = nn.Linear(128, 10) # 128 inputs (from previous  
layer), 10 output neurons (e.g., classes)
```

So:

- Layer 1 → outputs a vector of length 128
  - Layer 2 → outputs a vector of length 10 (final class logits)
- 

Would you like me to show how these connect together in an `nn.Sequential` MLP structure?

You said:

③ Define the MLP Model ----- python Copy code `class MLP(nn.Module): def  
`__init__(self, input_size, hidden_sizes, output_size): super(MLP, self).__init__()` layers = []  
`in_size = input_size` # Hidden layers for h in hidden\_sizes: layers.append(nn.Linear(in\_size, h))  
`layers.append(nn.ReLU())` `in_size = h` # Output layer layers.append(nn.Linear(in\_size, output\_size))`

```
self.net = nn.Sequential(*layers) def forward(self, x): return self.net(x)` Example instantiation for  
MNIST:
```

for a cnn as well, would the output be a linear layer?

ChatGPT said:

Perfect — that's a really good question, and yes 

In most CNN architectures, the final layer *is* typically a linear (fully connected) layer — just like in the MLP you mentioned.

---

### Why:

After the convolutional and pooling layers finish extracting spatial features (edges, shapes, textures, etc.), we need to:

1. Flatten those spatial features into a single vector, and
2. Map them to the desired number of outputs (e.g., class scores).

That mapping step is done using `nn.Linear`, i.e., a fully connected layer.

---

### Example:

Here's a minimal CNN for MNIST:

```
import torch import torch.nn as nn import torch.nn.functional as F class  
CNN(nn.Module): def __init__(self): super(CNN, self).__init__()  
    self.conv1 = nn.Conv2d(1, 32, 3, padding=1) # (1 channel → 32 filters)  
    self.pool = nn.MaxPool2d(2, 2) # halve spatial dimensions  
    self.conv2 = nn.Conv2d(32, 64, 3, padding=1) # (32 → 64 filters) # After two poolings on 28x28 → 7x7  
    (28/2/2 = 7)  
    self.fc1 = nn.Linear(64 * 7 * 7, 128) # flatten and connect  
    self.fc2 = nn.Linear(128, 10) # 10 classes for MNIST  
  
    def forward(self, x):  
        x = self.pool(F.relu(self.conv1(x))) # (N, 32, 14, 14)  
        x = self.pool(F.relu(self.conv2(x))) # (N, 64, 7, 7)  
        x = x.view(x.size(0), -1) # flatten to (N, 64*7*7)  
        x = F.relu(self.fc1(x))  
        x = self.fc2(x)  
        return x
```

---



## Summary:

Type	Typical Final Layer	Purpose
MLP	<code>nn.Linear(last_hidden, output)</code>	Direct mapping from features to outputs
CNN	<code>nn.Linear(flattened_features , output)</code>	Converts spatial feature maps to class scores

So yes — your CNN will usually end with one or more fully connected (`nn.Linear`) layers after the convolutional stack.

You said:

also, for the mnist and cifar is there any way to get dynamically the # of potential classes, or should i just hardcode them in?

ChatGPT said:

Excellent question — and one that comes up a lot when you're trying to make your model code more reusable 🤖

Here's the deal for MNIST and CIFAR specifically:

---

### 1. Number of Classes (Ground Truth)

- MNIST → 10 classes (digits 0–9)
- CIFAR-10 → 10 classes (airplane, car, bird, cat, deer, dog, frog, horse, ship, truck)

Both datasets are fixed — meaning they *always* have exactly 10 classes.

So in practice, it's completely fine (and common) to hardcode `num_classes = 10` for both.

---

### 2. If you still want to *dynamically* infer it:

You can just compute it from the dataset labels, like this:

```

from tensorflow.keras.datasets import mnist, cifar10 import numpy as np
(x_train, y_train), (x_test, y_test) = mnist.load_data() num_classes_mnist
= len(np.unique(y_train)) (x_train, y_train), (x_test, y_test) =
cifar10.load_data() num_classes_cifar = len(np.unique(y_train))

```

This way your code will automatically adapt if you ever swap in another dataset (like CIFAR-100, which has 100 classes).

---



## Summary:

Dataset	# Classes	How to get it	Recommendation
MNIST	10	len(np.unique(y_train))	Hardcode 10
CIFAR-10	10	len(np.unique(y_train))	Hardcode 10
CIFAR-100	100	len(np.unique(y_train))	Dynamic check useful

---

So:

✓ For MNIST and CIFAR-10 → hardcoding 10 is totally fine.

🤖 For general-purpose code (say, “any image dataset”) → use

len(np.unique(y\_train)).

Would you like me to show how to integrate that dynamic class detection directly into your dataset-loading method?

You said:

how would you use a CNN in python?

ChatGPT said:

Perfect — let’s go step-by-step on how to use a CNN in Python (with PyTorch) 🤝



## 1. Overview

A Convolutional Neural Network (CNN) is the go-to model for image data.

We'll show an example using MNIST (grayscale 28×28 digits), but this applies equally to CIFAR-10 (32×32 RGB images).

---

## 2. Imports

```
import torch import torch.nn as nn import torch.nn.functional as F import  
torch.optim as optim from torchvision import datasets, transforms from  
torch.utils.data import DataLoader
```

---

## 3. Load and Normalize the Dataset

Use `torchvision.datasets` (which gives ready-made datasets like MNIST and CIFAR).

```
# Transform to Tensor + normalize pixel values between 0 and 1 transform =  
transforms.Compose([ transforms.ToTensor(), transforms.Normalize((0.5,),  
(0.5,)) # for MNIST (grayscale) ]) # Download + load training/test sets  
train_set = datasets.MNIST(root='./data', train=True, download=True,  
transform=transform) test_set = datasets.MNIST(root='./data', train=False,  
download=True, transform=transform) # DataLoaders (handle batching and  
shuffling) train_loader = DataLoader(train_set, batch_size=64,  
shuffle=True) test_loader = DataLoader(test_set, batch_size=1000,  
shuffle=False)
```

(For CIFAR-10, replace `datasets.MNIST` with `datasets.CIFAR10` and normalize with 3 channels: `transforms.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))`.)

---

## 4. Define the CNN Model

Here's a simple CNN example:

```
class CNN(nn.Module): def __init__(self): super(CNN, self).__init__()  
self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1) # 1 input channel  
(MNIST), 32 filters self.conv2 = nn.Conv2d(32, 64, kernel_size=3,
```

```

padding=1) self.pool = nn.MaxPool2d(2, 2) # halves spatial dimensions
self.fc1 = nn.Linear(64 * 7 * 7, 128) # 7x7 from two poolings on 28x28
self.fc2 = nn.Linear(128, 10) # 10 output classes (digits) def
forward(self, x): x = self.pool(F.relu(self.conv1(x))) x =
self.pool(F.relu(self.conv2(x))) x = x.view(x.size(0), -1) # flatten x =
F.relu(self.fc1(x)) x = self.fc2(x) return x

```

For CIFAR-10, change the first layer to `nn.Conv2d(3, 32, ...)` since RGB has 3 channels.

---

## 5. Train the CNN

```

model = CNN() optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss() for epoch in range(5): for images,
labels in train_loader: optimizer.zero_grad() outputs = model(images) loss
= criterion(outputs, labels) loss.backward() optimizer.step()
print(f"Epoch {epoch+1}/5, Loss: {loss.item():.4f}")

```

## 6. Evaluate on Test Data

```

correct = 0 total = 0 model.eval() # turn off dropout/batchnorm with
torch.no_grad(): for images, labels in test_loader: outputs =
model(images) _, predicted = torch.max(outputs, 1) total += labels.size(0)
correct += (predicted == labels).sum().item() print(f"Test Accuracy: {100
* correct / total:.2f}%")

```

## Summary

Step	Description
1	Import <code>torch</code> , <code>torchvision</code>
2	Load + normalize data

---

3

Create a `CNN` class (`nn.Module`)

---

4

Train using `optimizer + loss.backward()`

---

5

Evaluate accuracy

---

Would you like me to adapt this example specifically for CIFAR-10 (with its 3 RGB channels and different shape)?

You said:

what is a fully connected layer?

ChatGPT said:

Excellent question — and this is *the* key building block behind both MLPs and the final stages of CNNs 

---

## Definition

A fully connected layer (also called a dense layer or `Linear` layer in PyTorch) is a layer where

 every neuron in the layer is connected to every neuron in the previous layer.

In math terms, it performs this operation:

$$y = Wx + b$$

$$y = Wx + b$$

Where:

- $x$
- $x$ : input vector (all features from the previous layer, flattened)
- $W$
- $W$ : weight matrix (learnable parameters)

- $b$
  - $b$ : bias vector
  - $y$
  - $y$ : output vector (the activations of the neurons in this layer)
- 

## In PyTorch

It's represented by:

```
nn.Linear(in_features, out_features)
```

### Example:

```
layer = nn.Linear(784, 128)
```

This creates:

- 784 inputs (e.g., flattened 28×28 MNIST image)
- 128 outputs (neurons in this layer)

So each of the 128 neurons has its own set of 784 weights + 1 bias.

---

## Why It's “Fully Connected”

Every neuron gets information from every input value of the previous layer — unlike convolutional layers, which only look at *local regions* of the input (like 3×3 patches).

So visually:

```
Prev Layer:   o o o o o  
                 \ / / / /  
Current Layer: o o o
```

Every circle (neuron) in the bottom layer is connected to every one in the top layer.

---

## Role in Different Architectures

Model Type	Where FC Layers Appear	Purpose
MLP (Fully Connected Network)	All layers	Entire network is FC layers
CNN	At the end	Combine extracted spatial features into final class logits
RNN / Transformer	Often at the output	Map hidden states to predictions



### Example in a CNN

After convolution and pooling layers, you flatten the feature maps and use FC layers to make predictions:

```
x = x.view(x.size(0), -1) # flatten (batch, features) x =
F.relu(self.fc1(x)) # fully connected layer x = self.fc2(x) # final output
layer
```

So in short:

- ◆ A fully connected layer connects *every* input to *every* output with learnable weights — it's what actually performs the final “decision-making” after earlier layers extract features.

Would you like a quick diagram (ASCII or image) to visualize the difference between convolutional and fully connected layers?

You said:

for CNN's, after doing the convolutional layers nad pooling layers, as you're about to send it to the linear layers, do you typically just use one linear layer or multiple?

ChatGPT said:

Fantastic question — and this is one that really shapes the *design depth* of your CNN



The short answer is:

- ✓ You *can* use just one linear layer, but most CNNs use multiple to help the model learn more abstract, high-level combinations of the extracted features before the final classification.

Let's unpack that a bit:

---

## What Happens After the Convs

After the convolutional and pooling layers:

- The output is a feature map (e.g., `(batch, channels, height, width)`).
  - You flatten it into a vector (e.g., `(batch, channels × height × width)`).
  - That vector is fed into one or more fully connected layers.
- 

## Design Choices

### Option 1: One Linear Layer (Minimalist)

```
x = x.view(x.size(0), -1) x = self.fc(x) # nn.Linear(flattened_size,  
num_classes)
```

✓ Simple and fast

- 🚫 Might underfit — because the model jumps directly from feature maps to predictions without combining those features in nonlinear ways.

→ Often used for smaller datasets like MNIST.

---

### Option 2: Multiple Linear Layers (Standard Practice)

```

x = x.view(x.size(0), -1) x = F.relu(self.fc1(x)) #
nn.Linear(flattened_size, 512) x = F.relu(self.fc2(x)) # nn.Linear(512,
128) x = self.fc3(x) # nn.Linear(128, num_classes)

```

Gives the network more representational power

Allows for complex decision boundaries

Slightly slower, needs more parameters, risk of overfitting

→ Typical in models like VGG, AlexNet, etc.

---

## Real Examples

Model	# of FC Layers	Notes
LeNet-5	2	Classic MNIST CNN
AlexNet	3	4096 → 4096 → 1000
VGG16	3	4096 → 4096 → 1000
ResNet	1	Just one FC at the end (because most depth is in conv blocks)

---

## Rule of Thumb

- Small/simple datasets (MNIST) → 1 or 2 FC layers
  - Medium/large datasets (CIFAR, ImageNet) → 2–3 FC layers
  - Modern architectures (ResNet, EfficientNet) → usually only 1 because most “learning” happens in the conv layers
- 

So in summary:

Dataset	Recommended # of Linear Layers	Example
---------	--------------------------------	---------

---

MNIST	1–2	( $64 \times 7 \times 7 \rightarrow 128 \rightarrow 10$ )
CIFAR-10	2–3	( $512 \times 4 \times 4 \rightarrow 256 \rightarrow 128 \rightarrow 10$ )
ImageNet	1–3 (depends on model)	2048 → 1000

---

Would you like me to show a compact PyTorch CNN example for CIFAR-10 that uses *two linear layers* (a very common real-world design)?

You said:

what parameters does nn.conv2d expect?

ChatGPT said:

Excellent — that's one of the most important layers in CNNs, so let's go over

`torch.nn.Conv2d` carefully 

---

## `torch.nn.Conv2d` — Overview

This layer performs a 2D convolution, typically used on images.

### Signature:

```
torch.nn.Conv2d( in_channels, out_channels, kernel_size, stride=1,  
padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros' )
```

Let's break those down:

---

## Parameters Explained

Parameter	Type	Description
<code>in_channels</code>	<code>int</code>	Number of input channels (e.g., 1 for grayscale, 3 for RGB).

---

<code>out_channels</code>	<code>int</code>	Number of filters (i.e., how many feature maps you want to produce).
<code>kernel_size</code>	<code>int or tuple</code>	Size of the filter (e.g., <code>3</code> or <code>(3, 3)</code> ).
<code>stride</code>	<code>int or tuple</code>	How far the filter moves at each step (default <code>1</code> ). Higher = smaller output.
<code>padding</code>	<code>int or tuple</code>	How many pixels to pad around the input (e.g., <code>1</code> keeps output size same for <code>3x3</code> kernels).
<code>dilation</code>	<code>int or tuple</code>	Spacing between kernel elements — used for <i>dilated convolutions</i> (default <code>1</code> ).
<code>groups</code>	<code>int</code>	Controls connections between inputs and outputs (rarely used; default <code>1</code> = fully connected).
<code>bias</code>	<code>bool</code>	Whether to include a bias term per filter (default <code>True</code> ).
<code>padding_mode</code>	<code>'zeros', 'reflect', 'replicate', 'circular'</code>	How to pad the image (default <code>'zeros'</code> ).

---



## Example 1 — Simple CNN Layer for MNIST

```
nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1,
          padding=1)
```

- Input: 1 channel (grayscale)
- Output: 32 filters
- Each filter:  $3 \times 3$  kernel
- Stride = 1 (move by 1 pixel each time)

- Padding = 1 (so 28×28 stays 28×28)
- 

## Example 2 — For CIFAR-10

```
nn.Conv2d(in_channels=3, out_channels=64, kernel_size=5, stride=1,
          padding=2)
```

- Input: 3 channels (RGB)
  - Output: 64 feature maps
  - Keeps same spatial size (because `padding=2` with `kernel=5`)
- 

## Output Size Formula

You can compute the output spatial size ( $H_{out} \times W_{out}$ ) with:

$$H_{out} = H_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel\_size} - 1) - 1 \times \text{stride} + 1$$

$H$

$out$

=

$\text{stride}$

$H$

$in$

$$+ 2 \times \text{padding} - \text{dilation} \times (\text{kernel\_size} - 1) - 1$$

+1

(Same formula for width.)

So you can control spatial shrinkage precisely.

---

## ✓ Typical Defaults You'll See

Dataset	Conv2d Example	Notes
MNIST	<code>Conv2d(1, 32, 3, padding=1)</code>	Keep 28×28 same size
CIFAR-10	<code>Conv2d(3, 64, 3, padding=1)</code>	RGB input
Deep CNNs	<code>Conv2d(64, 128, 3, stride=2, padding=1)</code>	Downsample features

---

Would you like me to show visually (with numbers) how an input like 28×28×1 changes shape after each conv + pooling layer?

You said:

do we usually nearly always use a stride of 1?

ChatGPT said:

Good question — not *always*, but very often, yes.

Here's the breakdown ⤵

✖ `stride` in `nn.Conv2d`

- `stride` controls how far the convolutional filter moves at each step (how much it slides across the input).
  - Default is `stride=1`, meaning the filter moves 1 pixel at a time → maximal overlap → more detailed feature maps.
- 

## ✓ Common Choices

Use case	Typical stride	Why
Preserve spatial resolution	<code>stride=1</code>	Keeps output close in size to input (only reduced by padding/filter size).

---

---

Downsample feature maps	<code>stride=2</code>	Reduces output width/height by half → similar to pooling layer effect.
Early layers	<code>1</code>	You want to capture fine details.
Later layers	<code>2 (sometimes)</code>	Reduce computation and spatial size before flattening.

---

## Example

```
import torch.nn as nn # First layer: stride=1 (preserve details) conv1 =
nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride=1,
padding=1) # Later layer: stride=2 (downsample) conv2 =
nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=2,
padding=1)
```

---

In short:

→ Yes, we *usually* start with `stride=1`, but we might use `stride=2` later to reduce feature map size instead of using pooling.

Would you like me to show you a minimal CNN example (like for MNIST or CIFAR) that includes both stride 1 and stride 2?

You said:

do you have to worry about padding if you're doing a strike of 1?

ChatGPT said:

Excellent question — and the answer is “sometimes, but not always.”

Let's unpack that 

---

## What Padding Does

Padding adds extra pixels (usually zeros) around the image so that convolutions near the border have enough context.

Without padding, each convolution “shrinks” the spatial size of the output because the filter can’t go beyond the edge.

---

### Formula for output size

For a 2D convolution:

$$\text{Output size} = \lfloor \frac{W+2P-KS}{S} \rfloor + 1$$

$$\text{Output size} = \lfloor$$

$$S$$

$$W+2P-K$$

$$\rfloor + 1$$

where:

- W
  - $W$ : input width/height
  - P
  - $P$ : padding
  - K
  - $K$ : kernel size
  - S
  - $S$ : stride
- 

### With stride = 1

Case	Padding	Result
------	---------	--------

---

No padding ( <code>padding=0</code> )	Output smaller than input (shrinks by $\frac{K-1}{2}$ pixels total).
“Same” padding ( <code>padding = K//2</code> )	Output has same width/height as input.
Large padding	Output slightly larger (rare).

---

## Example

```
import torch.nn as nn # Shrinks feature map: 28x28 -> 26x26 (kernel=3, no
                     padding) conv_no_pad = nn.Conv2d(1, 32, kernel_size=3, stride=1,
padding=0) # Keeps same size: 28x28 -> 28x28 conv_same = nn.Conv2d(1, 32,
                     kernel_size=3, stride=1, padding=1)
```

---

## Rule of Thumb

- If you want to preserve spatial dimensions, use `padding = kernel_size // 2`.
- If shrinking is okay (e.g., you’ll pool later anyway), you can skip padding.

So:

With `stride = 1`, you don’t *have* to use padding, but you often do, to preserve image size across layers and avoid losing border information.

---

Would you like me to show visually (with numbers) how the image shrinks or stays the same for `stride=1` with/without padding?

You said:

also, for `conv2d` do you not have to pass the height and width of it?

ChatGPT said:

Great catch — and you’re absolutely right to notice that.

👉 No, you do not need to pass the height or width of the input explicitly when creating a `Conv2d` layer in PyTorch.

---

### 🧠 Here's why:

PyTorch's `nn.Conv2d` only needs to know how many channels are coming in — not the spatial dimensions.

The layer automatically infers the height and width from the actual input tensor when you feed data through it.

---

### ✖ Example

```
import torch import torch.nn as nn # Create a convolutional layer conv =  
nn.Conv2d( in_channels=3, # RGB input out_channels=16, # number of filters  
kernel_size=3, # size of each filter stride=1, padding=1 ) # Input image  
batch (N=8 images, 3 channels, 32x32 each) x = torch.randn(8, 3, 32, 32) #  
Run forward pass y = conv(x) print(y.shape)
```

### ✓ Output:

```
torch.Size([8, 16, 32, 32])
```

PyTorch looked at your input tensor `(8, 3, 32, 32)` and did all the size math internally — no need to tell it the height/width beforehand.

---

### ✖ Why the shape works that way:

- Dimension 0 → batch size
  - Dimension 1 → number of input channels (you must match this to `in_channels`)
  - Dimensions 2 and 3 → height and width (deduced automatically)
- 

So you only specify:

```
nn.Conv2d(in_channels, out_channels, kernel_size, ...)
```

and PyTorch figures out the spatial part dynamically.

---

Would you like me to show how the output dimensions change automatically if you alter stride, padding, or kernel size?

You said:

also, why do we downsample using pooling layers instead of say just increasing the step size?

ChatGPT said:

Excellent question — this gets into one of the key design decisions of CNNs.

Let's unpack why pooling is preferred over just increasing the stride in convolutions.

---

## 1. Conceptual difference

Stride and pooling both reduce spatial resolution (downsample),

but they do it for *different purposes*:

Concept	What it does	When it's used
Stride > 1	Makes the convolution “jump” over pixels, reducing the output size directly during feature extraction	Used for <i>learned downsampling</i> — each filter learns features while moving less frequently
Pooling (e.g. MaxPool2d)	Applies a fixed function (like max or average) in non-overlapping regions, without learning new parameters	Used for <i>information condensation</i> — keeps the most important features

---

## 2. Pooling gives translation invariance

Pooling helps the network ignore small shifts or distortions in the image.

For example:

- If a feature (like an edge) moves a few pixels, max pooling still outputs the same activation.
- With just stride, the exact position affects the output much more — the network becomes *sensitive* to small translations.

So pooling makes the network more robust to tiny movements or noise in input.

---

### 3. Pooling separates feature extraction and downsampling

Using stride to downsample *during* convolution couples these two effects:

- The convolution filters learn *and* reduce resolution simultaneously.

Using pooling keeps them separate:

1. Convolution → learns features
2. Pooling → summarizes those features spatially

That separation helps the model generalize better and simplifies tuning.

---

### 4. Pooling has no learnable parameters

Unlike convolution (which learns weights), pooling is a fixed operation.

This prevents overfitting and adds a form of “structural prior” that enforces locality and smoothness.

---

### 5. That said — modern CNNs sometimes skip pooling!

Recent architectures like ResNet, EfficientNet, and Vision Transformers often:

- Use stride-2 convolutions instead of pooling,
- Or replace pooling with global average pooling at the end.

So, while pooling is conceptually clean and useful for translation invariance, modern models sometimes remove it for simplicity and better gradient flow — relying on strided convolutions instead.

---

## Summary

Property	Pooling	Strided Conv
Learns weights?	 No	 Yes
Reduces size?	 Yes	 Yes
Adds translation invariance?	 Yes	 Not inherently
Used in older CNNs (LeNet, AlexNet)?	 Yes	 Yes (for later layers)
Used in ResNet, EfficientNet?	Sometimes replaced with stride-2 convs	 Yes

---

Would you like me to show a small PyTorch example comparing max pooling and stride 2 convolution side-by-side with output shapes?

You said:

what's a normal amount of filters to have usually for the first conv layer?

ChatGPT said:

Excellent question — the number of filters (also called *output channels*) in the first convolutional layer is one of the most important architectural choices in a CNN.

Let's go over what's *typical* and *why*.

---

### 1. What the filters represent

Each filter (or kernel) learns to detect a certain pattern (edge, color blob, texture, etc.) in the input.

So more filters = more types of features that can be learned.

But — more filters also means:

- more parameters to train, and
  - more memory/computation cost.
- 

## 2. Typical values for the first conv layer

Here's what most CNN architectures use:

Dataset / Image Size	Typical Input Channels	Typical # of Filters (Conv1)	Examples
MNIST (1×28×28)	1 (grayscale)	16, 32	LeNet: 6 filters → 16 filters
CIFAR-10 (3×32×32)	3 (RGB)	32, 64	Simple CNNs often start with 32 filters
ImageNet (3×224×224)	3 (RGB)	64	VGG16, ResNet-18, AlexNet start with 64 filters

So in general:

-  For smaller datasets like MNIST/CIFAR → start with 16–64 filters in the first conv layer.
- 

## 3. Common pattern across layers

As you go *deeper* in the network, you usually double the number of filters:

Example for CIFAR-10:

```
# Simple CNN filter progression Conv2d(3, 32, kernel_size=3) Conv2d(32, 64, kernel_size=3) Conv2d(64, 128, kernel_size=3)
```

This lets early layers learn simple edges/colors, and deeper layers learn more abstract features (like shapes or textures).

---

## 4. Why not start too high?

Starting with too many filters (like 128+ in layer 1) can:

- Overfit on small datasets,
- Slow down training significantly,
- Require more GPU memory.

So the first conv layer is typically small and fast, and later layers do the heavy lifting.

---

### TL;DR

Dataset	Channels	Conv1 Filters (recommended)
MNIST (1×28×28)	1	16–32
CIFAR-10 (3×32×32)	3	32–64
ImageNet (3×224×224)	3	64

---

Would you like me to show a short PyTorch example of a CNN with a “good” starting structure for MNIST and CIFAR-10 (with typical filter counts)?

You said:

what's the default value for padding?

ChatGPT said:

Good question 

For `torch.nn.Conv2d`, the default value of `padding` is:

```
padding = 0
```

### What that means:

If you don't specify `padding`, then no extra border pixels are added around the input image.

That causes your output feature map to shrink depending on your kernel size and stride.

---

## Example

Let's say you have:

```
nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1)
```

and your input image is 32×32.

Then:

```
Output height/width = (32 - 3 + 2*0) / 1 + 1 = 30
```

So the output will be 30×30, not 32×32.

---

## To keep the same spatial size

If you want the output size to match the input (for stride=1), you can use:

```
padding = kernel_size // 2
```

So for a 3×3 kernel:

```
padding = 1
```

That's often called "same padding" (as in TensorFlow).

---

## TL;DR

Parameter	Default	Effect
padding	0	No padding → output shrinks
stride	1	Moves kernel 1 pixel at a time
kernel_size	(must set)	Defines how much shrinking happens

Would you like me to show how to calculate the exact output size formula with padding and stride (like how PyTorch does internally)?

You said:

when do you usuakky want to do paddoing of 0s, 1s, and beyond?

ChatGPT said:

Excellent question — this is one of the subtler design choices in CNNs. Padding might look like a small detail, but it has a big effect on how your network behaves.

Let's break it down clearly 

---

## What Padding Does

Padding adds extra pixels (usually zeros) around the image before convolution.

This changes:

- how many times the kernel "fits" on the image, and
  - whether edge pixels get equal attention.
- 

### 1. Padding = 0 ("valid" convolution)

 The kernel never goes outside the image boundary.

- Output shrinks: each conv reduces width/height by `(kernel_size - 1)` (for stride=1).
- Edge features may be lost since borders are used less often.

Use when:

- You *want* to reduce spatial dimensions gradually.
- You're okay losing border information (e.g., small objects in center).
- You're following classic architectures (like LeNet).

Example:

Input 32×32, kernel 3×3, stride 1 → output 30×30.

---

### 2. Padding = 1 (for 3×3 kernels) — “same” padding

→ This keeps the output same size as input (when stride=1).

This is the most common choice in modern CNNs.

Use when:

- You want to maintain spatial resolution throughout (common in ResNet, VGG).
- You're stacking many conv layers and don't want the feature map to shrink too fast.
- You care about border features (edges, corners).

Example:

Input 32×32, kernel 3×3, stride 1 → output 32×32.

---

### 3. Padding > 1

→ This gives extra context around borders or compensates for large kernels.

Use when:

- You're using larger kernels (e.g., 5×5 or 7×7).
- You want to preserve size even with those large kernels.
- You're building deeper architectures (like the first layer of AlexNet: 11×11 kernel, padding=2).

Rule of thumb:

To keep size the same:

```
padding = kernel_size // 2
```

### Summary Table

Padding	Typical Kernel	Effect	When to Use
0	Any	Output shrinks	You want downsampling or don't need border info

---

1	$3 \times 3$	Same output size	Most CNNs, balanced spatial preservation
2	$5 \times 5$	Same output size	Large kernels, early conv layers
3+	$7 \times 7+$	Same output size	Very large receptive fields (e.g., AlexNet, ResNet first layer)

---

### ✓ TL;DR:

- Padding 0 → shrinks image (old-school CNNs).
  - Padding 1 → keeps same size (modern CNNs).
  - Padding >1 → used with big kernels.
- 

Would you like me to show side-by-side how the feature map size changes for each padding value with an example input (say  $32 \times 32$ , stride 1, kernel 3 or 5)?

You said:

you say padding = 1 keeps output same size as input?

But let's say we have an input of  $10 \times 10$ , and kernel of  $5 \times 5$ , and padding of 1, so now it's like  $12 \times 12$ . When you have the kernel pass over, wouldn't it be like  $12 - 5 = 7$  as the output?

ChatGPT said:

Excellent — you're absolutely right to question that 🤓

Let's go through this *carefully* — because your intuition is spot on, and it'll clarify exactly when padding = 1 preserves size and when it doesn't.

---



### The convolution output size formula

For each spatial dimension (height or width):

$$\text{Output size} = (W - K + 2P)S + 1$$

Output size =

$S$

$$(W - K + 2P)$$

+1

Where:

- $W$
  - $W$  = input size
  - $K$
  - $K$  = kernel size
  - $P$
  - $P$  = padding
  - $S$
  - $S$  = stride
- 

### Example: your case

- $W=10$
- $W=10$
- $K=5$
- $K=5$
- $P=1$
- $P=1$
- $S=1$
- $S=1$

$$\text{Output} = (10 - 5 + 2(1))1 + 1 = 8$$

Output=

1

$$(10 - 5 + 2(1))$$

$$+1 = 8$$

✓ So yes — the output is  $8 \times 8$ , not  $10 \times 10$ .

---

💡 Why then does "padding = 1 keeps the size the same" come up so often?

Because that's only true when:

$$K=3, P=1, S=1$$

$$K=3, P=1, S=1$$

Then:

$$\text{Output} = (W - 3 + 2(1)) + 1 = W$$

$$\text{Output} = (W - 3 + 2(1)) + 1 = W$$

So the “same padding” rule that keeps dimensions equal is:

$$P = (K - 1)2$$

$$P =$$

$$2$$

$$(K - 1)$$

—but that only gives an integer

$$P$$

$P$  when the kernel size

K

$K$  is odd (3, 5, 7, ...).

---

### So in your case (5×5 kernel)

If you wanted to keep the same spatial size (10×10), you'd need:

$$P = (K-1)2 = 2$$

$$P =$$

$$2$$

$$(K-1)$$

$$= 2$$

 So:

```
nn.Conv2d(..., kernel_size=5, padding=2)
```

→ Keeps same size.

---

### Summary

Kernel Size	Padding	Keeps Same Size?
3×3	1	 Yes
5×5	1	 No (output shrinks)
5×5	2	 Yes
7×7	3	 Yes

---

 TL;DR:

“Padding = 1 keeps output same size” is only true for  $3 \times 3$  kernels.

In general, use:

$$P=(K-1)/2$$

$$P=$$

$$2$$

$$(K-1)$$

to preserve spatial size.

---

Would you like me to show a quick PyTorch example comparing the actual output shapes for those padding values (so you can see it in action)?

You said:

also, what filter sizes do we usually use across different conv layers?

ChatGPT said:

Fantastic question — this is one of those things that separates “just stacking layers” from designing effective CNN architectures.

Let’s go over how filter (kernel) sizes are typically chosen across different convolutional layers and *why* ↴

---

## ✳️ 1. General idea

The kernel size (filter size) controls how much of the input each neuron “sees” — its *receptive field*.

So:

- Small kernels (e.g.,  $3 \times 3$ ) → capture local patterns (edges, corners, small textures).
- Larger kernels (e.g.,  $5 \times 5$ ,  $7 \times 7$ ) → capture more global context (shapes, objects).

Most modern CNNs prefer many small kernels stacked deeply rather than one large kernel.

---

## 2. Typical filter sizes by layer depth

Layer Depth	Typical Kernel Size	Purpose	Example Networks
First conv layer	$3 \times 3$ , $5 \times 5$ , sometimes $7 \times 7$	Capture low-level edges, colors, corners	AlexNet ( $11 \times 11$ ), ResNet ( $7 \times 7$ ), VGG ( $3 \times 3$ )
Intermediate layers	$3 \times 3$ (almost universal)	Combine local patterns into shapes	VGG, ResNet, DenseNet
Deeper layers	$3 \times 3$ or $1 \times 1$	Compact feature maps, mix channel info	ResNet uses $1 \times 1$ for bottlenecks
Final conv before FC	$1 \times 1$	Channel mixing or dimensionality reduction	Inception, ResNet

---

## 3. Why $3 \times 3$ is the gold standard

- Smaller kernels = fewer parameters, less overfitting.
- Stacking two  $3 \times 3$  layers  $\approx$  receptive field of one  $5 \times 5$  kernel — but with fewer weights and more non-linearities (better expressiveness).

Example:

- One  $5 \times 5$  conv  $\rightarrow$  25 weights per channel.
- Two  $3 \times 3$  convs  $\rightarrow$  18 weights total ( $9 + 9$ ), same receptive field.

So  $3 \times 3$  became the default building block in almost all modern CNNs.

---

## 4. When to use others

Kernel Size	When to Use	Notes
1×1	To change # of channels (dim reduction or mixing)	Often used in ResNet, Inception
3×3	Default, balanced	Used in almost every conv layer
5×5	To capture wider patterns	Usually replaced by multiple 3×3s
7×7+	In very first layer for large inputs (like 224×224)	ResNet, AlexNet

## ✳️ 5. Example filter progression

Here's what a typical CNN looks like for CIFAR-10 (32×32 RGB):

```
Conv2d(3, 32, kernel_size=3, padding=1) Conv2d(32, 64, kernel_size=3,
padding=1) MaxPool2d(2, 2) Conv2d(64, 128, kernel_size=3, padding=1)
Conv2d(128, 128, kernel_size=3, padding=1) MaxPool2d(2, 2)
```

All 3×3 kernels, but as depth increases, filters double — so the network learns more *what* to look for, not *where*.

## ✓ TL;DR

Layer	Kernel Size	Common Choice	Reason
Early	3×3–7×7	3×3 or 5×5	Capture local textures
Middle	3×3	Default everywhere	Efficient + expressive
Deep	1×1	Channel mixing	Reduces computation

---

Input large images	7x7	Initial feature extraction	Used in ResNet/AlexNet
--------------------	-----	----------------------------	------------------------

---

Would you like me to show an example CNN architecture (PyTorch) that uses these kernel patterns across layers — like how a typical small CIFAR-10 network might look?

You said:

what are the parameters of MaxPool2d?

ChatGPT said:

Perfect question — because `torch.nn.MaxPool2d` looks simple, but its parameters control *how aggressively* you downsample your feature maps.

Let's break it down 

---



## `torch.nn.MaxPool2d` — Overview

This layer downsamples spatial dimensions (height, width) by taking the maximum value in each local region (like a sliding window).

---

## Syntax

```
torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1,  
                    return_indices=False, ceil_mode=False)
```

---

## Parameters Explained

Parameter	Type	Description
<code>kernel_size</code>	int or tuple	Size of the window to take the max over (e.g. <code>2</code> , <code>3</code> , <code>(2, 2)</code> ). Determines how big the pooling area is.
<code>stride</code>	int or tuple, optional	How far the pooling window moves each step.  Default: <code>stride = kernel_size</code> .

---

---

<code>padding</code>	int or tuple, optional	Adds zeros around the input before pooling. Usually 0.
<code>dilation</code>	int or tuple, optional	Controls spacing between kernel elements (rarely used for pooling). Default = 1.
<code>return_indices</code>	bool, optional	If <code>True</code> , returns both max values and their indices — used for unpooling layers (like in segmentation).
<code>ceil_mode</code>	bool, optional	If <code>True</code> , uses <code>ceil()</code> instead of <code>floor()</code> when computing output size. Helps with uneven input sizes.

---

## Output size formula

For each spatial dimension:

$$\text{Output size} = \lfloor \frac{(W-K+2P)S}{S} \rfloor + 1$$

$$\text{Output size} = \lfloor \frac{(W-K+2P)}{S} \rfloor + 1$$

$$S$$

$$(W-K+2P)$$

$$\rfloor + 1$$

or if `ceil_mode=True`:

$$\text{Output size} = \lceil \frac{(W-K+2P)S}{S} \rceil + 1$$

$$\text{Output size} = \lceil \frac{(W-K+2P)}{S} \rceil + 1$$

$$S$$

$$(W-K+2P)$$

]+1

---

## 🔍 Example 1 — Standard pooling

```
pool = nn.MaxPool2d(kernel_size=2, stride=2) x = torch.randn(1, 3, 32, 32)  
y = pool(x) print(y.shape)
```

✓ Output:

```
torch.Size([1, 3, 16, 16])
```

It halves the height and width (from 32×32 → 16×16).

---

## 🔍 Example 2 — Overlapping pooling

```
pool = nn.MaxPool2d(kernel_size=3, stride=2)
```

Here, windows overlap — gives smoother downsampling.

---

## ✓ TL;DR — Common settings

Goal	Common Settings
Standard downsampling	<code>kernel_size=2, stride=2</code>
Slightly smoother overlap	<code>kernel_size=3, stride=2</code>
Maintain size (rare)	<code>stride=1</code>
Semantic segmentation (unpooling later)	<code>return_indices=True</code>

---

Would you like me to show how MaxPool2d affects feature map shapes across multiple conv layers in a small CNN example (so you can see spatial size progression)?

You said:

also, is it better to use maxpool2d or avgpool2d?

ChatGPT said:

Great follow-up — this one actually touches on a design choice that depends on *what*

*you want your network to learn* 



## MaxPool2d vs AvgPool2d — Core Idea

Both are downsampling (reducing spatial resolution), but:

Operation	What it does	Intuition
MaxPool2d	Takes maximum value in each window	Keeps the <i>strongest</i> or most <i>dominant feature</i> (good for edge or texture detection)
AvgPool2d	Takes average value in each window	Keeps the <i>overall presence</i> of a feature (smoother, less aggressive)



## Key Differences in Practice

Aspect	MaxPool2d	AvgPool2d
Used in	Most CNNs (e.g., LeNet, AlexNet, VGG, ResNet pre-avg stage)	Some architectures (e.g., Global Average Pooling in classification heads)
Effect	Preserves strong activations → good for detecting key patterns like edges or shapes	Smooths features → better for estimating <i>presence</i> rather than <i>location</i>
Invariance	More robust to small shifts (keeps dominant activations)	Can lose contrast between features
Information loss	Higher (only keeps max)	Lower (retains average info)
Tendency	Encourages sparse, high-level activations	Encourages distributed, smoother activations

---

## Typical Usage

Layer location	Typical pooling used	Reason
Early / mid CNN layers	 MaxPool2d(kernel_size=2, stride=2)	Emphasizes strong edges and features
Final classification stage	 AvgPool2d (or AdaptiveAvgPool2d(1))	Used to <i>average spatial info</i> before the fully connected layer (common in ResNet, MobileNet, etc.)
Autoencoders / segmentation	Often skip pooling or use AvgPool2d	Need to preserve spatial detail

## Example

```
import torch import torch.nn as nn x = torch.tensor([ [[1., 2.], [3., 4.]]  
]).unsqueeze(0) # shape (1,1,2,2) maxp = nn.MaxPool2d(2) avgp =  
nn.AvgPool2d(2) print(maxp(x)) # tensor([[[[4.]]]]) print(avgp(x)) #  
tensor([[[[2.5]]]])
```

- Max pooling highlights the strongest feature (4)
- Avg pooling smooths it (2.5)

---

## TL;DR

Scenario	Recommended Pooling
Feature extraction (classification CNNs)	MaxPool2d
Smoothing / regularization	AvgPool2d

---

Global summarization (e.g., before fully connected layer)	Global AvgPool2d (or AdaptiveAvgPool2d((1,1)))
Maintaining detail (segmentation, autoencoders)	Often no pooling or AvgPool2d

---

If you tell me what kind of network you're designing (e.g., image classifier, autoencoder, object detector), I can tell you exactly *which pooling type and size* would fit best. Would you like that?

You said:

also, do you not have a RELU activation for any of the filters un til the very end?

ChatGPT said:

Excellent question — and actually, it's the opposite of that 😊

You *do* typically apply ReLU (or another nonlinearity) after every convolutional layer, *not* just at the end.

---

### 🧠 Why we add ReLU after each conv layer

Each convolution operation is linear — it's just a weighted sum of nearby pixels. If you stacked multiple convolutional layers *without* activation functions, the entire stack would still be just one big linear transformation — and that wouldn't let your CNN learn complex, nonlinear patterns.

So after each convolution, you insert a nonlinearity like ReLU to let the network learn more expressive, nonlinear representations.

---

### ✓ Typical pattern

You'll usually see something like:

```

self.conv_block = nn.Sequential( nn.Conv2d(in_channels=3, out_channels=16,
    kernel_size=3, stride=1, padding=1), nn.ReLU(), # activation for the
    filters nn.MaxPool2d(kernel_size=2), # downsample )

```

Then maybe:

```

self.conv_block2 = nn.Sequential( nn.Conv2d(16, 32, 3, 1, 1), nn.ReLU(),
    nn.MaxPool2d(2) )

```

After a few of these, you flatten the output and pass it to fully connected (linear) layers, which also have ReLUs in between:

```

self.fc = nn.Sequential( nn.Linear(32 * 7 * 7, 128), nn.ReLU(),
    nn.Linear(128, 10) # final output layer (no ReLU here!) )

```

## Summary

Stage	ReLU used?	Why
After each Conv2D	 Yes	Introduce nonlinearity so filters can detect complex features
After each Linear (hidden)	 Yes	Same reason — nonlinear decision boundaries
After the final output layer	 Usually no	You usually apply Softmax or Sigmoid instead, depending on the task

So:

You do use ReLU after *each convolution and hidden layer*, but not after the final output layer (since you want it to represent class scores or probabilities instead).

Would you like me to show a minimal full CNN example (with conv, pooling, ReLU, and classifier layers) for MNIST or CIFAR to illustrate this structure?

You said:

do you pool after the last conv layer too?

ChatGPT said:

Excellent question — and this one actually depends on the architecture and your design goals.

Here's the breakdown 

---

### 1. Pooling after **most** conv layers — but not always the last one

Typically, you do pooling after the early or middle convolutional layers to progressively reduce spatial size (height × width) and keep computation manageable.

But for the final conv block, you have two main choices:

#### **Option A: Pool after the last conv (common in simpler models)**

This is common in classic CNNs like LeNet or early VGG-style networks.

You might see something like:

```
Conv → ReLU → Pool  
Conv → ReLU → Pool  
Flatten → Linear → ...
```

Pooling here shrinks the feature map one last time before flattening.

#### **Option B: Skip pooling, or use *global average pooling***

Modern architectures (like ResNet, Inception, EfficientNet) often don't use MaxPool2d at the end.

Instead, they either:

- Skip pooling entirely and flatten the last feature map directly, or

- Use `nn.AdaptiveAvgPool2d((1, 1))`, which averages each feature map into one scalar — this is called Global Average Pooling.

That's often cleaner than flattening a huge tensor, since it keeps the model small and position-invariant.

---

## Intuition

- Early pooling: reduces computational load and makes filters capture more global structure.
  - Late pooling: can throw away spatial detail right before classification — sometimes that's okay, sometimes not.
  - No pooling at end: keeps richer spatial info for final classification.
- 

## Example

```
self.conv_layers = nn.Sequential( nn.Conv2d(3, 16, 3, padding=1),  
nn.ReLU(), nn.MaxPool2d(2), # pool early nn.Conv2d(16, 32, 3, padding=1),  
nn.ReLU(), # No pooling here – keep info for flatten ) self.fc_layers =  
nn.Sequential( nn.Linear(32 * 16 * 16, 128), nn.ReLU(), nn.Linear(128, 10)  
)
```

or with global average pooling:

```
self.gap = nn.AdaptiveAvgPool2d((1, 1))
```

Then before feeding into `Linear`, you'd do:

```
x = self.gap(x) x = torch.flatten(x, 1)
```

---

## Rule of thumb

Situation	Pool after last conv?
Simple CNNs (e.g., MNIST, CIFAR small models)	 Yes, usually MaxPool
Deep architectures (ResNet, etc.)	 No, use global avg pooling instead

---

When you want spatial detail preserved

✗ No

---

Would you like me to show a clean PyTorch CNN example *both ways* (with and without final pooling) so you can compare the architectures?

You said:

so when converting from the conv stuff to the fully connected layer, do you have to flatten the entire 3D map?

ChatGPT said:

Yes — exactly! ✓

When you go from convolutional layers (which output a 3D feature map: channels × height × width) to a fully connected (linear) layer, you need to flatten the spatial dimensions into a 1D vector per sample.

---

## 🧠 Why?

- Fully connected layers (`nn.Linear`) expect 2D input:  
`(batch_size, num_features)`
  - where `num_features` is the total number of inputs per sample.
  - A convolutional output has shape:  
`(batch_size, channels, height, width)`
  - So you flatten channels × height × width into a single dimension.
- 

## 🔍 Example

Suppose your last conv output is:

```
x.shape # torch.Size([8, 32, 7, 7])
```

- 8 = batch size

- 32 = number of channels / filters
- $7 \times 7$  = spatial dimensions

You flatten each sample:

```
x = x.view(x.size(0), -1) # shape becomes [8, 32*7*7] = [8, 1568]
```

Then feed into a linear layer:

```
fc = nn.Linear(32*7*7, 128) x = fc(x) # output shape [8, 128]
```

---

## ⚡ TL;DR

- Yes, flatten everything from conv output: channels  $\times$  height  $\times$  width
- Do not include the batch size in flattening
- Then feed it into `nn.Linear` for classification

---

💡 Note: In modern architectures, sometimes people use Global Average Pooling instead of flattening, which reduces each channel to a single number — this reduces parameters drastically and keeps it more position-invariant.

---

If you want, I can draw a visual example showing conv output  $\rightarrow$  flatten  $\rightarrow$  linear layer so it's crystal clear. Do you want me to do that?

You said:  
what does adaptive avg pool layer do?

ChatGPT said:

Ah, `AdaptiveAvgPool2d` is super useful — it's like an "automatic resizing" pooling layer.

Let me explain clearly.

---

## ✳️ 1. Core idea

- Normal `AvgPool2d` requires you to choose kernel size, stride, etc., manually.
- `AdaptiveAvgPool2d` lets you specify the output size you want, and PyTorch figures out the pooling kernel and stride for you automatically.

So instead of saying “pool 2×2, stride 2”, you say:

```
nn.AdaptiveAvgPool2d((1, 1))
```

...and the layer will downsample any input spatial size to 1×1 per channel.

---

## 2. Why it's useful

### 1. Input-size agnostic:

- Your network can accept images of different sizes, and the adaptive pooling will always produce the same output size for the next layer (often before the fully connected classifier).

### 2. Global feature summarization:

- Often used as Global Average Pooling in modern architectures (ResNet, EfficientNet, MobileNet).
  - Reduces each feature map to a single number (the average activation across height and width).
- 

## 3. Example

Suppose your conv output is:

```
x.shape # torch.Size([8, 64, 7, 7])
```

Apply global adaptive average pooling:

```
gap = nn.AdaptiveAvgPool2d((1, 1)) x = gap(x) print(x.shape)
```

Output:

```
torch.Size([8, 64, 1, 1])
```

Then flatten before fully connected layer:

```
x = x.view(x.size(0), -1) # [8, 64]
```

Now it's ready for a linear layer, no matter the original height/width of the conv output.

---

## 4. Key points

Feature	Benefit
Specify output size instead of kernel/stride	Automatically adapts to input
Common usage: <code>(1, 1)</code>	Produces 1 value per channel (global average)
Avoids hardcoding feature map size	Makes networks flexible to different image sizes

In short: AdaptiveAvgPool2d is “set the size you want, forget the stride/kernel math” — perfect for connecting conv layers to fully connected layers.

If you want, I can make a mini PyTorch CNN example showing AdaptiveAvgPool2d before the classifier so you see how it replaces flattening and regular pooling. Do you want me to do that?

You said:  
typically, ahs the last pooling before the linear layer work?

ChatGPT said:

Yes — in typical CNNs, the last pooling before the linear layer is handled in one of two main ways, depending on the architecture:

## ① Standard Max/Average Pooling

- You can use a regular `MaxPool2d` or `AvgPool2d` as the last pooling layer.
- It reduces the spatial dimensions once more before flattening for the fully connected (linear) layers.
- Works well in smaller CNNs, like those for MNIST or CIFAR-10.

Example:

```
self.conv_layers = nn.Sequential( nn.Conv2d(3, 32, 3, padding=1),
nn.ReLU(), nn.MaxPool2d(2), # early pooling nn.Conv2d(32, 64, 3,
```

```

padding=1), nn.ReLU(), nn.MaxPool2d(2) # last pooling ) # flatten after
last pooling x = x.view(x.size(0), -1) x = self.fc(x)

```

- The last pooling here reduces feature maps, e.g., from  $64 \times 8 \times 8 \rightarrow 64 \times 4 \times 4$  before flattening.

## 2 Global Average Pooling (modern approach)

- Instead of a fixed-size pooling (like  $2 \times 2$ ), you can use `AdaptiveAvgPool2d((1,1))`.
- This reduces each channel to a single number regardless of input spatial size.
- Advantages:
  - Fewer parameters (less overfitting)
  - Position invariance (focus on “presence of features” rather than exact location)
  - Works even if input size changes

Example:

```

self.conv_layers = nn.Sequential( nn.Conv2d(3, 32, 3, padding=1),
nn.ReLU(), nn.Conv2d(32, 64, 3, padding=1), nn.ReLU() ) # global average
pooling instead of last maxpool self.gap = nn.AdaptiveAvgPool2d((1, 1)) x
= self.gap(x) # output shape: (batch, 64, 1, 1) x = torch.flatten(x, 1) #
shape: (batch, 64) x = self.fc(x) # feed into linear layer

```

## ✓ Rule of Thumb

Network type	Last pooling before FC?
Small/simple CNNs (CIFAR-10, MNIST)	✓ Regular Max/Avg Pooling, then flatten
Modern architectures (ResNet, MobileNet)	✓ Global Average Pooling → flatten → FC

💡 Summary:

- You almost always do some form of pooling or global averaging before the fully connected layers.
  - MaxPool/AvgPool is fine for small CNNs.
  - Global AvgPool is more elegant for deeper networks and flexible input sizes.
- 

If you want, I can draw a diagram showing conv layers → pooling → flatten → linear for both approaches — it really makes the flow clear. Do you want me to do that?

You said:

does a fully connected layer mean that you need as many perceptrons as there are inputs?

ChatGPT said:

Not quite — this is a common point of confusion. Let me clarify carefully.

---

## 1 What a fully connected (linear) layer really is

- In PyTorch, a fully connected layer is `nn.Linear(in_features, out_features)`.
  - Each of the `out_features` neurons receives input from all `in_features` values.
  - But the number of neurons (`out_features`) does not have to equal the number of inputs.
- 

## 2 How it works

Suppose you have:

```
fc = nn.Linear(in_features=64, out_features=10)
```

- Input vector: 64 values (could be flattened conv output, or features from previous layer)
- Layer has 10 neurons (`out_features=10`)
- Each neuron has 64 weights + 1 bias connecting to all 64 inputs
- Output: a vector of 10 values

So the number of neurons is your design choice, not forced to match input size.

---

## 3 Examples

## Example A: Flattened conv output to classifier

```
x = torch.randn(8, 64, 4, 4) # batch=8, 64 channels, 4x4 spatial x =
x.view(x.size(0), -1) # flatten -> [8, 64*4*4] = [8, 1024] fc =
nn.Linear(1024, 128) # 128 neurons, not 1024 y = fc(x) # output shape [8,
128]
```

- 1024 inputs, 128 neurons → totally fine.
- The layer is “fully connected” because each neuron sees all 1024 inputs.

## Example B: Final output layer for classification

```
fc_final = nn.Linear(128, 10) # 10 classes
```

- 128 inputs → 10 neurons, one per class
- Still fully connected — each neuron receives all 128 inputs.

---

### ✓ TL;DR

- Fully connected = all inputs connected to all outputs.
  - Number of outputs (`out_features`) is your choice, not equal to input size.
  - What matters is connection pattern, not the number of neurons.
- 

If you want, I can show a diagram showing flattened conv output → fully connected layer → output, including weights, so you can see visually why the number of neurons doesn't have to match the inputs.

Do you want me to do that?

You said:

if say i end up with 64 filters in the end, and call adaptiveavgpoollayer to make each of them 1x1, how many perceptrons would you say i need?

ChatGPT said:

Perfect — let's reason through this step by step.

---

## 1 What your output looks like after conv + adaptive avg pooling

Suppose:

```
x.shape # after last conv layer # (batch_size, 64, H, W) H and W can be  
anything
```

Then you do global adaptive average pooling:

```
gap = nn.AdaptiveAvgPool2d((1, 1)) x = gap(x) print(x.shape)
```

- Output shape: `(batch_size, 64, 1, 1)`
- Each of the 64 channels has been reduced to a single scalar (the average activation over the spatial map).

---

## 2 Flatten before fully connected layer

```
x = x.view(x.size(0), -1) print(x.shape)
```

- Output shape: `(batch_size, 64)`
- So you have 64 input values per sample.

---

## 3 Choosing the number of perceptrons in the linear layer

- Input size = 64 (from the 64 pooled channels)
- Output size / number of perceptrons = your design choice, depending on task:

**Example:**

- Hidden layer: 128 neurons

```
fc1 = nn.Linear(64, 128) # 64 inputs, 128 neurons
```

- Final classification layer: say 10 classes

```
fc2 = nn.Linear(128, 10) # 128 inputs, 10 outputs
```

- Notice: you don't need 64 perceptrons just because you have 64 inputs.
- You could even go directly from 64 → 10 if you want a simple network:

```
fc = nn.Linear(64, 10)
```

---

 **Rule of thumb**

Layer type	Input size	Output size (neurons)
After global avg pool	#channels (64)	Hidden layer: 64–256 neurons (design choice)
Final output	Hidden layer size	#classes (e.g., 10 for CIFAR/MNIST)

💡 Key point:

- Fully connected layer sees all 64 inputs per neuron — that's what makes it “fully connected”.
- The number of neurons (perceptrons) is flexible, not forced to match the number of inputs.

If you want, I can draw a visual example of 64-channel conv output → adaptive avg pooling → fully connected layer to show exactly how the numbers match up.

Do you want me to do that?

You said:

but for CNN, after pooling how many layers do you want relative to the size of the last filter?

ChatGPT said:

Ah — now you're asking about the relationship between the last convolutional output (number of channels / filters) and the size of the fully connected layers. Let's break it down carefully.

## 1 What you have after convolution + pooling

Suppose your last conv block outputs:

(batch\_size, C, H, W)

Where:

- $C$  = number of filters (channels)
- $H \times W$  = spatial dimensions (after pooling)

If you use global average pooling (AdaptiveAvgPool2d((1,1))):

$(batch\_size, C, 1, 1) \rightarrow \text{flatten} \rightarrow (batch\_size, C)$

So your fully connected layer input size = number of filters  $C$ .

## 2 How to choose the fully connected layer size

There's no strict formula, but some rules of thumb:

Scenario	Recommendation
Small network / small dataset (MNIST, CIFAR-10)	Fully connected layer size $\approx$ number of channels $(C) \rightarrow 2-4 \times C$
Larger network	Can have one or more hidden layers, often $2-4 \times C$ neurons for first hidden FC layer
Final classification layer	Number of neurons = number of classes (e.g., 10 for CIFAR-10, 10 for MNIST)

So if your last conv layer has 64 filters, then:

- A hidden FC layer of 64  $\rightarrow$  128 neurons is reasonable.
- Then final output 128  $\rightarrow$  10 (for classification).

## 3 Why not match filter count exactly?

- The number of neurons in the fully connected layer doesn't need to match  $C$ .
- FC layers are flexible; you can:
  - Increase neurons (more capacity / expressive power)
  - Decrease neurons (regularization, fewer parameters)

## 4 Example for 64 filters

```
# after GAP: (batch_size, 64) fc = nn.Sequential( nn.Linear(64, 128), #  
hidden layer nn.ReLU(), nn.Linear(128, 10) # final output layer )
```

- Input: 64 → Hidden: 128 → Output: 10
  - Fully connected layer size is roughly 2×–4× the last filter count (common practice, not strict).
- 

## ✓ TL;DR

- Input to FC layer = number of channels after last conv / pooling
  - Hidden FC layer size = usually 1–4× number of channels, depends on model capacity
  - Output layer size = number of classes
- 

If you want, I can sketch a small CNN from conv → pooling → FC layers with channel numbers and neuron counts to show these ratios visually. Do you want me to do that?

You said:

here, should i be calling relu at the end ?

```
from model import Model  
import torch  
import torch.nn as nn  
import torch.optim as optim  
from torch.utils.data import DataLoader, TensorDataset  
  
class MLP_Manager(Model):  
    def doCustomProcessing(self):  
        # a mlp just expects a 1 dimensional array as its input, so we must flatten our 2D  
        # array to a 1D one.  
        self.mnistXTrain = self.mnistXTrain.reshape(self.mnistXTrain.shape[0],  
                                                    self.mnistXTrain.shape[1] * self.mnistXTrain.shape[2])  
        self.mnistXTest = self.mnistXTest.reshape(self.mnistXTrain.shape[0],  
                                                self.mnistXTrain.shape[1] * self.mnistXTrain.shape[2])
```

```

# may not make a difference, but just in case we transpose before flattening. Do
    this because for the cnn, we have to do this as
        # it expects channel first
        self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)
        self.cifarXTest = self.cifarXTest.transpose(0, 3, 1, 2)

# a mlp just expects a 1 dimensional array as its input, so we must flatten our 3D
    array to a 1D one.
    self.cifarXTrain = self.cifarXTrain.reshape(self.cifarXTrain.shape[0],
                                                self.cifarXTrain.shape[1] * self.cifarXTrain.shape[2] *
                                                self.cifarXTrain.shape[3])

    self.cifarXTest = self.cifarXTest.reshape(self.cifarXTest.shape[0],
                                              self.cifarXTest.shape[1] * self.cifarXTest.shape[2] *
                                              self.cifarXTest.shape[3])

# do the common pre processing before doing what's special to the MLP
    def __init__(self):
        super().__init__()
        # now, custom stuff done
        self.doCustomProcessing()

        class MLP(nn.Module):
    def __init__(self, complexity, dataset):
        super().__init__()
        layers = []

# first dimension is # of points, so the one after is the number of features
        if(dataset == "mnist"):
            numFeatures = self.mnistXTrain.shape[1]
            else:
                numFeatures = self.cifarXTrain.shape[1]

# do all the layers except for the final layer
        beforeHiddenLayerLength = -1

```

```

        if(complexity == "low"):
            numPerceptrons = 128
        layers.append(nn.Linear(numFeatures, numPerceptrons))
            layers.append(nn.ReLU())
        beforeHiddenLayerLength = numPerceptrons
        elif(complexity == "medium"):
            numPerceptronsFirst = 512
            numPerceptronsSecond = 256
            numPerceptronsThird = 128

        layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
            layers.append(nn.ReLU())

        beforeHiddenLayerLength = numPerceptronsThird
        elif(complexity == "high"):
            numPerceptronsFirst = 1024
            numPerceptronsSecond = 512
            numPerceptronsThird = 256
            numPerceptronsFourth = 128
            numPerceptronsFifth = 64
            numPerceptronsSixth = 32

        layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsThird, numPerceptronsFourth))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsFourth, numPerceptronsFifth))

```

```

        layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFifth, numPerceptronsSixth))
        layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsSixth

# now do the final layer
numClasses = 10
layers.append(beforeHiddenLayerLength, numClasses)
        layers.append(nn.ReLU())

self.model = nn.Sequential(*layers)

```

ChatGPT said:

Ah — I see exactly what's happening in your `MLP` code, and there are a couple of issues around activations and the final layer. Let me explain clearly.

---

## 1 Should you call `ReLU` at the end?

No, usually you don't apply `ReLU` on the final layer in a classification network.

- The final layer of an MLP (or CNN) that outputs class scores should usually be raw scores (logits).
- If you apply `ReLU` there, you clamp all outputs to  $\geq 0$ , which:
  - Removes the ability to represent negative logits
  - Can mess up `CrossEntropyLoss`, which expects raw scores

 Correct pattern:

```
layers.append(nn.Linear(beforeHiddenLayerLength, numClasses)) # DO NOT add
                                                               nn.ReLU() here
```

- Then, when training, you can feed the logits directly into `nn.CrossEntropyLoss()`, which internally applies softmax.
-

## 2 Other issues in your code

### a) The final layer append syntax is wrong

```
layers.append(beforeHiddenLayerLength, numClasses) # ✗ invalid
```

- `append` takes one object, not two numbers.
- It should be:

```
layers.append(nn.Linear(beforeHiddenLayerLength, numClasses))
```

---

### b) Flattening input handling

- In `MLP_Manager`, you are reshaping `mnistXTest` using  
`self.mnistXTrain.shape[0]` — that's likely a bug; it should be:

```
self.mnistXTest = self.mnistXTest.reshape(self.mnistXTest.shape[0],  
                                         self.mnistXTest.shape[1] * self.mnistXTest.shape[2])
```

- Otherwise your test set may be incorrectly reshaped.

---

## 3 Corrected final MLP snippet

```
# final layer numClasses = 10  
layers.append(nn.Linear(beforeHiddenLayerLength, numClasses)) # raw  
logits, no ReLU self.model = nn.Sequential(*layers)
```

---

### ✓ TL;DR

- Do NOT put ReLU after the last layer — leave it as raw scores.
- CrossEntropyLoss will handle converting logits to probabilities internally.
- All previous hidden layers should have ReLU.

---

If you want, I can rewrite your whole `MLP` class in a clean, correct version with low/medium/high complexity and correct activations so it's ready to train.

Do you want me to do that?

You said:

how would i do bath normalization or dropout in a CNN in python?

ChatGPT said:

Ah, perfect! Both Batch Normalization and Dropout are common techniques to stabilize and regularize CNNs in PyTorch. Let's go step by step.

---

## 1 Batch Normalization (nn.BatchNorm2d)

### ✓ What it does:

- Normalizes each feature map across the batch:
  - Makes the activations mean  $\approx 0$ , variance  $\approx 1$
  - Helps training converge faster and can improve generalization
- Usually applied after convolution and before or after ReLU (people often do Conv → BatchNorm → ReLU)

### 🔍 Example:

```
import torch.nn as nn
self.conv_block = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),
    nn.BatchNorm2d(32), # normalize 32 output channels
    nn.ReLU(),
    nn.MaxPool2d(2) )
```

- nn.BatchNorm2d(num\_features) → num\_features = number of channels coming out of the conv layer.

### Notes:

- BatchNorm learns scale and shift parameters (`gamma` and `beta`) during training.
- It behaves slightly differently during training vs evaluation (`model.train()` vs `model.eval()`).

## 2 Dropout (nn.Dropout / nn.Dropout2d)

### ✓ What it does:

- Randomly sets some activations to zero during training with probability  $p$ .
- Reduces overfitting by forcing the network not to rely too heavily on specific neurons.

### Example (fully connected / 1D):

```
self.fc_block = nn.Sequential( nn.Linear(128, 64), nn.ReLU(),
nn.Dropout(0.5), # 50% of neurons randomly zeroed nn.Linear(64, 10) )
```

### Example (spatial / 2D conv features):

```
self.conv_block = nn.Sequential( nn.Conv2d(32, 64, 3, padding=1),
nn.ReLU(), nn.Dropout2d(0.3) # drops entire channels randomly )
```

- Use `Dropout2d` for conv layers → drops entire channels, not individual pixels.
- Use `Dropout` for linear layers → drops individual neurons.

## 3 Typical pattern in CNN

```
self.conv_block = nn.Sequential( nn.Conv2d(3, 32, 3, padding=1),
nn.BatchNorm2d(32), nn.ReLU(), nn.MaxPool2d(2), nn.Dropout2d(0.2) # optional for conv layers ) self.fc_block = nn.Sequential(
nn.Linear(32*16*16, 128), nn.ReLU(), nn.Dropout(0.5), # for fully connected layers nn.Linear(128, 10) )
```

- BatchNorm → stabilizes and speeds up training
- Dropout → prevents overfitting

## ✓ TL;DR

Layer	Where to use	PyTorch class
Convolutional	After conv, before/after ReLU	<code>nn.BatchNorm2d, nn.Dropout2d</code>
Fully connected	After linear, before/after ReLU	<code>nn.Dropout</code>

If you want, I can write a full small CNN for CIFAR/MNIST showing Conv → BatchNorm → ReLU → Pool → Dropout → FC ready to train, so you see exactly where each one goes.

Do you want me to do that?

You said:

do you do dropout right before connecting to a linear layer always?

ChatGPT said:

Not always — it depends on where you want to regularize the network. Let me clarify the typical practices:

---

## 1 Dropout in fully connected layers (most common)

- Dropout is most often applied after hidden linear layers.
- Reason: fully connected layers have many parameters and are prone to overfitting.
- Example:

```
self.fc_block = nn.Sequential( nn.Linear(512, 256), nn.ReLU(),
nn.Dropout(0.5), # regularizes this hidden layer nn.Linear(256, 10) #
                  output layer, no dropout here )
```

- Key point: Do not put dropout after the final output layer — you want raw logits for classification.
- 

## 2 Dropout in convolutional layers (optional)

- Can also apply `nn.Dropout2d` after conv layers, but less common than in FC layers.
- Usually with small probability (0.1–0.3), because conv layers are spatially local and have fewer parameters than FC layers.

```
self.conv_block = nn.Sequential( nn.Conv2d(32, 64, 3, padding=1),
nn.ReLU(), nn.Dropout2d(0.2) )
```

---

### 3 When not to use dropout

- Output layer: Don't apply dropout here — you want the model to produce stable class logits.
  - BatchNorm layers: Some argue that combining BatchNorm + Dropout is unnecessary for conv layers, but it can still help for FC layers.
- 

#### ✓ TL;DR Rules

Layer type	Dropout?	Notes
Hidden linear layer	✓ Yes	Most common, 0.3–0.5 probability
Hidden conv layer	✓ Optional	Use <code>Dropout2d</code> , smaller probability
Output layer	✗ No	Keep logits intact
BatchNorm	✗ Usually no	BN already stabilizes activations

---

So, you don't always put dropout right before a linear layer — you put it after hidden layers that you want to regularize.

---

If you want, I can show a diagram of a small CNN + MLP with BatchNorm and Dropout in all the right places, so you can see visually where to insert them.

Do you want me to do that?

You said:

what about dropout specifically in CNNs, where is that usually applied?

ChatGPT said:

Ah, got it! Let's talk specifically about dropout in CNNs, which is a bit different from MLPs.

---

## 1 Dropout in convolutional layers

- Standard `nn.Dropout` drops individual neurons — in conv layers, this would drop individual pixels in a feature map.
- Usually, we use `nn.Dropout2d` (or `nn.Dropout3d`) for conv layers:
  - Drops entire channels randomly
  - Helps prevent the network from relying too much on a few feature maps
- Typically applied after conv + ReLU, optionally before pooling

Example:

```
self.conv_block = nn.Sequential( nn.Conv2d(32, 64, 3, padding=1),  
nn.ReLU(), nn.Dropout2d(0.2), # 20% of channels dropped nn.MaxPool2d(2) )
```

- Dropout in conv layers is less aggressive than in fully connected layers, because conv layers have fewer parameters and local connectivity.

---

## 2 Dropout in fully connected layers after CNN

- After flattening the conv output and before the first hidden linear layer:
  - Apply `nn.Dropout` with higher probability (0.3–0.5)
  - This is the most common use in CNNs

Example:

```
x = x.view(x.size(0), -1) # flatten x = self.fc1(x) x = nn.ReLU()(x) x =  
nn.Dropout(0.5)(x) # high probability in FC layer x = self.fc2(x) # output  
layer, no dropout
```

---

## 3 Key points

Layer type	Dropout type	Probability	Notes
Conv layer	<code>nn.Dropout2d</code>	0.1–0.3	After ReLU, optional
FC hidden layer	<code>nn.Dropout</code>	0.3–0.5	Most important place to prevent overfitting

---

Output layer

✗ None

— Keep raw logits for classification

- Order matters: Conv → ReLU → Dropout → Pool is common
  - Global Avg Pooling: If you use GAP before FC, dropout usually goes after GAP, before FC
- 

💡 TL;DR:

- In CNNs, dropout is optional for conv layers, but common and higher probability for FC layers.
  - Never apply dropout to the final output layer.
- 

If you want, I can sketch a full CNN showing exactly where Dropout2d and Dropout go, with conv → pooling → flatten → FC → output, so you can see visually.

Do you want me to do that?

You said:

how do i use SGD optimizer in python?

ChatGPT said:

Perfect question — let's go through how to use SGD optimizer (Stochastic Gradient Descent) in PyTorch step by step 

---

## 1 Basic setup

To use SGD in PyTorch:

- You define your model
- Define a loss function
- Create the optimizer with the model parameters

Example:

```

import torch import torch.nn as nn import torch.optim as optim # Example
model = nn.Linear(784, 10) # for MNIST (28x28 -> 784 input features)
# Loss function criterion = nn.CrossEntropyLoss() # SGD optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

```

## 2 Typical training loop

Here's how it's used during training:

```

for epoch in range(num_epochs): for inputs, labels in train_loader: #
    Forward pass outputs = model(inputs) loss = criterion(outputs, labels) #
    Backward pass optimizer.zero_grad() # clear previous gradients
    loss.backward() # compute gradients optimizer.step() # update parameters
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}")

```

## 3 Common SGD parameters

Parameter	Meaning	Typical value
lr	Learning rate	0.01–0.1
momentum	Adds inertia to updates, helps escape local minima	0.8–0.9
weight_decay	L2 regularization	1e-4 or 5e-4
nesterov	Use Nesterov accelerated gradient	True/False

Example with all options:

```

optimizer = optim.SGD( model.parameters(), lr=0.01, momentum=0.9,
                      weight_decay=5e-4, nesterov=True )

```

## 4 How it works conceptually

SGD updates weights using this rule:

$$w^{t+1} = w^t - \eta \cdot \nabla w L(w^t)$$

$w$

$t+1$

$= w$

$t$

$-\eta \cdot \nabla$

$w$

$L(w$

$t$

)

- $\eta$
- $\eta$  = learning rate
- $\nabla w L(w^t)$
- $\nabla$
- $w$
- 
- $L(w$
- $t$
- 
- ) = gradient of loss wrt weights
- momentum smooths out oscillations and speeds convergence

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer.zero_grad() loss.backward() optimizer.step()
```

That's all you need to use it in your training loop 🚀

---

Would you like me to show you how to compare SGD with Adam (including when to use each)?

You said:

will this unpack the parameters correctly?

```
from abc import ABC, abstractmethod
from tensorflow.keras.datasets import mnist, cifar10
```

```
class Model(ABC):
    @abstractmethod
    def doCustomProcessing(self):
        pass
    def retrieveDataset(self, setType):
        if setType == "mnist":
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
        else:
            (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

        # normalizing them. Converting pixel values between 0 - 255 to 0 - 1
        xTrain /= 255.0
        xTest /= 255.0

        return (xTrain, yTrain, xTest, yTest)

    # initialize method
    def __init__(self):
        (self.mnistXTrain, self.mnistYTrain, self.mnistXTest, self.mnistYTest) =
            self.retrieveDataset("mnist")
```

```

(self.cifarXTrain, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
    self.retrieveDataset("cifar")

        self.parameters = {
    "learningRate": [0.01, 0.001, 0.0001],
    "batchSize": [32, 64, 128],
    "optimizer": ["sgd", "adam"],
    "dropoutRate": [0.2, 0.5]
}

def sampleParams(self):
    return {key: random.choice(values) for key, values in self.parameters.items()}

from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

class MLP_Manager(Model):
    def doCustomProcessing(self):
# a mlp just expects a 1 dimensional array as its input, so we must flatten our 2D
array to a 1D one.
        self.mnistXTrain = self.mnistXTrain.reshape(self.mnistXTrain.shape[0],
                                                 self.mnistXTrain.shape[1] * self.mnistXTrain.shape[2])
        self.mnistXTest = self.mnistXTest.reshape(self.mnistXTest.shape[0],
                                                self.mnistXTest.shape[1] * self.mnistXTest.shape[2])

# may not make a difference, but just in case we transpose before flattening. Do
this because for the cnn, we have to do this as
        # it expects channel first
        self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)
        self.cifarXTest = self.cifarXTest.transpose(0, 3, 1, 2)

```

```

# a mlp just expects a 1 dimensional array as its input, so we must flatten our 3D
# array to a 1D one.
self.cifarXTrain = self.cifarXTrain.reshape(self.cifarXTrain.shape[0],
                                             self.cifarXTrain.shape[1] * self.cifarXTrain.shape[2] *
                                             self.cifarXTrain.shape[3])

self.cifarXTest = self.cifarXTest.reshape(self.cifarXTest.shape[0],
                                         self.cifarXTest.shape[1] * self.cifarXTest.shape[2] *
                                         self.cifarXTest.shape[3])

# do the common pre processing before doing what's special to the MLP
def __init__(self):
    super().__init__()
    # now, custom stuff done
    self.doCustomProcessing()

def validateModelParams(self, complexity):
    numValuesUsed = 16
    for _ in range(0, numValuesUsed, 1):
        sampledParams = self.sampleParams()
    self.currentModel = MLP(complexity, **sampledParams)

class MLP(nn.Module):
def __init__(self, complexity, dataset, learningRate, optimizer, dropoutRate,
            batchSize):
    super().__init__()
    layers = []

# first dimension is # of points, so the one after is the number of features
    if(dataset == "mnist"):
        numFeatures = self.mnistXTrain.shape[1]
    else:
        numFeatures = self.cifarXTrain.shape[1]

# do all the layers except for the final layer

```

```

        beforeHiddenLayerLength = -1
        if(complexity == "low"):
            numPerceptrons = 128
        layers.append(nn.Linear(numFeatures, numPerceptrons))
            layers.append(nn.ReLU())
        beforeHiddenLayerLength = numPerceptrons
        elif(complexity == "medium"):
            numPerceptronsFirst = 512
            numPerceptronsSecond = 256
            numPerceptronsThird = 128

        layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
            layers.append(nn.ReLU())

        beforeHiddenLayerLength = numPerceptronsThird
        elif(complexity == "high"):
            numPerceptronsFirst = 1024
            numPerceptronsSecond = 512
            numPerceptronsThird = 256
            numPerceptronsFourth = 128
            numPerceptronsFifth = 64
            numPerceptronsSixth = 32

        layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsThird, numPerceptronsFourth))
            layers.append(nn.ReLU())

```

```

        layers.append(nn.Linear(numPerceptronsFourth, numPerceptronsFifth))
                    layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFifth, numPerceptronsSixth))
                    layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsSixth

# first, remove on hidden layers
layers.append(nn.Dropout(dropoutRate))

# now do the final layer
numClasses = 10
layers.append(nn.Linear(beforeHiddenLayerLength, numClasses))

self.model = nn.Sequential(*layers)

self.criterion = nn.CrossEntropyLoss()
self.num_epochs = 50

if(optimizer == "adam"):
    self.optimizer = optim.Adam(self.parameters(), lr = learningRate)
else:
    self.optimizer = optim.SGD(self.parameters(), lr = learningRate)

```

ChatGPT said:

You're *very* close — that's a solid structure — but there are a few small issues to fix before your parameter unpacking and model creation will work correctly. Let's go over what's happening and what needs adjustment 

### Problem 1: validateModelParams() unpacking mismatch

Your code does:

```

sampledParams = self.sampleParams() self.currentModel = MLP(complexity,
**sampledParams)

```

But your `MLP.__init__()` expects:

```
def __init__(self, complexity, dataset, learningRate, optimizer,
             dropoutRate, batchSize):
```

That means your `sampleParams()` dictionary only contains:

```
{ "learningRate": ..., "batchSize": ..., "optimizer": ..., "dropoutRate": ...
... }
```

so `dataset` is missing.

→ Fix: Include the dataset name explicitly when calling MLP:

```
self.currentModel = MLP(complexity, "mnist", **sampledParams)
```

(or `"cifar"` depending on which one you're validating).

---

⚠ **Problem 2:** `self.mnistXTrain / self.cifarXTrain` access inside `MLP`

Your `MLP` class directly references `self.mnistXTrain` and `self.cifarXTrain`, but those belong to the manager (`MLP_Manager`), not the model itself.

→ Fix Option 1 (recommended):

Pass the dataset size (number of features) into the model instead of relying on attributes from the manager.

Example:

```
numFeatures = xTrain.shape[1] mlp = MLP(complexity,
                                         numFeatures=numFeatures, **sampledParams)
```

Then modify `MLP`'s `__init__` to accept `numFeatures`:

```
def __init__(self, complexity, numFeatures, learningRate, optimizer,
             dropoutRate, batchSize):
```

and remove the `if(dataset == "mnist")` check.

---

## ⚠ Problem 3: Layer creation typo

This line will throw an error:

```
layers.append(beforeHiddenLayerLength, numClasses)
```

because `append()` takes a single argument.

→ Fix:

```
layers.append(nn.Linear(beforeHiddenLayerLength, numClasses))
```

## ✓ Cleaned-up version of the key parts

Here's how your `validateModelParams` and `MLP` constructors should look:

```
def validateModelParams(self, complexity, dataset="mnist"): numValuesUsed = 16 for _ in range(numValuesUsed): sampledParams = self.sampleParams() if dataset == "mnist": numFeatures = self.mnistXTrain.shape[1] else: numFeatures = self.cifarXTrain.shape[1] self.currentModel = MLP(complexity, numFeatures, **sampledParams)
```

and:

```
class MLP(nn.Module): def __init__(self, complexity, numFeatures, learningRate, optimizer, dropoutRate, batchSize): super().__init__() layers = [] # build hidden layers if complexity == "low": layers += [nn.Linear(numFeatures, 128), nn.ReLU()] beforeHiddenLayerLength = 128 elif complexity == "medium": layers += [nn.Linear(numFeatures, 512), nn.ReLU(), nn.Linear(512, 256), nn.ReLU(), nn.Linear(256, 128), nn.ReLU()] beforeHiddenLayerLength = 128 elif complexity == "high": layers += [nn.Linear(numFeatures, 1024), nn.ReLU(), nn.Linear(1024, 512), nn.ReLU(), nn.Linear(512, 256), nn.ReLU(), nn.Linear(256, 128), nn.ReLU(), nn.Linear(128, 64), nn.ReLU(), nn.Linear(64, 32), nn.ReLU()] beforeHiddenLayerLength = 32 # dropout + final layer layers.append(nn.Dropout(dropoutRate)) layers.append(nn.Linear(beforeHiddenLayerLength, 10)) # 10 classes
```

```
self.model = nn.Sequential(*layers) # optimizer setup self.criterion =  
nn.CrossEntropyLoss() self.num_epochs = 50 if optimizer == "adam":  
self.optimizer = optim.Adam(self.parameters(), lr=learningRate) else:  
    self.optimizer = optim.SGD(self.parameters(), lr=learningRate)
```

---

## ✓ Summary

- ✓ `**sampledParams` unpacking works fine
  - ✓ You just need to include `dataset` (or `numFeatures`) explicitly
  - ✓ Fix the `append()` typo
  - ✓ Don't access training data directly inside your model class
- 

Would you like me to show you how to train and evaluate this MLP on MNIST  
end-to-end using your parameter sampler?

You said:  
wiill this work?

```
from model import Model  
import torch  
import torch.nn as nn  
import torch.optim as optim  
from torch.utils.data import DataLoader, TensorDataset  
  
class MLP_Manager(Model):  
    def doCustomProcessing(self):  
        # a mlp just expects a 1 dimensional array as its input, so we must flatten our 2D  
        # array to a 1D one.  
        self.mnistXTrain = self.mnistXTrain.reshape(self.mnistXTrain.shape[0],  
                                                    self.mnistXTrain.shape[1] * self.mnistXTrain.shape[2])  
        self.mnistXTest = self.mnistXTest.reshape(self.mnistXTest.shape[0],  
                                                self.mnistXTest.shape[1] * self.mnistXTest.shape[2])
```

```

# may not make a difference, but just in case we transpose before flattening. Do
    this because for the cnn, we have to do this as
        # it expects channel first
        self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)
        self.cifarXTest = self.cifarXTest.transpose(0, 3, 1, 2)

# a mlp just expects a 1 dimensional array as its input, so we must flatten our 3D
    array to a 1D one.
    self.cifarXTrain = self.cifarXTrain.reshape(self.cifarXTrain.shape[0],
                                                self.cifarXTrain.shape[1] * self.cifarXTrain.shape[2] *
                                                self.cifarXTrain.shape[3])

    self.cifarXTest = self.cifarXTest.reshape(self.cifarXTest.shape[0],
                                              self.cifarXTest.shape[1] * self.cifarXTest.shape[2] *
                                              self.cifarXTest.shape[3])

# do the common pre processing before doing what's special to the MLP
    def __init__(self):
        super().__init__()
        # now, custom stuff done
        self.doCustomProcessing()

def validateModelParams(self, complexity, dataset):
    numValuesUsed = 16
    for _ in range(0, numValuesUsed, 1):
        sampledParams = self.sampleParams()
    # first dimension is # of points, so the one after is the number of features
        if(dataset == "mnist"):
            numFeatures = self.mnistXTrain.shape[1]
        else:
            numFeatures = self.cifarXTrain.shape[1]

    self.currentModel = MLP(complexity, dataset, numFeatures, **sampledParams)

```

```

        class MLP(nn.Module):
def __init__(self, complexity, dataset, numFeatures, learningRate, optimizer,
            dropoutRate, batchSize):
            super().__init__()
            layers = []

# do all the layers except for the final layer
beforeHiddenLayerLength = -1
if(complexity == "low"):
    numPerceptrons = 128
layers.append(nn.Linear(numFeatures, numPerceptrons))
    layers.append(nn.ReLU())
beforeHiddenLayerLength = numPerceptrons
elif(complexity == "medium"):
    numPerceptronsFirst = 512
    numPerceptronsSecond = 256
    numPerceptronsThird = 128

layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
    layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
    layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
    layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsThird
elif(complexity == "high"):
    numPerceptronsFirst = 1024
    numPerceptronsSecond = 512
    numPerceptronsThird = 256
    numPerceptronsFourth = 128
    numPerceptronsFifth = 64
    numPerceptronsSixth = 32

```

```

        layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
                layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
                layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
                layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsThird, numPerceptronsFourth))
                layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFourth, numPerceptronsFifth))
                layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFifth, numPerceptronsSixth))
                layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsSixth

        # first, remove one hidden layer
        layers.append(nn.Dropout(dropoutRate))

        # now do the final layer
        numClasses = 10
layers.append(nn.Linear(beforeHiddenLayerLength, numClasses))

        self.model = nn.Sequential(*layers)

        self.criterion = nn.CrossEntropyLoss()
        self.num_epochs = 50

        if(optimizer == "adam"):
            self.optimizer = optim.Adam(self.parameters(), lr = learningRate)
        else:
            self.optimizer = optim.SGD(self.parameters(), lr = learningRate)

from abc import ABC, abstractmethod
from tensorflow.keras.datasets import mnist, cifar10

```

```

class Model(ABC):
    @abstractmethod
    def doCustomProcessing(self):
        pass
    def retrieveDataset(self, setType):
        if setType == "mnist":
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
        else:
            (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

        # normalizing them. Converting pixel values between 0 - 255 to 0 - 1
        xTrain /= 255.0
        xTest /= 255.0

        return (xTrain, yTrain, xTest, yTest)

    # initialize method
    def __init__(self):
        (self.mnistXTrain, self.mnistYTrain, self.mnistXTest, self.mnistYTest) =
            self.retrieveDataset("mnist")
        (self.cifarXTrain, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
            self.retrieveDataset("cifar")

        self.parameters = {
            "learningRate": [0.01, 0.001, 0.0001],
            "batchSize": [32, 64, 128],
            "optimizer": ["sgd", "adam"],
            "dropoutRate": [0.2, 0.5]
        }

    def sampleParams(self):
        return {key: random.choice(values) for key, values in self.parameters.items()}


```

**ChatGPT said:**

You're very close — this is overall solid design  and has good modularity between the base `Model` class, the `MLP_Manager`, and the PyTorch `MLP`.

There are just a few important corrections and best-practice notes to make it actually run and train properly.

---

## Main Notes & Fixes

### 1. Missing `import random`

You call `random.choice` in `sampleParams()` but never import it:

```
import random
```

---

### 2. The order of initialization in `MLP_Manager`

When you call:

```
super().__init__() self.doCustomProcessing()
```

that's correct — since you want to load datasets from the parent class first, then reshape them.

 Keep this.

---

### 3. Incorrect use of `**sampledParams`

This line:

```
self.currentModel = MLP(complexity, dataset, numFeatures, **sampledParams)
```

is fine as long as your sampled dictionary exactly matches the `MLP` constructor parameters:

```
learningRate, optimizer, dropoutRate, batchSize
```

and you've confirmed that's the case in `self.parameters`.  Good.

---

#### 4. You need a `forward()` method in your MLP

Your MLP is a subclass of `nn.Module`, but you haven't defined a `forward()` method — so it won't run during training.

Add this:

```
def forward(self, x): return self.model(x)
```

Without that, PyTorch won't know how to run data through your network.

---

#### 5. Optimizer definition

You currently do this:

```
if(optimizer == "adam"): self.optimizer = optim.Adam(self.parameters(), lr = learningRate) else: self.optimizer = optim.SGD(self.parameters(), lr = learningRate)
```

 That's correct — but note that `"sgd"` is lowercase in your parameter list, so you should check for `"sgd"` explicitly or `.lower()` the string.

---

#### 6. Dataset reshape logic

Everything you did here is right for MLPs (flatten inputs).

For CIFAR, transposing to channel-first before flattening doesn't really matter for MLPs since you're destroying spatial structure anyway — but it's fine and consistent with CNN preprocessing.

 No changes needed.

---

#### 7. Batch normalization (optional improvement)

You could add this easily to your `MLP` if you want to improve stability:

```
layers.append(nn.BatchNorm1d(numPerceptrons))
```

after each `Linear(...)` layer before the ReLU.

---

## 8. Training loop missing

Right now your `MLP` defines layers and optimizers but doesn't actually train.

You can add a simple method inside `MLP`:

```
def train_model(self, train_loader): self.train() for epoch in range(self.num_epochs): for inputs, labels in train_loader: outputs = self.forward(inputs) loss = self.criterion(outputs, labels) self.optimizer.zero_grad() loss.backward() self.optimizer.step()
```

---

## ✓ Final Example Fix

Here's the cleaned-up version of your `MLP` class:

```
class MLP(nn.Module): def __init__(self, complexity, dataset, numFeatures, learningRate, optimizer, dropoutRate, batchSize): super().__init__() layers = [] # hidden layers if complexity == "low": numPerceptrons = 128 layers += [nn.Linear(numFeatures, numPerceptrons), nn.ReLU()] beforeHiddenLayerLength = numPerceptrons elif complexity == "medium": sizes = [512, 256, 128] in_size = numFeatures for size in sizes: layers += [nn.Linear(in_size, size), nn.ReLU()] in_size = size beforeHiddenLayerLength = sizes[-1] else: # high complexity sizes = [1024, 512, 256, 128, 64, 32] in_size = numFeatures for size in sizes: layers += [nn.Linear(in_size, size), nn.ReLU()] in_size = size beforeHiddenLayerLength = sizes[-1] layers.append(nn.Dropout(dropoutRate)) layers.append(nn.Linear(beforeHiddenLayerLength, 10)) # 10 classes self.model = nn.Sequential(*layers) self.criterion = nn.CrossEntropyLoss() self.num_epochs = 50 # optimizer setup optimizer = optimizer.lower() if optimizer == "adam": self.optimizer = optim.Adam(self.parameters(), lr=learningRate) else: self.optimizer = optim.SGD(self.parameters(), lr=learningRate) def forward(self, x): return self.model(x)
```

Would you like me to show you how to connect this with your dataset (i.e. create a `DataLoader`, train it for MNIST or CIFAR, and print accuracy)?

You said:

for testing, how would i then get the accuracy?

```
def trainAndEvaluate(self, trainSet, testSet):
    for epoch in range(self.num_epochs):
        for batchX, batchY in trainSet:
            # zero out gradients
            self.optimizer.zero_grad()

            rawOutputs = self(batchX)

            loss = self.criterion(rawOutputs, batchY)

            # do backward step
            loss.backward()

            self.optimizer.step()

            numCorrect = 0
            numTotal = 0
            with torch.no_grad():
                for batchX, batchY in testSet:
                    output = self(batchX)
                    output = torch.flatten(output)
                    probabilities = torch.softmax(output, dim=0)
                    predictedProb, predictedClass = torch.max(probabilities, 0)
                    predictedClass = predictedClass.item() # now a Python scalar
                    predictedProb = predictedProb.item() # also a Python scalar
```

ChatGPT said:

You're very close — your training loop is solid  — but the accuracy calculation section needs a few small but crucial fixes.

Here's a breakdown:

---

## ✗ Issues in Your Current Code

### 1. Flattening the entire batch output

#### 2. `output = torch.flatten(output)`

This destroys the batch structure — you'll end up mixing all class logits from all samples into one long vector.

→ You should not flatten.

Each batch output is shaped `(batch_size, num_classes)`, so you should pick the argmax across classes (`dim=1`), not across the whole thing.

### 3. Softmax is unnecessary before `argmax`

Since `torch.max` over logits gives the same predicted class as over probabilities, you don't need to apply softmax — it just wastes computation.

### 4. Your loop assumes single-sample batches

Right now you're treating each `batchX` as one image, but `DataLoader` gives batches (tensors of shape `[batch_size, ...]`).

---

## ✓ Corrected `trainAndEvaluate()` method

Here's the fixed version of your code:

```
def trainAndEvaluate(self, trainSet, testSet): for epoch in
range(self.num_epochs): self.train() # sets model to training mode
(important for dropout) for batchX, batchY in trainSet:
    self.optimizer.zero_grad() outputs = self(batchX) loss =
self.criterion(outputs, batchY) loss.backward() self.optimizer.step() #
Evaluate after each epoch (optional) self.eval() numCorrect = 0 numTotal =
0 with torch.no_grad(): for batchX, batchY in testSet: outputs =
self(batchX) # shape: (batch_size, num_classes) predicted =
torch.argmax(outputs, dim=1) # class with highest logit numCorrect +=
(predicted == batchY).sum().item() numTotal += batchY.size(0) accuracy =
```

```
    numCorrect / numTotal print(f"Epoch [{epoch+1}/{self.num_epochs}],  
                                Accuracy: {accuracy:.4f}")
```

---

## Notes

- `self.train()` and `self.eval()` toggle behaviors like dropout and batch normalization — always call them.
  - `torch.argmax(outputs, dim=1)` chooses the best class per sample.
  - You don't need softmax since CrossEntropyLoss already expects raw logits and internally applies it.
- 

Would you like me to show how to build the `trainSet` and `testSet` DataLoaders for your MNIST/CIFAR reshaped tensors too (so you can actually run this end-to-end)?

You said:

if i wanted to do it this way, how would i fix it? cause i have a feeling its incorrect

```
def trainAndEvaluate(self, trainSet, testSet):  
    for epoch in range(self.num_epochs):  
        for batchX, batchY in trainSet:  
            # zero out gradients  
            self.optimizer.zero_grad()  
  
            rawOutputs = self(batchX)  
  
            loss = self.criterion(rawOutputs, batchY)  
  
            # do backward step  
            loss.backward()  
  
            self.optimizer.step()  
  
            numCorrect = 0  
            numTotal = 0
```

```
with torch.no_grad():
    for batchX, batchY in testSet:
        output = self(batchX)
        predictedOutputs = torch.argmax(output, dim = 1)
```

```
for i in range(0, len(predictedOutputs), 1):
    if(predictedOutputs[i] == batchY[i]):
        numCorrect += 1
        numTotal += 1
```

ChatGPT said:

Good — that's actually pretty close! 

You're right to suspect it's *almost* correct but slightly inefficient / subtly off. Let's go over exactly what's happening.

---

## Your code right now

```
with torch.no_grad(): for batchX, batchY in testSet: output = self(batchX)
    predictedOutputs = torch.argmax(output, dim=1) for i in range(0,
len(predictedOutputs), 1): if(predictedOutputs[i] == batchY[i]):
    numCorrect += 1 numTotal += 1
```

---

## What's good

- You're correctly calling `torch.argmax(..., dim=1)` to get the predicted class for each sample.
  - You're iterating over the test set in batches.
  - You're correctly using `torch.no_grad()` to disable gradient tracking during evaluation.
- 

## What's slightly wrong or inefficient

1. Manual for-loop over batch elements is unnecessary — PyTorch tensors support vectorized comparison.
2. `numTotal += 1` should only increase by the batch size, not 1 per element.

3. No `.eval()` mode set — dropout/batchnorm layers (if added later) will behave incorrectly in evaluation.
  4. It's fine logically, but slower for large datasets.
- 

## ✓ Cleaned-up, equivalent version (your same logic, just vectorized)

This version behaves identically but is faster and correct in all edge cases:

```
def trainAndEvaluate(self, trainSet, testSet): for epoch in range(self.num_epochs): self.train() for batchX, batchY in trainSet: self.optimizer.zero_grad() rawOutputs = self(batchX) loss = self.criterion(rawOutputs, batchY) loss.backward() self.optimizer.step() # Evaluation self.eval() numCorrect = 0 numTotal = 0 with torch.no_grad(): for batchX, batchY in testSet: outputs = self(batchX) predicted = torch.argmax(outputs, dim=1) numCorrect += (predicted == batchY).sum().item() numTotal += batchY.size(0) accuracy = numCorrect / numTotal print(f"Accuracy: {accuracy:.4f}") return accuracy
```

---

## 💡 Why this works

- `(predicted == batchY)` gives a Boolean tensor of shape `(batch_size,)`.
  - `.sum()` counts how many were correct in that batch.
  - `.item()` converts the tensor to a Python number.
  - `batchY.size(0)` is the number of samples in the batch.
- 

## ✓ TL;DR fix summary

If you really want to *keep your loop style*, minimally change your code like this:

```
for i in range(len(predictedOutputs)): if predictedOutputs[i].item() == batchY[i].item(): numCorrect += 1 numTotal += len(predictedOutputs)
```

That fixes the logic, but the vectorized version above is the recommended, clean

PyTorch way.

---

Would you like me to show how to integrate this method into your existing `MLP` class cleanly (so it can run after `validateModelParams()`)?

You said:

is there like a stopwatch thing or something similar in python?

ChatGPT said:

Yes — there are a few great ways to measure elapsed time in Python depending on what you're doing 🚀

Here are the most common “stopwatch” options, from simplest to most precise:

---



### 1. Using `time.time()` (simple wall-clock timing)

```
import time start = time.time() # your code here time.sleep(2) # simulate
something that takes time end = time.time() print(f"Elapsed time: {end -
start:.4f} seconds")
```

✓ Easy to use

⚠️ Can be affected by system clock changes (e.g., daylight savings, manual clock adjustments)

---



### 2. Using `time.perf_counter()` (best for precise timing)

```
import time start = time.perf_counter() # your code here time.sleep(2) end
= time.perf_counter() print(f"Elapsed time: {end - start:.4f} seconds")
```

✓ High-resolution timer

✓ Immune to clock changes



Recommended for benchmarking code or model training

---

⌚ 3. Using `timeit.default_timer()` (same as `perf_counter()` under the hood)

```
from timeit import default_timer as timer start = timer() # your code here  
end = timer() print(f"Elapsed time: {end - start:.4f} seconds")
```

✓ Common in performance testing scripts

✓ Slightly more portable across platforms

---



## 4. Using `datetime` (for human-readable timestamps)

```
from datetime import datetime start = datetime.now() # your code here end  
= datetime.now() print(f"Elapsed time: {end - start}")
```

✓ Great for logs, debugging, or timestamped printouts

⚠ Not ideal for precise benchmarking

---



### Example — timing your training

Here's how you could integrate it in your training loop:

```
import time def trainAndEvaluate(self, trainSet, testSet): start =  
time.perf_counter() for epoch in range(self.num_epochs): self.train() for  
batchX, batchY in trainSet: self.optimizer.zero_grad() outputs =  
self(batchX) loss = self.criterion(outputs, batchY) loss.backward()  
self.optimizer.step() end = time.perf_counter() print(f"Training took {end  
- start:.2f} seconds")
```

Would you like me to show how to make it display epoch-by-epoch timing or total +

average per epoch?

You said:

can i convert it to minutes/seconds?

ChatGPT said:

Absolutely ✓ — you can easily convert elapsed time from seconds to minutes and seconds (or hours if you like).

Here are a few simple ways:

---

## Option 1: Manual division

```
import time start = time.perf_counter() time.sleep(125) # simulate 125
seconds end = time.perf_counter() elapsed = end - start minutes =
int(elapsed // 60) seconds = elapsed % 60 print(f"Elapsed time: {minutes}
min {seconds:.2f} sec")
```

Output:

```
Elapsed time: 2 min 5.00 sec
```

---

## Option 2: Use `divmod()` for cleaner code

```
elapsed = end - start minutes, seconds = divmod(elapsed, 60)
print(f"Elapsed time: {int(minutes)} min {seconds:.2f} sec")
```

---

## Option 3: If you want hours too

```
hours, remainder = divmod(elapsed, 3600) minutes, seconds =
divmod(remainder, 60) print(f"Elapsed time: {int(hours)}h {int(minutes)}m
{seconds:.2f}s")
```

---

## Example in your training loop:

```
import time def trainAndEvaluate(self, trainSet, testSet): start =
time.perf_counter() for epoch in range(self.num_epochs): self.train() for
batchX, batchY in trainSet: self.optimizer.zero_grad() outputs =
self(batchX) loss = self.criterion(outputs, batchY) loss.backward()
self.optimizer.step() end = time.perf_counter() elapsed = end - start
minutes, seconds = divmod(elapsed, 60) print(f"Training took
{int(minutes)} min {seconds:.2f} sec")
```

---

Would you like me to make it print timing after each epoch too (like “Epoch 3 finished in 1m 12s”)?

You said:  
will this code run?

```
from mlp import MLP_Manager

complexities = ["low", "medium", "high"]

datasetTypes = ["mnist", "cifar"]

for dataset in datasetTypes:
    for complexityLevel in complexities:
        newModel = MLP_Manager()
        newModel.createFullModel(complexityLevel, dataset)

from abc import ABC, abstractmethod
from tensorflow.keras.datasets import mnist, cifar10
import random
import time

class Model(ABC):
    @abstractmethod
    def doCustomProcessing(self):
        pass
    def retrieveDataset(self, setType):
        if setType == "mnist":
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
        else:
            (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

        # normalizing them. Converting pixel values between 0 - 255 to 0 - 1
        xTrain /= 255.0
```

```

        xTest /= 255.0

    return (xTrain, yTrain, xTest, yTest)

    # initialize method
    def __init__(self):
        (self.mnistXTrain, self.mnistYTrain, self.mnistXTest, self.mnistYTest) =
            self.retrieveDataset("mnist")
        (self.cifarXTrain, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
            self.retrieveDataset("cifar")

        self.parameters = {
            "learningRate": [0.01, 0.001, 0.0001],
            "batchSize": [32, 64, 128],
            "optimizer": ["sgd", "adam"],
            "dropoutRate": [0.2, 0.5]
        }

    def sampleParams(self):
        return {key: random.choice(values) for key, values in self.parameters.items()}

    def createTensorsAndSplitting(self):
        mnistTrain = TensorDataset(self.mnistXTrain, self.mnistYTrain)
        cifarTrain = TensorDataset(self.cifarXTrain, self.cifarYTrain)

        self.mnistTrain, self.mnistValid = random_split(mnistTrain, [50000, 10000])

        self.cifarTrain, self.cifarValid = random_split(cifarTrain, [45000, 5000])

    @abstractmethod
    def doModelCreation(self, complexity, numFeatures, sampledParams):
        pass

    def createBatches(self, batchSize, dataset):

```

```

        if(dataset == "mnist"):

self.trainBatch = Dataloader(self.mnistTrain, batch_size = batchSize, shuffle =
True)

self.testBatch = Dataloader(self.mnistValid, batch_size = batchSize)
else:

self.trainBatch = Dataloader(self.cifarTrain, batch_size = batchSize, shuffle =
True)

self.testBatch = Dataloader(self.cifarValid, batch_size = batchSize)

def createFullModel(self, complexity, dataset):

    startTime = time.time()
    numValuesUsed = 16
    self.finalParams = {}
    highestAccuracy = 0

    for _ in range(0, numValuesUsed, 1):
        sampledParams = self.sampleParams()

# first dimension is # of points, so the one after is the number of features

        if(dataset == "mnist"):

            numFeatures = self.mnistXTrain.shape[1]
        else:

            numFeatures = self.cifarXTrain.shape[1]

        self.doModelCreation(complexity, numFeatures, sampledParams)

        self.createBatches(sampledParams["batchSize"])

accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch)

        if(accuracy > highestAccuracy):

            highestAccuracy = accuracy
            self.finalParams = sampledParams

self.doModelCreation(complexity, numFeatures, self.finalParams)

self.createBatches(self.finalParams["batchSize"])

```

```

accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch)

endTime = time.time()

numMinutes = (endTime - startTime) / 60
numSeconds = (endTime - startTime) % 60

print(f"Accuracy for complexity of {complexity} and dataset of {dataset} was
{accuracy}, while total elapsed time was {numMinutes} minutes and {numSeconds}
seconds. Additionally, the parameters chosen were: {self.finalParams}")

```

```

from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

```

```

class MLP_Manager(Model):
    def doCustomProcessing(self):
        # a mlp just expects a 1 dimensional array as its input, so we must flatten our 2D
        # array to a 1D one.
        self.mnistXTrain = self.mnistXTrain.reshape(self.mnistXTrain.shape[0],
                                                    self.mnistXTrain.shape[1] * self.mnistXTrain.shape[2])
        self.mnistXTest = self.mnistXTest.reshape(self.mnistXTest.shape[0],
                                                self.mnistXTest.shape[1] * self.mnistXTest.shape[2])

        # may not make a difference, but just in case we transpose before flattening. Do
        # this because for the cnn, we have to do this as
        # it expects channel first
        self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)
        self.cifarXTest = self.cifarXTest.transpose(0, 3, 1, 2)

        # a mlp just expects a 1 dimensional array as its input, so we must flatten our 3D
        # array to a 1D one.

```

```

self.cifarXTrain = self.cifarXTrain.reshape(self.cifarXTrain.shape[0],
                                             self.cifarXTrain.shape[1] * self.cifarXTrain.shape[2] *
                                             self.cifarXTrain.shape[3])

self.cifarXTest = self.cifarXTest.reshape(self.cifarXTest.shape[0],
                                         self.cifarXTest.shape[1] * self.cifarXTest.shape[2] *
                                         self.cifarXTest.shape[3])

# do the common pre processing before doing what's special to the MLP
    def __init__(self):
        super().__init__()
        # now, custom stuff done
        self.doCustomProcessing()

        self.createTensorsAndSplitting()

def doModelCreation(self, complexity, numFeatures, sampledParams):
    self.currentModel = MLP(complexity, numFeatures, **sampledParams)

class MLP(nn.Module):
    def __init__(self, complexity, numFeatures, learningRate, optimizer, dropoutRate,
                 batchSize):
        super().__init__()
        layers = []

        # do all the layers except for the final layer
        beforeHiddenLayerLength = -1
        if(complexity == "low"):
            numPerceptrons = 128
        layers.append(nn.Linear(numFeatures, numPerceptrons))
        layers.append(nn.ReLU())
        beforeHiddenLayerLength = numPerceptrons
        elif(complexity == "medium"):
```

```

        numPerceptronsFirst = 512
        numPerceptronsSecond = 256
        numPerceptronsThird = 128

    layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
        layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsThird
    elif(complexity == "high"):
        numPerceptronsFirst = 1024
        numPerceptronsSecond = 512
        numPerceptronsThird = 256
        numPerceptronsFourth = 128
        numPerceptronsFifth = 64
        numPerceptronsSixth = 32

    layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsThird, numPerceptronsFourth))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsFourth, numPerceptronsFifth))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsFifth, numPerceptronsSixth))
        layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsSixth

```

```

        # first, remove on hidden layers
        layers.append(nn.Dropout(dropoutRate))

        # now do the final layer
        numClasses = 10
    layers.append(nn.Linear(beforeHiddenLayerLength, numClasses))

    self.model = nn.Sequential(*layers)

    self.criterion = nn.CrossEntropyLoss()
    self.num_epochs = 50

    if(optimizer == "adam"):
        self.optimizer = optim.Adam(self.parameters(), lr = learningRate)
    else:
        self.optimizer = optim.SGD(self.parameters(), lr = learningRate)

    def trainAndEvaluate(self, trainSet, testSet):
        for epoch in range(self.num_epochs):
            for batchX, batchY in trainSet:
                # zero out gradients
                self.optimizer.zero_grad()

                rawOutputs = self(batchX)

                loss = self.criterion(rawOutputs, batchY)

                # do backward step
                loss.backward()

                self.optimizer.step()

                numCorrect = 0
                numTotal = 0
                with torch.no_grad():

```

```
        for batchX, batchY in testSet:  
            output = self(batchX)  
            predictedOutputs = torch.argmax(output, dim = 1)  
  
            for i in range(0, len(predictedOutputs), 1):  
                if(predictedOutputs[i] == batchY[i]):  
                    numCorrect += 1  
                numTotal += 1  
  
        return numCorrect / numTotal
```

ChatGPT said:

That's a *very* solid and well-structured script — you've clearly put a lot of thought into the architecture. 

However, as written, the code will not run successfully yet. It has several critical issues (some minor, some structural). Let's go through the main ones step-by-step.

---



## Major issues that will break execution

### 1. `TensorDataset` and `DataLoader` are used before being imported

You import them in `MLP_Manager`, but not in the parent `Model` class file — where they are first referenced:

```
mnistTrain = TensorDataset(self.mnistXTrain, self.mnistYTrain)
```

 Fix:

Move these imports to the top of the `Model` file:

```
from torch.utils.data import TensorDataset, DataLoader, random_split
```

---

### 2. Incorrect capitalization of `DataLoader`

You wrote `Dataloader` instead of `DataLoader` in a few places:

```
self.trainBatch = Dataloader(self.mnistTrain, batchSize,  
                             shuffle = True)
```

✓ Fix:

```
self.trainBatch = DataLoader(self.mnistTrain, batchSize=batchSize,  
                            shuffle=True)
```

---

### 3. Tensor conversion needed before creating `TensorDataset`

The MNIST and CIFAR data you load from Keras are NumPy arrays, but

`TensorDataset` expects PyTorch tensors.

✓ Fix: Add this conversion in your `retrieveDataset` or after normalization:

```
import torch xTrain = torch.tensor(xTrain, dtype=torch.float32) xTest =  
torch.tensor(xTest, dtype=torch.float32) yTrain = torch.tensor(yTrain,  
               dtype=torch.long).squeeze() yTest = torch.tensor(yTest,  
               dtype=torch.long).squeeze()
```

---

### 4. Transposing CIFAR arrays

You're calling:

```
self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)
```

But PyTorch tensors use the `.permute()` method, not `.transpose()` with multiple axes.

✓ Fix:

```
self.cifarXTrain = self.cifarXTrain.permute(0, 3, 1, 2) self.cifarXTest =  
self.cifarXTest.permute(0, 3, 1, 2)
```

---

### 5. `trainAndEvaluate` calls `self(batchX)` — but `MLP` doesn't define a `forward()` method

In PyTorch, `self(batchX)` implicitly calls `.forward(batchX)`.

You need to define that method.

✓ Fix:

```
def forward(self, x): return self.model(x)
```

---

## 6. Shape mismatch likely for CIFAR input

Flattening CIFAR after transposing to `(N, 3, 32, 32)` gives a `numFeatures` of  $3 * 32$

$* 32 = 3072$ , but your code gets:

```
numFeatures = self.cifarXTrain.shape[1]
```

That only returns 3 (the channel dimension).

✓ Fix: compute correctly:

```
numFeatures = self.cifarXTrain.shape[1] * self.cifarXTrain.shape[2] *  
            self.cifarXTrain.shape[3]
```

---

## 7. `random_split` import missing

Same issue — you must import it explicitly:

```
from torch.utils.data import random_split
```

---

## 8. Minor: wrong argument count in `createBatches()` call

In `createFullModel():`

```
self.createBatches(sampledParams["batchSize"])
```

But your method definition expects two parameters (`batchSize, dataset`).

✓ Fix:

```
self.createBatches(sampledParams["batchSize"], dataset)
```

 After these fixes, your script will likely run properly

---

 Additional suggestions (for performance + clarity)

1. Normalize CIFAR correctly — CIFAR has pixel values 0–255, so dividing by 255 is good, but it's often better to use mean/std normalization for better convergence.

2. Use GPU if available:

3. 

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu") self.model.to(device)
```

4. Batch training loop optimization:

Move batches to device and cast properly:

5. 

```
batchX, batchY = batchX.to(device), batchY.to(device)
```

6. Use tqdm for progress if you want to track epoch progress cleanly.

---

Would you like me to rewrite a fixed, fully runnable version of this code (with all corrections and GPU support included)?

You said:  
will this code work correct?

```
from mlp import MLP_Manager  
from cnn import CNN_Manager
```

```
complexities = ["low", "medium", "high"]
```

```
datasetTypes = ["mnist", "cifar"]
```

```
mlpModel = MLP_Manager()
```

```
cnnModel = CNN_Manager()
```

```
for dataset in datasetTypes:  
    for complexityLevel in complexities:
```

```

mlpModel.createFullModel(complexityLevel,dataset)

        for dataset in datasetTypes:
            for complexityLevel in complexities:
                cnnModel.createFullModel(complexityLevel,dataset)

from abc import ABC, abstractmethod
from tensorflow.keras.datasets import mnist, cifar10
import random
import time
from torch.utils.data import TensorDataset, DataLoader, random_split
import torch

class Model(ABC):
    @abstractmethod
    def doCustomProcessing(self):
        pass
    def retrieveDataset(self, setType):
        if setType == "mnist":
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
        else:
            (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

# normalizing them. Converting pixel values between 0 - 255 to 0 - 1
        xTrain /= 255.0
        xTest /= 255.0

        return (xTrain, yTrain, xTest, yTest)

    # initialize method
    def __init__(self):
        (self.mnistXTrain, self.mnistYTrain, self.mnistXTest, self.mnistYTest) =
            self.retrieveDataset("mnist")

```

```

        (self.cifarXTrain, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
            self.retrieveDataset("cifar")

            self.parameters = {
                "learningRate": [0.01, 0.001, 0.0001],
                "batchSize": [32, 64, 128],
                "optimizer": ["sgd", "adam"],
                "dropoutRate": [0.2, 0.5]
            }

        def sampleParams(self):
            return {key: random.choice(values) for key, values in self.parameters.items()}

            def createTensorsAndSplitting(self):
                self.mnistXTrain = torch.tensor(self.mnistXTrain, dtype=torch.float32)
                self.mnistXTest = torch.tensor(self.mnistXTest, dtype=torch.float32)
                self.mnistYTrain = torch.tensor(self.mnistYTrain, dtype=torch.long).squeeze()
                self.mnistYTest = torch.tensor(self.mnistYTest, dtype=torch.long).squeeze()

                self.cifarXTrain = torch.tensor(self.cifarXTrain, dtype=torch.float32)
                self.cifarXTest = torch.tensor(self.cifarXTest, dtype=torch.float32)
                self.cifarYTrain = torch.tensor(self.cifarYTrain, dtype=torch.long).squeeze()
                self.cifarYTest = torch.tensor(self.cifarYTest, dtype=torch.long).squeeze()

mnistTrain = TensorDataset(self.mnistXTrain, self.mnistYTrain)
cifarTrain = TensorDataset(self.cifarXTrain, self.cifarYTrain)

self.mnistTrain, self.mnistValid = random_split(mnistTrain, [50000, 10000])

self.cifarTrain, self.cifarValid = random_split(cifarTrain, [45000, 5000])

    @abstractmethod
def doModelCreation(self, complexity, dataset, sampledParams):
    pass

```

```

def createBatches(self, batchSize, dataset):

    if(dataset == "mnist"):
        self.trainBatch = DataLoader(self.mnistTrain, batch_size = batchSize, shuffle =
            True)
        self.testBatch = DataLoader(self.mnistValid, batch_size = batchSize)
    else:
        self.trainBatch = DataLoader(self.cifarTrain, batch_size = batchSize, shuffle =
            True)
        self.testBatch = DataLoader(self.cifarValid, batch_size = batchSize)

    def createFullModel(self, complexity, dataset):
        startTime = time.time()
        numValuesUsed = 16
        self.finalParams = {}
        highestAccuracy = 0

        for _ in range(0, numValuesUsed, 1):
            sampledParams = self.sampleParams()

            self.doModelCreation(complexity, sampledParams)

            self.createBatches(sampledParams["batchSize"], dataset)

            accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch)

            if(accuracy > highestAccuracy):
                highestAccuracy = accuracy
                self.finalParams = sampledParams

            self.doModelCreation(complexity, numFeatures, self.finalParams)
            self.createBatches(self.finalParams["batchSize"], dataset)

accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch)

```

```

        endTime = time.time()

        numMinutes = (endTime - startTime) / 60
        numSeconds = (endTime - startTime) % 60

    print(f"Accuracy for complexity of {complexity} and dataset of {dataset} was
{accuracy}, while total elapsed time was {numMinutes} minutes and {numSeconds}
seconds. Additionally, the parameters chosen were: {self.finalParams}")

```

```

from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

```

```

class MLP_Manager(Model):
    def doCustomProcessing(self):
# a mlp just expects a 1 dimensional array as its input, so we must flatten our 2D
array to a 1D one.
        self.mnistXTrain = self.mnistXTrain.reshape(self.mnistXTrain.shape[0],
self.mnistXTrain.shape[1] * self.mnistXTrain.shape[2])
        self.mnistXTest = self.mnistXTest.reshape(self.mnistXTest.shape[0],
self.mnistXTest.shape[1] * self.mnistXTest.shape[2])

# may not make a difference, but just in case we transpose before flattening. Do
this because for the cnn, we have to do this as
        # it expects channel first
        self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)
        self.cifarXTest = self.cifarXTest.transpose(0, 3, 1, 2)

# a mlp just expects a 1 dimensional array as its input, so we must flatten our 3D
array to a 1D one.
        self.cifarXTrain = self.cifarXTrain.reshape(self.cifarXTrain.shape[0],

```

```

        self.cifarXTrain.shape[1] * self.cifarXTrain.shape[2] *
        self.cifarXTrain.shape[3])

    self.cifarXTest = self.cifarXTest.reshape(self.cifarXTest.shape[0],
                                              self.cifarXTest.shape[1] * self.cifarXTest.shape[2] *
                                              self.cifarXTest.shape[3])

# do the common pre processing before doing what's special to the MLP
    def __init__(self):
        super().__init__()
        # now, custom stuff done
        self.doCustomProcessing()

        self.createTensorsAndSplitting()

    def doModelCreation(self, complexity, dataset, sampledParams):
        # first dimension is # of points, so the one after is the number of features
        if(dataset == "mnist"):
            numFeatures = self.mnistXTrain.shape[1]
        else:
            numFeatunumFeaturesresOrChannels = self.cifarXTrain.shape[1]

        self.currentModel = MLP(complexity, numFeatures, **sampledParams)

class MLP(nn.Module):
    def __init__(self, complexity, numFeatures, learningRate, optimizer, dropoutRate,
                 batchSize):
        super().__init__()
        layers = []

        # do all the layers except for the final layer
        beforeHiddenLayerLength = -1
        if(complexity == "low"):
```

```

        numPerceptrons = 128
    layers.append(nn.Linear(numFeatures, numPerceptrons))
        layers.append(nn.ReLU())
    beforeHiddenLayerLength = numPerceptrons
    elif(complexity == "medium"):
        numPerceptronsFirst = 512
        numPerceptronsSecond = 256
        numPerceptronsThird = 128

    layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
        layers.append(nn.ReLU())

    beforeHiddenLayerLength = numPerceptronsThird
    elif(complexity == "high"):
        numPerceptronsFirst = 1024
        numPerceptronsSecond = 512
        numPerceptronsThird = 256
        numPerceptronsFourth = 128
        numPerceptronsFifth = 64
        numPerceptronsSixth = 32

    layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsThird, numPerceptronsFourth))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsFourth, numPerceptronsFifth))
        layers.append(nn.ReLU())

```

```

layers.append(nn.Linear(numPerceptronsFifth, numPerceptronsSixth))
    layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsSixth

# first, remove one hidden layer
layers.append(nn.Dropout(dropoutRate))

# now do the final layer
numClasses = 10
layers.append(nn.Linear(beforeHiddenLayerLength, numClasses))

self.model = nn.Sequential(*layers)

self.criterion = nn.CrossEntropyLoss()
self.num_epochs = 50

if(optimizer == "adam"):
    self.optimizer = optim.Adam(self.parameters(), lr = learningRate)
else:
    self.optimizer = optim.SGD(self.parameters(), lr = learningRate)

def forward(self, x):
    return self.model(x)

def trainAndEvaluate(self, trainSet, testSet):
    for epoch in range(self.num_epochs):
        for batchX, batchY in trainSet:
            # zero out gradients
            self.optimizer.zero_grad()

            rawOutputs = self(batchX)

            loss = self.criterion(rawOutputs, batchY)

```

```

        # do backward step
        loss.backward()

        self.optimizer.step()

        numCorrect = 0
        numTotal = 0
        with torch.no_grad():
            for batchX, batchY in testSet:
                output = self(batchX)
                predictedOutputs = torch.argmax(output, dim = 1)

                for i in range(0, len(predictedOutputs), 1):
                    if(predictedOutputs[i] == batchY[i]):
                        numCorrect += 1
                        numTotal += 1

        return numCorrect / numTotal

from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import numpy as np

class CNN_Manager(Model):
    # this method does custom processing needed for the CNN
    def doCustomProcessing(self):
        # the CNN expects a 3D array; the MNIST is 2D as it has no dimension for color
        # (its monochrome), so we
        # squeeze in another dimension
        self.mnistXTrain = np.expand_dims(self.mnistXTrain, axis = 1)
        self.mnistXTest = np.expand_dims(self.mnistXTest, axis = 1)

```

```

# the cnn expects color dimension first, so just swap the dimensions for this
    self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)
    self.cifarXTest = self.cifarXTest.transpose(0, 3, 1, 2)

# initial function that runs the common processing, then does what's special to the
    CNN
        def __init__(self):
            super().__init__()
            # now, custom stuff done
            self.doCustomProcessing()

            self.createTensorsAndSplitting()

    def doModelCreation(self, complexity, dataset, sampledParams):
        # first dimension is # of points, so the one after is the number of features
        if(dataset == "mnist"):
            numChannels = 1
        else:
            numChannels = 3

        self.currentModel = CNN(complexity, numChannels, **sampledParams)

    class CNN(nn.Module):
        def __init__(self, complexity, numChannels, learningRate, optimizer, dropoutRate,
                     batchSize):
            super().__init__()
            layers = []

            finalNumFilters = 0

            if(complexity == "low"):
                numFiltersFirstLayer = 32
                convStepSize = 1
                firstKernelSize = 5

```

```
layers.append(nn.Conv2d(numChannels, numFiltersFirstLayer, firstKernelSize,
                      padding = 1))
    layers.append(nn.ReLU())
```

# dimension of each filter after will be 26 x 26 for MNIST, and 30 x 30 for CIFAR.  
SO, we can use the following pooling fine

```
    poolKernelSize = 2
    poolStepSize = poolKernelSize
    layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))
```

# after this, dimension of each filter will be 13 x 13 for MNIST, and 15 x 15 for  
CIFAR

```
    numFiltersSecondLayer = 64
    secondKernelSize = 3
    layers.append(nn.Conv2d(numFiltersFirstLayer, numFiltersSecondLayer,
                          secondKernelSize, padding = 1))
        layers.append(nn.ReLU())
```

```
    layers.append(nn.AdaptiveAvgPool2d(1))
```

```
    finalNumFilters = numFiltersSecondLayer
    elif(complexity == "medium"):
        numFiltersFirstLayer = 32
        convStepSize = 1
        firstKernelSize = 5
```

```
    layers.append(nn.Conv2d(numChannels, numFiltersFirstLayer, firstKernelSize,
                          padding = 1))
        layers.append(nn.BatchNorm2d(numFiltersFirstLayer))
            layers.append(nn.ReLU())
```

# dimension of each filter after will be 26 x 26 for MNIST, and 30 x 30 for CIFAR.  
SO, we can use the following pooling fine

```
    poolKernelSize = 2
```

```

        poolStepSize = poolKernelSize
        layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))

# after this, dimension of each filter will be 12 x 12 for MNIST, and 14 x 14 for
# CIFAR
        numFiltersSecondLayer = 64
        secondKernelSize = 3
        layers.append(nn.Conv2d(numFiltersFirstLayer, numFiltersSecondLayer,
                             secondKernelSize, padding = 1))
        layers.append(nn.BatchNorm2d(numFiltersSecondLayer))
        layers.append(nn.ReLU())

        layers.append(nn.AdaptiveAvgPool2d(1))

finalNumFilters = numFiltersSecondLayer
elif(complexity == "high"):
    numFiltersFirstLayer = 32
    convStepSize = 1
    firstKernelSize = 3
layers.append(nn.Conv2d(numChannels, numFiltersFirstLayer, firstKernelSize,
                      padding = 1))
layers.append(nn.BatchNorm2d(numFiltersFirstLayer))
layers.append(nn.ReLU())

# dimension of each filter after will be 28 x 28 for MNIST, and 32 x 32 for CIFAR.
# SO, we can use the following pooling fine
```

```

        poolKernelSize = 2
        poolStepSize = poolKernelSize
        layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))
```

```

# after this, dimension of each filter will be 14 x 14 for MNIST, and 16 x 16 for
# CIFAR
        numFiltersSecondLayer = 64
        secondKernelSize = 3
```

```

        layers.append(nn.Conv2d(numFiltersFirstLayer, numFiltersSecondLayer,
                               secondKernelSize, padding = 1))
        layers.append(nn.BatchNorm2d(numFiltersSecondLayer))
        layers.append(nn.ReLU())

        layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))

        numFiltersThirdLayer = 128
        secondKernelSize = 2
        layers.append(nn.Conv2d(numFiltersSecondLayer, numFiltersThirdLayer,
                               secondKernelSize, padding = 1))
        layers.append(nn.BatchNorm2d(numFiltersThirdLayer))
        layers.append(nn.ReLU())

        layers.append(nn.AdaptiveAvgPool2d(1))

        finalNumFilters = numFiltersThirdLayer

        layers.append(nn.flatten())
        layers.append(nn.Linear(finalNumFilters, finalNumFilters * 2))
        layers.append(nn.ReLU())

        # first, remove one hidden layers
        layers.append(nn.Dropout(dropoutRate))

        numClasses = 10
        layers.append(nn.Linear(finalNumFilters * 2, numClasses))

        self.preModel = nn.Sequential(*layers)

        self.criterion = nn.CrossEntropyLoss()
        self.num_epochs = 50

        if(optimizer == "adam"):
            self.optimizer = optim.Adam(self.parameters(), lr = learningRate)

```

```

        else:
    self.optimizer = optim.SGD(self.parameters(), lr = learningRate)

    def forward(self, x):
        return self.model(x)

def trainAndEvaluate(self, trainSet, testSet):
    for epoch in range(self.num_epochs):
        for batchX, batchY in trainSet:
            # zero out gradients
            self.optimizer.zero_grad()

            rawOutputs = self(batchX)

            loss = self.criterion(rawOutputs, batchY)

            # do backward step
            loss.backward()

            self.optimizer.step()

            numCorrect = 0
            numTotal = 0
            with torch.no_grad():
                for batchX, batchY in testSet:
                    output = self(batchX)
                    predictedOutputs = torch.argmax(output, dim = 1)

                    for i in range(0, len(predictedOutputs), 1):
                        if(predictedOutputs[i] == batchY[i]):
                            numCorrect += 1
                            numTotal += 1

    return numCorrect / numTotal

```

ChatGPT said:

That's a great start, but no — the code won't run correctly yet. There are several important issues scattered across the files that will cause runtime errors or logical bugs.

Let's go through them carefully and fix each one.

---

## 1. Top-level loop (`main` script)

Problem:

You create one `MLP_Manager()` and one `CNN_Manager()`, and reuse them for all dataset/complexity combinations.

But each `createFullModel` call mutates the manager's internal dataset tensors (train/test splits, etc.).

So later runs will use *incorrect or mismatched* data shapes (especially between MNIST and CIFAR).

Fix:

Create new manager instances inside the loops:

```
for dataset in datasetTypes: for complexityLevel in complexities: mlpModel = MLP_Manager() mlpModel.createFullModel(complexityLevel, dataset) for dataset in datasetTypes: for complexityLevel in complexities: cnnModel = CNN_Manager() cnnModel.createFullModel(complexityLevel, dataset)
```

---

## 2. `Model.createFullModel` – missing arguments and incorrect signature calls

Problem:

In your loop:

```
self.doModelCreation(complexity, sampledParams)
```

but your abstract definition is:

```
def doModelCreation(self, complexity, dataset, sampledParams):
```

So you're missing the `dataset` parameter, and later you also call:

```
self.doModelCreation(complexity, numFeatures, self.finalParams)
```

which passes `numFeatures` instead of `dataset`.

Fix:

Make the calls consistent:

```
self.doModelCreation(complexity, dataset, sampledParams) ...  
self.doModelCreation(complexity, dataset, self.finalParams)
```



### 3. `MLP_Manager.doModelCreation` – variable typo

You wrote:

```
numFeatunumFeaturesresOrChannels = self.cifarXTrain.shape[1]
```

That's a malformed variable name.

Should be:

```
numFeatures = self.cifarXTrain.shape[1]
```



### 4. `CNN` – incorrect variable name for the main model

You define:

```
self.preModel = nn.Sequential(*layers)
```

but the forward method uses:

```
return self.model(x)
```

→ This will throw `AttributeError: 'CNN' object has no attribute 'model'`.

Fix:

Rename `self.preModel` → `self.model`:

```
self.model = nn.Sequential(*layers)
```

---

## 5. `Model.createFullModel` – **undefined** `numFeatures`

You reference `numFeatures` here:

```
self.doModelCreation(complexity, numFeatures, self.finalParams)
```

but `numFeatures` doesn't exist in this scope.

This should just be:

```
self.doModelCreation(complexity, dataset, self.finalParams)
```

---

## 6. CNN training – missing batch dimension adjustment

MNIST inputs are shaped `[batch, 1, 28, 28]` and CIFAR `[batch, 3, 32, 32]`.

Your preprocessing is correct for that — but make sure

`self.createTensorsAndSplitting()` runs after `doCustomProcessing()`.

(You already do this  — good.)

---

## 7. Typo: `Dataloader` vs `DataLoader`

You use:

```
self.trainBatch = Dataloader(...)
```

in one part of the old version — make sure everywhere it's `DataLoader` (capital L).

---

## 8. Optional: Improve time formatting

Instead of printing decimal minutes and seconds:

```
numMinutes = int((endTime - startTime) // 60) numSeconds = int((endTime -  
startTime) % 60) print(f"Elapsed time: {numMinutes}m {numSeconds}s")
```

---

## Summary of critical fixes

Problem	Fix
Reuse of model instances	Recreate <code>MLP_Manager</code> and <code>CNN_Manager</code> inside loops
Missing <code>dataset</code> argument	Pass it to <code>doModelCreation</code> correctly
Typo in <code>numFeatunumFeaturesresOrChannels</code>	Change to <code>numFeatures</code>
Wrong model variable in CNN	Use <code>self.model</code> not <code>self.preModel</code>
Undefined <code>numFeatures</code> in <code>createFullModel</code>	Replace with <code>dataset</code>
Possible typo <code>Dataloader</code>	Change to <code>DataLoader</code>

Would you like me to show the fully corrected version of all three files (main script, `model.py`, `mlp.py`, and `cnn.py`) in one cohesive block so you can just copy-paste and run it?

You said:

with the first point you mentioned, is there really an issue there?

ChatGPT said:

Excellent question — and you're right to challenge that first point 🙌

Let's dig into that carefully, because whether it's actually an issue depends on how your `Model` (and its subclasses like `MLP_Manager` and `CNN_Manager`) handle state internally.

## ⚙️ What happens right now

You have something like:

```
mlpModel = MLP_Manager()
for dataset in datasetTypes:
    for complexityLevel
        in complexities:
            mlpModel.createFullModel(complexityLevel, dataset)
```

Inside `createFullModel`, you do things like:

```
self.dataset = dataset self.createTensorsAndSplitting(dataset)  
self.doModelCreation(complexity, dataset, sampledParams)
```

and likely store:

```
self.trainBatch, self.testBatch = ...
```

So after each call:

- The `MLP_Manager` object keeps references to the latest dataset tensors and loaders.
- When the next dataset or complexity is processed, those attributes get overwritten.

If you never go back to re-use the old ones (i.e., you train, evaluate, and discard results before moving to the next dataset/complexity), this won't *break anything*.

✓ So: functionally, it's fine — each call just resets the state.



## When it *would* cause problems

You'd only run into real issues if:

1. You try to compare or store results from multiple runs inside the same manager (e.g., average accuracy per dataset, or keep model weights),
2. You later try to reuse the manager for a different dataset without fully reinitializing its DataLoaders, or
3. You accidentally mix MNIST and CIFAR (different input shapes) within one manager without clearing old model parameters.

For example, if you ran:

```
mlpModel.createFullModel("easy", "mnist") mlpModel.createFullModel("hard",  
"cifar")
```

and your CNN internally expected 3 channels for CIFAR, but some lingering tensor from MNIST (1 channel) got used → boom, shape mismatch.

## In short

- If each call to `createFullModel()` completely recreates all necessary internal objects (model, optimizer, criterion, dataloaders, etc.),  
→  then reusing a single manager is *fine*.
  - If the manager holds any persistent state you might later want to compare or analyze (e.g. store results across runs, or maintain multiple models in memory),  
→  then it's safer to instantiate a new one per run.
- 

## TL;DR

Situation	Reuse One Manager?
Each run is independent and results are printed then discarded	 Yes, fine
You store or compare results later	 Better to reinstantiate
Datasets differ in input shape (MNIST vs CIFAR) and same manager handles both	 Risky unless you fully reset internal state

---

Would you like me to show a *lightweight refactor* that makes reusing the same manager fully safe across datasets (so you don't need to instantiate multiple times)?

You said:

also how can i upload code from vs code to google colab?

ChatGPT said:

Great question — there are three easy ways to upload and run your VS Code code in Google Colab, depending on how your files are organized. Here's the full breakdown 



## Option 1 — Upload files directly in Colab

Best for small scripts or quick tests

1. Open a new Colab notebook: <https://colab.research.google.com>

2. In a code cell, run:
  3. `from google.colab import files` `uploaded = files.upload()`
  4. Choose your `.py` files (from VS Code's project folder).
  5. After upload, Colab shows their names. You can then import or run them:
  6. `import my_script # or !python my_script.py`
-  Tip: if you have multiple related files (like `model.py`, `mlp.py`, `cnn.py`), upload all of them at once.
- 

## Option 2 — Mount your Google Drive

Best if your VS Code project is in Google Drive or you want to persist files

1. In Colab:
2. `from google.colab import drive` `drive.mount('/content/drive')`
3. Navigate to your project:
4. `%cd /content/drive/MyDrive/your_folder`
5. Then just run:
6. `!python main.py`  
or import your modules normally if they're in the same directory.

 This way, any edits or new outputs are saved back to Drive.

---

## Option 3 — Push to GitHub (recommended for real projects)

Best for organized or versioned codebases

1. In VS Code, push your code to a GitHub repo.
2. In Colab, clone it:
3. `!git clone https://github.com/yourusername/your-repo.git` `%cd your-repo`
4. Run normally:
5. `!python main.py`

 Bonus: you can `!git pull` later to sync new changes from VS Code.

---

## Optional: make Colab behave more like VS Code

- Use Colab's file browser (left panel → “Files” tab) to drag/drop .py files directly.
  - Or upload a .zip of your project and unzip it:
  - ```
from google.colab import files
uploaded = files.upload() # upload yourproject.zip
!unzip yourproject.zip
%cd yourproject
!python main.py
```
- 

## Summary

| Goal                          | Best Method                 |
|-------------------------------|-----------------------------|
| Quick test                    | <code>files.upload()</code> |
| Persistent environment        | Mount Google Drive          |
| Full project or collaboration | GitHub clone                |
| Whole project folder          | Zip + unzip                 |

---

Would you like me to show you the exact commands for uploading and running your current `MLP_Manager + Model` code in Colab (so everything imports correctly)?

You said:

im trying it out in colab, got this

---

ValueError Traceback (most recent call last)

/tmp/ipython-input-2853117360.py in <cell line: 0>()

----> 1 from mlp import MLP\_Manager

2 from cnn import CNN\_Manager

3

4 complexities = ["low", "medium", "high"]

5

2 frames

```
/usr/local/lib/python3.12/dist-packages/torch/__init__.py in <module>
    414     if USE_GLOBAL_DEPS:
    415         _load_global_deps()
--> 416     from torch._C import * # noqa: F403
    417
    418
```

ValueError: module functions cannot set METH\_CLASS or METH\_STATIC

ChatGPT said:

That's a known PyTorch bug — good catch 😊

This error:

```
ValueError: module functions cannot set METH_CLASS or METH_STATIC
```

usually happens in Google Colab when it's running Python 3.12, which Colab switched to recently, while PyTorch (preinstalled) isn't fully compatible with it yet.

---

## ✓ Fix (takes 30 seconds)

Run this at the top of your Colab notebook (before importing torch):

```
!pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cpu
```

or, if you have a GPU runtime enabled:

```
!pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu121
```

Then restart the runtime (click “Runtime → Restart runtime”), and import again:

```
import torch from mlp import MLP_Manager
```

---

## Why this happens

Colab recently upgraded to Python 3.12, but the preinstalled PyTorch version is compiled for Python 3.11.

Reinstalling PyTorch from the official wheel fixes that mismatch.

---

Would you like me to give you a short setup cell that installs all dependencies (torch, tensorflow, etc.) in the correct order for your `MLP_Manager + Model` code? That way you can just copy/paste it into the top of your Colab notebook.

You said:

when i push a change to github, how can i get google colab to see those changes?

ChatGPT said:

Excellent question — this is one of the most common workflow pain points when using Google Colab + GitHub together 😊

Here's the deal:

---



## When you “push” changes to GitHub...

Colab doesn't automatically know about them — it only pulls the repo snapshot that existed when you last cloned it.

So you need to manually pull the latest changes inside Colab.

---



## To make Colab see your new GitHub changes

### Option 1: Use `!git pull` (recommended if you cloned the repo already)

If you've already cloned your repo into Colab, just re-pull the updates:

```
%cd /content/your-repo-name !git pull
```

That'll fetch your latest commits from GitHub.

If you see something like:

Already up to date.

then your Colab copy already matches your GitHub repo.

---

## Option 2: Re-clone from scratch (useful for fresh sessions)

Colab resets its environment every time you restart it, so re-clone the repo each time:

```
!git clone https://github.com/your-username/your-repo-name.git %cd  
          your-repo-name
```

If you push new changes later, either:

- do `!git pull` inside the same Colab session, or
  - re-run the two lines above to get a fresh copy.
- 

## Option 3: Force Colab to “refresh” a specific file

If you only changed one file (say `mlp.py`), you can directly overwrite it:

```
!rm mlp.py !wget
```

```
https://raw.githubusercontent.com/your-username/your-repo-name/main/mlp.py
```

This re-downloads the latest version from GitHub without re-cloning the repo.

---

### Important note

If you made local edits to the Colab copy (e.g., tweaked a file inside `/content/your-repo-name`) without committing them, `git pull` may fail because of merge conflicts.

In that case, reset your Colab environment or delete the folder first:

```
!rm -rf /content/your-repo-name !git clone  
https://github.com/your-username/your-repo-name.git
```

---

Would you like me to give you a short setup cell that:

1. Clones your repo,
2. Installs dependencies, and
3. Automatically reloads updates when you push new commits to GitHub?

You said:

any idea why these ran so slow? the high complexity MNIST took over an hour, when tuning wasn't even really done. This was on Google Colab's Free GPU

```
from abc import ABC, abstractmethod
from tensorflow.keras.datasets import mnist, cifar10
import random
import time
from torch.utils.data import TensorDataset, DataLoader, random_split
import torch

class Model(ABC):
    @abstractmethod
    def doCustomProcessing(self):
        pass
    def retrieveDataset(self, setType):
        if setType == "mnist":
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
        else:
            (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

        # normalizing them. Converting pixel values between 0 - 255 to 0 - 1
        xTrain = xTrain.astype('float32') / 255.0
        xTest = xTest.astype('float32') / 255.0

        return (xTrain, yTrain, xTest, yTest)

    # initialize method
    def __init__(self):
```

```

(self.mnistXTrain, self.mnistYTrain, self.mnistXTest, self.mnistYTest) =
    self.retrieveDataset("mnist")
(self.cifarXTrain, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
    self.retrieveDataset("cifar")

        self.parameters = {
    "learningRate": [0.01, 0.001, 0.0001],
    "batchSize": [32, 64, 128],
    "optimizer": ["sgd", "adam"],
    "dropoutRate": [0.2, 0.5]
}

def sampleParams(self):
    return {key: random.choice(values) for key, values in self.parameters.items()}

def createTensorsAndSplitting(self):
    self.mnistXTrain = torch.tensor(self.mnistXTrain, dtype=torch.float32)
    self.mnistXTest = torch.tensor(self.mnistXTest, dtype=torch.float32)
    self.mnistYTrain = torch.tensor(self.mnistYTrain, dtype=torch.long).squeeze()
    self.mnistYTest = torch.tensor(self.mnistYTest, dtype=torch.long).squeeze()

    self.cifarXTrain = torch.tensor(self.cifarXTrain, dtype=torch.float32)
    self.cifarXTest = torch.tensor(self.cifarXTest, dtype=torch.float32)
    self.cifarYTrain = torch.tensor(self.cifarYTrain, dtype=torch.long).squeeze()
    self.cifarYTest = torch.tensor(self.cifarYTest, dtype=torch.long).squeeze()

mnistTrain = TensorDataset(self.mnistXTrain, self.mnistYTrain)
cifarTrain = TensorDataset(self.cifarXTrain, self.cifarYTrain)

self.mnistTrain, self.mnistValid = random_split(mnistTrain, [50000, 10000])

self.cifarTrain, self.cifarValid = random_split(cifarTrain, [45000, 5000])

@abstractmethod

```

```

def doModelCreation(self, complexity, dataset, sampledParams):
    pass

    def createBatches(self, batchSize, dataset):

        if(dataset == "mnist"):

self.trainBatch = DataLoader(self.mnistTrain, batch_size = batchSize, shuffle =
True)

self.testBatch = DataLoader(self.mnistValid, batch_size = batchSize)
else:

self.trainBatch = DataLoader(self.cifarTrain, batch_size = batchSize, shuffle =
True)

self.testBatch = DataLoader(self.cifarValid, batch_size = batchSize)

    def createFullModel(self, complexity, dataset, modelName):

        startTime = time.time()
        tunelIterations = 1
        self.finalParams = {}
        highestAccuracy = 0

        for _ in range(0, tunelIterations, 1):
            sampledParams = self.sampleParams()

            self.doModelCreation(complexity, dataset, sampledParams)

            self.createBatches(sampledParams["batchSize"], dataset)

accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch)

        if(accuracy > highestAccuracy):
            highestAccuracy = accuracy
            self.finalParams = sampledParams

    self.doModelCreation(complexity, dataset, self.finalParams)
    self.createBatches(self.finalParams["batchSize"], dataset)

```

```
accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch)

endTime = time.time()

numMinutes = (endTime - startTime) / 60
numSeconds = (endTime - startTime) % 60

print(f"Accuracy for {modelName} model with complexity of {complexity} and
dataset of {dataset} was {accuracy}, while total elapsed time was {numMinutes} minutes
and {numSeconds} seconds. Additionally, the parameters chosen were:
{self.finalParams}")
```

```
from mlp import MLP_Manager
from cnn import CNN_Manager

complexities = ["low", "medium", "high"]

datasetTypes = ["mnist", "cifar"]

mlpModel = MLP_Manager()

cnnModel = CNN_Manager()

for dataset in datasetTypes:
    for complexityLevel in complexities:
        mlpModel.createFullModel(complexityLevel,dataset)

        for dataset in datasetTypes:
            for complexityLevel in complexities:
                cnnModel.createFullModel(complexityLevel,dataset)

from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
```

```

from torch.utils.data import DataLoader, TensorDataset

class MLP_Manager(Model):
    def doCustomProcessing(self):
        # a mlp just expects a 1 dimensional array as its input, so we must flatten our 2D
        # array to a 1D one.
        self.mnistXTrain = self.mnistXTrain.reshape(self.mnistXTrain.shape[0],
  self.mnistXTrain.shape[1] * self.mnistXTrain.shape[2])
        self.mnistXTest = self.mnistXTest.reshape(self.mnistXTest.shape[0],
  self.mnistXTest.shape[1] * self.mnistXTest.shape[2])

        # may not make a difference, but just in case we transpose before flattening. Do
        # this because for the cnn, we have to do this as
        # it expects channel first
        self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)
        self.cifarXTest = self.cifarXTest.transpose(0, 3, 1, 2)

        # a mlp just expects a 1 dimensional array as its input, so we must flatten our 3D
        # array to a 1D one.
        self.cifarXTrain = self.cifarXTrain.reshape(self.cifarXTrain.shape[0],
  self.cifarXTrain.shape[1] * self.cifarXTrain.shape[2] *
  self.cifarXTrain.shape[3])

        self.cifarXTest = self.cifarXTest.reshape(self.cifarXTest.shape[0],
  self.cifarXTest.shape[1] * self.cifarXTest.shape[2] *
  self.cifarXTest.shape[3])

        # do the common pre processing before doing what's special to the MLP
        def __init__(self):
            super().__init__()
            # now, custom stuff done
            self.doCustomProcessing()

            self.createTensorsAndSplitting()

```

```

def doModelCreation(self, complexity, dataset, sampledParams):
    # first dimension is # of points, so the one after is the number of features
        if(dataset == "mnist"):
            numFeatures = self.mnistXTrain.shape[1]
        else:
            numFeatures = self.cifarXTrain.shape[1]

    self.currentModel = MLP(complexity, numFeatures, **sampledParams)

class MLP(nn.Module):
    def __init__(self, complexity, numFeatures, learningRate, optimizer, dropoutRate,
                 batchSize):
        super().__init__()
        layers = []

        # do all the layers except for the final layer
        beforeHiddenLayerLength = -1
        if(complexity == "low"):
            numPerceptrons = 128
        layers.append(nn.Linear(numFeatures, numPerceptrons))
        layers.append(nn.ReLU())
        beforeHiddenLayerLength = numPerceptrons
        elif(complexity == "medium"):
            numPerceptronsFirst = 512
            numPerceptronsSecond = 256
            numPerceptronsThird = 128

        layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
        layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
        layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))

```

```

        layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsThird
    elif(complexity == "high"):
        numPerceptronsFirst = 1024
        numPerceptronsSecond = 512
        numPerceptronsThird = 256
        numPerceptronsFourth = 128
        numPerceptronsFifth = 64
        numPerceptronsSixth = 32

        layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
        layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
    layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
    layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsThird, numPerceptronsFourth))
    layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFourth, numPerceptronsFifth))
    layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFifth, numPerceptronsSixth))
    layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsSixth

    # first, remove one hidden layer
    layers.append(nn.Dropout(dropoutRate))

    # now do the final layer
    numClasses = 10
layers.append(nn.Linear(beforeHiddenLayerLength, numClasses))

self.model = nn.Sequential(*layers)

```

```

        self.criterion = nn.CrossEntropyLoss()
        self.num_epochs = 50

        if(optimizer == "adam"):
            self.optimizer = optim.Adam(self.parameters(), lr = learningRate)
        else:
            self.optimizer = optim.SGD(self.parameters(), lr = learningRate)

        def forward(self, x):
            return self.model(x)

    def trainAndEvaluate(self, trainSet, testSet):
        for epoch in range(self.num_epochs):
            for batchX, batchY in trainSet:
                # zero out gradients
                self.optimizer.zero_grad()

                rawOutputs = self(batchX)

                loss = self.criterion(rawOutputs, batchY)

                # do backward step
                loss.backward()

                self.optimizer.step()

                numCorrect = 0
                numTotal = 0
                with torch.no_grad():
                    for batchX, batchY in testSet:
                        output = self(batchX)
                        predictedOutputs = torch.argmax(output, dim = 1)

                        for i in range(0, len(predictedOutputs), 1):
                            if(predictedOutputs[i] == batchY[i]):

```

```

        numCorrect += 1
        numTotal += 1

    return numCorrect / numTotal

from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import numpy as np

class CNN_Manager(Model):
    # this method does custom processing needed for the CNN
    def doCustomProcessing(self):
        # the CNN expects a 3D array; the MNIST is 2D as it has no dimension for color
        # (its monochrome), so we
        # squeeze in another dimension
        self.mnistXTrain = np.expand_dims(self.mnistXTrain, axis = 1)
        self.mnistXTest = np.expand_dims(self.mnistXTest, axis = 1)

        # the cnn expects color dimension first, so just swap the dimensions for this
        self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)
        self.cifarXTest = self.cifarXTest.transpose(0, 3, 1, 2)

    # initial function that runs the common processing, then does what's special to the
    # CNN
    def __init__(self):
        super().__init__()
        # now, custom stuff done
        self.doCustomProcessing()

        self.createTensorsAndSplitting()

def doModelCreation(self, complexity, dataset, sampledParams):

```

```

# first dimension is # of points, so the one after is the number of features
    if(dataset == "mnist"):
        numChannels = 1
    else:
        numChannels = 3

self.currentModel = CNN(complexity, numChannels, **sampledParams)

class CNN(nn.Module):
def __init__(self, complexity, numChannels, learningRate, optimizer, dropoutRate,
             batchSize):
    super().__init__()
    layers = []

    finalNumFilters = 0

    if(complexity == "low"):
        numFiltersFirstLayer = 32
        convStepSize = 1
        firstKernelSize = 5
    layers.append(nn.Conv2d(numChannels, numFiltersFirstLayer, firstKernelSize,
                          padding = 1))
    layers.append(nn.ReLU())

# dimension of each filter after will be 26 x 26 for MNIST, and 30 x 30 for CIFAR.
# SO, we can use the following pooling fine

```

```

    poolKernelSize = 2
    poolStepSize = poolKernelSize
    layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))

```

```

# after this, dimension of each filter will be 13 x 13 for MNIST, and 15 x 15 for
# CIFAR
    numFiltersSecondLayer = 64
    secondKernelSize = 3

```

```

layers.append(nn.Conv2d(numFiltersFirstLayer, numFiltersSecondLayer,
                      secondKernelSize, padding = 1))
    layers.append(nn.ReLU())

    layers.append(nn.AdaptiveAvgPool2d(1))

finalNumFilters = numFiltersSecondLayer
elif(complexity == "medium"):
    numFiltersFirstLayer = 32
        convStepSize = 1
        firstKernelSize = 5
layers.append(nn.Conv2d(numChannels, numFiltersFirstLayer, firstKernelSize,
                      padding = 1))
    layers.append(nn.BatchNorm2d(numFiltersFirstLayer))
    layers.append(nn.ReLU())

```

# dimension of each filter after will be 26 x 26 for MNIST, and 30 x 30 for CIFAR.  
SO, we can use the following pooling fine

```

poolKernelSize = 2
poolStepSize = poolKernelSize
layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))

```

# after this, dimension of each filter will be 12 x 12 for MNIST, and 14 x 14 for  
CIFAR

```

numFiltersSecondLayer = 64
secondKernelSize = 3
layers.append(nn.Conv2d(numFiltersFirstLayer, numFiltersSecondLayer,
                      secondKernelSize, padding = 1))

```

```

    layers.append(nn.BatchNorm2d(numFiltersSecondLayer))
    layers.append(nn.ReLU())

```

```

    layers.append(nn.AdaptiveAvgPool2d(1))

```

```

finalNumFilters = numFiltersSecondLayer

```

```

        elif(complexity == "high"):
            numFiltersFirstLayer = 32
            convStepSize = 1
            firstKernelSize = 3
layers.append(nn.Conv2d(numChannels, numFiltersFirstLayer, firstKernelSize,
                      padding = 1))
layers.append(nn.BatchNorm2d(numFiltersFirstLayer))
layers.append(nn.ReLU())

```

# dimension of each filter after will be 28 x 28 for MNIST, and 32 x 32 for CIFAR.  
 SO, we can use the following pooling fine

```

        poolKernelSize = 2
        poolStepSize = poolKernelSize
layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))

```

# after this, dimension of each filter will be 14 x 14 for MNIST, and 16 x 16 for  
 CIFAR

```

        numFiltersSecondLayer = 64
        secondKernelSize = 3
layers.append(nn.Conv2d(numFiltersFirstLayer, numFiltersSecondLayer,
                      secondKernelSize, padding = 1))
layers.append(nn.BatchNorm2d(numFiltersSecondLayer))
layers.append(nn.ReLU())

```

```
layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))
```

```

        numFiltersThirdLayer = 128
        secondKernelSize = 2
layers.append(nn.Conv2d(numFiltersSecondLayer, numFiltersThirdLayer,
                      secondKernelSize, padding = 1))
layers.append(nn.BatchNorm2d(numFiltersThirdLayer))
layers.append(nn.ReLU())

```

```
layers.append(nn.AdaptiveAvgPool2d(1))
```

```

finalNumFilters = numFiltersThirdLayer

    layers.append(nn.flatten())
layers.append(nn.Linear(finalNumFilters, finalNumFilters * 2))
    layers.append(nn.ReLU())

    # first, remove one hidden layers
layers.append(nn.Dropout(dropoutRate))

    numClasses = 10
layers.append(nn.Linear(finalNumFilters * 2, numClasses))

self.model = nn.Sequential(*layers)

self.criterion = nn.CrossEntropyLoss()
self.num_epochs = 50

if(optimizer == "adam"):
self.optimizer = optim.Adam(self.parameters(), lr = learningRate)
else:
self.optimizer = optim.SGD(self.parameters(), lr = learningRate)

def forward(self, x):
    return self.model(x)

def trainAndEvaluate(self, trainSet, testSet):
    for epoch in range(self.num_epochs):
        for batchX, batchY in trainSet:
            # zero out gradients
            self.optimizer.zero_grad()

            rawOutputs = self(batchX)

            loss = self.criterion(rawOutputs, batchY)

```

```
# do backward step
loss.backward()

self.optimizer.step()

numCorrect = 0
numTotal = 0
with torch.no_grad():
    for batchX, batchY in testSet:
        output = self(batchX)
predictedOutputs = torch.argmax(output, dim = 1)

for i in range(0, len(predictedOutputs), 1):
    if(predictedOutputs[i] == batchY[i]):
        numCorrect += 1
        numTotal += 1

return numCorrect / numTotal
```

ChatGPT said:

Yeah — that *definitely* shouldn't take an hour on Colab's GPU for MNIST 😅

MNIST is a tiny dataset, and even a moderately complex CNN should finish in under 2–5 minutes per model on a free GPU (like a T4).

So if it's taking that long, something's off.

Let's go step-by-step — here are the main causes and fixes ↴



## 1 Your model isn't actually using the GPU

This is *the* most common cause.

In your code, there's no `.to(device)` for either your model or your data batches.

So all training is running on the CPU — which in Colab is slow for PyTorch.

### ✓ Fix:

Add this near the top of your file (e.g. in `Model` or inside `trainAndEvaluate`):

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Then update:

```
    self.to(device)
```

And inside training:

```
for batchX, batchY in trainSet: batchX, batchY = batchX.to(device),  
    batchY.to(device)
```

And similarly for testing:

```
for batchX, batchY in testSet: batchX, batchY = batchX.to(device),  
    batchY.to(device)
```

This alone will probably cut your training time down by 10x to 50x.

---

## ⌚ 2 Your training loop is doing full dataset training for every model + 50 epochs

You're looping over both MLP and CNN, for both MNIST and CIFAR, and for low, medium, and high complexity — each with 50 epochs.

That's:

```
2 models × 2 datasets × 3 complexity levels × 50 epochs = 600 epochs total
```

And you're doing this sequentially on a single Colab runtime.

Even on GPU, that's hours of work.

### ✓ Fix:

Start small — maybe:

```
self.num_epochs = 10
```

Or even 5 for testing your setup.

Once it works fast and correctly, raise it later.

---



### 3 No batching on GPU before training

When you move your data to GPU, always do it per batch, not before creating DataLoader.

(You're already doing this right, since your DataLoader yields CPU tensors — just ensure you move them inside the loop, not before.)

---



### 4 No progress printing or `model.train() / model.eval()` mode switching

You're running a full training loop without visibility — so it looks “frozen.”

Add this inside your training loop:

```
if epoch % 5 == 0: print(f"Epoch [{epoch+1}/{self.num_epochs}] - Loss:  
{loss.item():.4f}")
```

Also add:

```
self.train()
```

before training, and

```
self.eval()
```

before evaluation (with `torch.no_grad()`).

---



### 5 Your dataset conversion + splitting happen twice

You're calling `createTensorsAndSplitting()` for both MLP and CNN managers.

That's fine — but be aware it creates new `TensorDataset` objects each time, which is

fine but memory-heavy.

Not the main slowdown, though.

---

## 6 Extra overhead: DataLoader default settings

By default, `DataLoader` uses `num_workers=0`, which can make loading slow.

 Fix:

When creating `DataLoader`:

```
self.trainBatch = DataLoader(self.mnistTrain, batch_size=batchSize,  
                            shuffle=True, num_workers=2, pin_memory=True)
```

---



## TL;DR — Quick fixes checklist

| Problem                                                   | Fix                                                                |
|-----------------------------------------------------------|--------------------------------------------------------------------|
| Model running on CPU                                      | Add <code>.to(device)</code> everywhere                            |
| Too many epochs                                           | Reduce to 5–10 for now                                             |
| No <code>model.train()</code> / <code>model.eval()</code> | Add them before loops                                              |
| DataLoader slow                                           | Use <code>num_workers</code> and <code>pin_memory=True</code>      |
| No GPU batch transfers                                    | Move <code>batchX</code> , <code>batchY</code> to GPU inside loops |

---

If you add those changes, the “high” complexity MNIST should train in under 3 minutes on Colab’s free GPU.

If you want, I can show you exactly how to modify your `trainAndEvaluate()` to properly handle device + mode switching (a 5-line edit). Want me to?

You said:

if say i have two TensorDatasets, like say a training and validation set both of that type,  
how can i merge them in the Dataloader?

ChatGPT said:

Good question — that's actually a common thing to want to do 

If you have two `TensorDataset` objects (say `train_dataset` and `valid_dataset`) and  
you want to combine them into one dataset (so you can pass them both to a single  
`DataLoader`), there are two main ways to do it:

---

### **Option 1 — Use `torch.utils.data.ConcatDataset` (recommended)**

This is the clean and official way to merge any two datasets of the same structure.

```
from torch.utils.data import ConcatDataset, DataLoader
combined_dataset = ConcatDataset([train_dataset, valid_dataset])
combined_loader = DataLoader(combined_dataset, batch_size=64, shuffle=True)
```

 You can combine *any number* of datasets this way — e.g.

```
ConcatDataset([train_dataset, valid_dataset, test_dataset])
```

---

### **Option 2 — Manually concatenate tensors (if you know both are `TensorDatasets`)**

If both datasets are specifically `TensorDatasets`, you can access their internal tensors  
and merge them directly.

```
import torch
from torch.utils.data import TensorDataset, DataLoader #  

# Assuming both have same feature/label shapes
x_combined = torch.cat((train_dataset.tensors[0], valid_dataset.tensors[0]), dim=0)
y_combined = torch.cat((train_dataset.tensors[1],
valid_dataset.tensors[1]), dim=0)
combined_dataset =
```

```
TensorDataset(X_combined, y_combined) combined_loader =  
DataLoader(combined_dataset, batch_size=64, shuffle=True)
```

This gives you more control (e.g., if you want to apply extra transformations or label adjustments before merging).

---

### A note on `shuffle=True`

If you merge datasets and use `shuffle=True` in the DataLoader, your batches will mix samples from both sets randomly.

If you still want to keep them distinct, you should keep separate DataLoaders for train and validation instead of merging them.

---

### TL;DR

- Use `ConcatDataset` → simplest and cleanest.
  - Use `torch.cat()` → more control but more code.
  - Don't merge if you actually need to evaluate separately.
- 

Would you like me to show how to merge them conditionally (like only if you pass a flag `mergeTrainAndValid=True`)? That's a neat pattern if you sometimes want them merged and sometimes not.

You said:

also, how do you do the gpu tool thing?

ChatGPT said:

Ah — great question 🤓 you're talking about using your GPU in PyTorch (for faster training), right?

Here's the quick guide to doing it properly — especially for Google Colab.

---

 1 Enable GPU in Colab

Go to:

Runtime → Change runtime type → Hardware accelerator → GPU

Then click Save.

You can confirm it's available with:

```
import torch
print(torch.cuda.is_available()) # should print True
print(torch.cuda.get_device_name(0)) # prints the GPU model, like 'Tesla
                                    T4'
```

## 2 Set up your device in code

You typically define a `device` variable once, like this:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Then whenever you create a model or tensor, you move it to that device:

```
model = MyModel()
model.to(device)
```

And for your data:

```
batchX, batchY = batchX.to(device), batchY.to(device)
```

## 3 Full example inside your training loop

You can modify your training loop like this:

```
for epoch in range(num_epochs):
    for batchX, batchY in train_loader:
        batchX, batchY = batchX.to(device), batchY.to(device)
        optimizer.zero_grad()
        outputs = model(batchX)
        loss = criterion(outputs, batchY)
        loss.backward()
        optimizer.step()
```

And for testing/evaluation:

```
with torch.no_grad():
    for batchX, batchY in test_loader:
        batchX, batchY = batchX.to(device), batchY.to(device)
        outputs = model(batchX)
```

## 4 Move entire datasets at once (optional)

If your dataset fits in GPU memory, you can move tensors directly before creating your

DataLoader:

```
train_dataset = TensorDataset(X_train.to(device), y_train.to(device))
```

But this can use a lot of GPU RAM, so the usual approach is to move them batch by batch inside the loop (like above).

---

## Summary

| Step         | Code                                                                                   | Purpose              |
|--------------|----------------------------------------------------------------------------------------|----------------------|
| Check GPU    | <code>torch.cuda.is_available()</code>                                                 | See if GPU is active |
| Set device   | <code>device = torch.device("cuda" if<br/>torch.cuda.is_available() else "cpu")</code> | Use GPU if possible  |
| Move model   | <code>model.to(device)</code>                                                          | Load model to GPU    |
| Move tensors | <code>x, y = x.to(device), y.to(device)</code>                                         | Load batch to GPU    |

---

Would you like me to show you how to integrate this `device` setup directly into your existing `trainAndEvaluate()` function from your MLP code? That would make it GPU-ready with minimal edits.

You said:

`model = MyModel()` `model.to(device)`  
where would you do this part?

ChatGPT said:

Perfect question — this part:

```
model = MyModel() model.to(device)
```

should happen right after you create your model — so in your case, that's inside your `doModelCreation()` method for both the MLP and CNN classes.

---

## Example for your code (MLP)

You currently have this in `MLP_Manager`:

```
def doModelCreation(self, complexity, dataset, sampledParams): if(dataset == "mnist"): numFeatures = self.mnistXTrain.shape[1] else: numFeatures = self.cifarXTrain.shape[1] self.currentModel = MLP(complexity, numFeatures, **sampledParams)
```

You'd modify it to:

```
def doModelCreation(self, complexity, dataset, sampledParams): if(dataset == "mnist"): numFeatures = self.mnistXTrain.shape[1] else: numFeatures = self.cifarXTrain.shape[1] device = torch.device("cuda" if torch.cuda.is_available() else "cpu") self.device = device # store it for later use self.currentModel = MLP(complexity, numFeatures, **sampledParams).to(device)
```

---

## Then, update your training function

Inside your `trainAndEvaluate()` method (in both `MLP` and `CNN`), right before using your batches, move them to the same device:

```
def trainAndEvaluate(self, trainSet, testSet): device = next(self.parameters()).device # gets whatever device the model is on for epoch in range(self.num_epochs): for batchX, batchY in trainSet: batchX, batchY = batchX.to(device), batchY.to(device) self.optimizer.zero_grad() rawOutputs = self(batchX) loss = self.criterion(rawOutputs, batchY) loss.backward() self.optimizer.step()
```

And same for evaluation:

```
with torch.no_grad(): for batchX, batchY in testSet: batchX, batchY =  
    batchX.to(device), batchY.to(device) output = self(batchX)  
    predictedOutputs = torch.argmax(output, dim=1)
```

---

✓ In short:

1. Add `.to(device)` when you create the model.
2. Move each batch to the same device during training/testing.

That's all you need for Colab to use the GPU efficiently 🚀

Would you like me to show how to do this for both your `MLP` and `CNN` classes together in one clean edit (so you can just copy-paste it)?

You said:

`device = next(self.parameters()).device`

how does this work?

ChatGPT said:

Excellent question — that line:

```
device = next(self.parameters()).device
```

is a PyTorch shortcut for finding out what device (CPU or GPU) your model's parameters are currently stored on.

Let's break it down 👇

---

## 🔍 Step-by-step explanation

1. `self.parameters()`
  - Every PyTorch `nn.Module` (like your MLP or CNN) has a method called `.parameters()`.
  - It returns an iterator over all the model's trainable tensors (the weights and biases).
2. `next(self.parameters())`

- Since `.parameters()` returns an iterator, calling `next()` just gives you the first parameter tensor in the model.
- Every parameter is a `torch.Tensor` that lives on some device (CPU or GPU).

### 3. `.device`

- Each tensor in PyTorch has a `.device` attribute that tells you where it's stored — for example:

- `device(type='cpu')`
- `device(type='cuda', index=0)`

So, this one-liner:

```
device = next(self.parameters()).device
```

grabs the first parameter in the model and checks which device it's on, so you can then move your input batches to the same device automatically.

---

## Why it's useful

Instead of manually tracking whether the model is on the CPU or GPU (like storing `self.device` everywhere), you just ask the model directly — it will always tell you the correct device.

So you can safely write:

```
device = next(self.parameters()).device batchX, batchY =  
        batchX.to(device), batchY.to(device)
```

and this works no matter whether you're running on:

- Google Colab GPU,
  - CPU-only laptop,
  - or even multi-GPU setup.
- 

Would you like a tiny print-debug version that shows the device of the model and each batch (so you can confirm it's running on GPU)?

You said:

Alright, any errors left here?

```
from abc import ABC, abstractmethod
from tensorflow.keras.datasets import mnist, cifar10
import random
import time
from torch.utils.data import TensorDataset, DataLoader, random_split, ConcatDataset
import torch

class Model(ABC):
    @abstractmethod
    def doCustomProcessing(self):
        pass
    def retrieveDataset(self, setType):
        if setType == "mnist":
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
        else:
            (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

        # normalizing them. Converting pixel values between 0 - 255 to 0 - 1
        xTrain = xTrain.astype('float32') / 255.0
        xTest = xTest.astype('float32') / 255.0

        return (xTrain, yTrain, xTest, yTest)

    # initialize method
    def __init__(self):
        (self.mnistXTrain, self.mnistYTrain, self.mnistXTest, self.mnistYTest) =
            self.retrieveDataset("mnist")
        (self.cifarXTrain, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
            self.retrieveDataset("cifar")

        self.parameters = {
```

```

        "learningRate": [0.01, 0.001, 0.0001],
        "batchSize": [32, 64, 128],
        "optimizer": ["sgd", "adam"],
        "dropoutRate": [0.2, 0.5]
    }

self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def sampleParams(self):
    return {key: random.choice(values) for key, values in self.parameters.items()}

def createTensorsAndSplitting(self):
    self.mnistXTrain = torch.tensor(self.mnistXTrain, dtype=torch.float32)
    self.mnistXTest = torch.tensor(self.mnistXTest, dtype=torch.float32)
    self.mnistYTrain = torch.tensor(self.mnistYTrain, dtype=torch.long).squeeze()
    self.mnistYTest = torch.tensor(self.mnistYTest, dtype=torch.long).squeeze()

    self.cifarXTrain = torch.tensor(self.cifarXTrain, dtype=torch.float32)
    self.cifarXTest = torch.tensor(self.cifarXTest, dtype=torch.float32)
    self.cifarYTrain = torch.tensor(self.cifarYTrain, dtype=torch.long).squeeze()
    self.cifarYTest = torch.tensor(self.cifarYTest, dtype=torch.long).squeeze()

mnistTrain = TensorDataset(self.mnistXTrain, self.mnistYTrain)
cifarTrain = TensorDataset(self.cifarXTrain, self.cifarYTrain)

self.mnistTest = TensorDataset(self.mnistXTest, self.mnistYTest)
self.cifarTest = TensorDataset(self.cifarXTest, self.cifarYTest)

self.mnistTrain, self.mnistValid = random_split(mnistTrain, [50000, 10000])

self.cifarTrain, self.cifarValid = random_split(cifarTrain, [45000, 5000])

@abstractmethod
def doModelCreation(self, complexity, dataset, sampledParams):

```

```

        pass

def createBatches(self, batchSize, dataset, doingFinalTrain):

    if(dataset == "mnist"):
        if(doingFinalTrain):
            mergedTrain = ConcatDataset([self.mnistTrain, self.mnistValid])
    self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
                                True)
    self.testBatch = DataLoader(self.mnistTest, batch_size = batchSize)
    else:
        self.trainBatch = DataLoader(self.mnistTrain, batch_size = batchSize, shuffle
                                    = True)
        self.testBatch = DataLoader(self.mnistValid, batch_size = batchSize)
    else:
        if(doingFinalTrain):
            mergedTrain = ConcatDataset([self.cifarTrain, self.cifarValid])
    self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
                                True)
    self.testBatch = DataLoader(self.cifarTest, batch_size = batchSize)
    else:
        self.trainBatch = DataLoader(self.cifarTrain, batch_size = batchSize, shuffle =
                                    True)
        self.testBatch = DataLoader(self.cifarValid, batch_size = batchSize)

def createFullModel(self, complexity, dataset, modelName):
    startTime = time.time()
    tunelIterations = 1
    self.finalParams = {}
    highestAccuracy = 0

    for _ in range(0, tunelIterations, 1):
        sampledParams = self.sampleParams()

        self.doModelCreation(complexity, dataset, sampledParams)

```

```
    self.createBatches(sampledParams["batchSize"], dataset, False)

    accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,
  False)

        if(accuracy > highestAccuracy):
            highestAccuracy = accuracy
            self.finalParams = sampledParams

    self.doModelCreation(complexity, dataset, self.finalParams)

    self.createBatches(self.finalParams["batchSize"], dataset, True)
    accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,
  True)

    endTime = time.time()

    numMinutes = (endTime - startTime) / 60
    numSeconds = (endTime - startTime) % 60

    print(f"Accuracy for {modelName} model with complexity of {complexity} and
dataset of {dataset} was {accuracy}, while total elapsed time was {numMinutes} minutes
and {numSeconds} seconds. Additionally, the parameters chosen were:
{self.finalParams}")
```

```
from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
```

```
class MLP_Manager(Model):
    def doCustomProcessing(self):
```

```
# a mlp just expects a 1 dimensional array as its input, so we must flatten our 2D  
array to a 1D one.
```

```
self.mnistXTrain = self.mnistXTrain.reshape(self.mnistXTrain.shape[0],  
   self.mnistXTrain.shape[1] * self.mnistXTrain.shape[2])  
self.mnistXTest = self.mnistXTest.reshape(self.mnistXTest.shape[0],  
   self.mnistXTest.shape[1] * self.mnistXTest.shape[2])
```

```
# may not make a difference, but just in case we transpose before flattening. Do  
this because for the cnn, we have to do this as
```

```
# it expects channel first
```

```
self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)  
self.cifarXTest = self.cifarXTest.transpose(0, 3, 1, 2)
```

```
# a mlp just expects a 1 dimensional array as its input, so we must flatten our 3D  
array to a 1D one.
```

```
self.cifarXTrain = self.cifarXTrain.reshape(self.cifarXTrain.shape[0],  
   self.cifarXTrain.shape[1] * self.cifarXTrain.shape[2] *  
   self.cifarXTrain.shape[3])
```

```
self.cifarXTest = self.cifarXTest.reshape(self.cifarXTest.shape[0],  
   self.cifarXTest.shape[1] * self.cifarXTest.shape[2] *  
   self.cifarXTest.shape[3])
```

```
# do the common pre processing before doing what's special to the MLP
```

```
def __init__(self):  
    super().__init__()  
    # now, custom stuff done  
    self.doCustomProcessing()
```

```
self.createTensorsAndSplitting()
```

```
def doModelCreation(self, complexity, dataset, sampledParams):
```

```
# first dimension is # of points, so the one after is the number of features
```

```
if(dataset == "mnist"):
```

```
    numFeatures = self.mnistXTrain.shape[1]
```

```

        else:
            numFeatures = self.cifarXTrain.shape[1]

        self.currentModel = MLP(complexity, numFeatures,
                               **sampledParams).to(self.device)

class MLP(nn.Module):
    def __init__(self, complexity, numFeatures, learningRate, optimizer, dropoutRate,
                 batchSize):
        super().__init__()
        layers = []

        # do all the layers except for the final layer
        beforeHiddenLayerLength = -1
        if(complexity == "low"):
            numPerceptrons = 128
        layers.append(nn.Linear(numFeatures, numPerceptrons))
        layers.append(nn.ReLU())
        beforeHiddenLayerLength = numPerceptrons
        elif(complexity == "medium"):
            numPerceptronsFirst = 512
            numPerceptronsSecond = 256
            numPerceptronsThird = 128

        layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
        layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
        layers.append(nn.ReLU())
        layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
        layers.append(nn.ReLU())

        beforeHiddenLayerLength = numPerceptronsThird
        elif(complexity == "high"):
```

```

        numPerceptronsFirst = 1024
        numPerceptronsSecond = 512
        numPerceptronsThird = 256
        numPerceptronsFourth = 128
        numPerceptronsFifth = 64
        numPerceptronsSixth = 32

    layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsThird, numPerceptronsFourth))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsFourth, numPerceptronsFifth))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(numPerceptronsFifth, numPerceptronsSixth))
        layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsSixth

        # first, remove one hidden layers
    layers.append(nn.Dropout(dropoutRate))

        # now do the final layer
        numClasses = 10
    layers.append(nn.Linear(beforeHiddenLayerLength, numClasses))

self.model = nn.Sequential(*layers)

self.criterion = nn.CrossEntropyLoss()

        if(optimizer == "adam"):
    self.optimizer = optim.Adam(self.parameters(), lr = learningRate)

```

```

        else:
    self.optimizer = optim.SGD(self.parameters(), lr = learningRate)

    def forward(self, x):
        return self.model(x)

def trainAndEvaluate(self, trainSet, testSet, finalTrain, device):
    if(finalTrain):
        numEpochs = 50
    else:
        numEpochs = 30

    for epoch in range(numEpochs):
        for batchX, batchY in trainSet:
            batchX, batchY = batchX.to(device), batchY.to(device)

            # zero out gradients
            self.optimizer.zero_grad()

            rawOutputs = self(batchX)

            loss = self.criterion(rawOutputs, batchY)

            # do backward step
            loss.backward()

            self.optimizer.step()

            numCorrect = 0
            numTotal = 0
            with torch.no_grad():
                for batchX, batchY in testSet:
                    batchX, batchY = batchX.to(device), batchY.to(device)
                    output = self(batchX)
                    predictedOutputs = torch.argmax(output, dim = 1)

```

```

        for i in range(0, len(predictedOutputs), 1):
            if(predictedOutputs[i] == batchY[i]):
                numCorrect += 1
                numTotal += 1

        return numCorrect / numTotal

from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import numpy as np

class CNN_Manager(Model):
    # this method does custom processing needed for the CNN
    def doCustomProcessing(self):
        # the CNN expects a 3D array; the MNIST is 2D as it has no dimension for color
        # (its monochrome), so we
        # squeeze in another dimension
        self.mnistXTrain = np.expand_dims(self.mnistXTrain, axis = 1)
        self.mnistXTest = np.expand_dims(self.mnistXTest, axis = 1)

        # the cnn expects color dimension first, so just swap the dimensions for this
        self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)
        self.cifarXTest = self.cifarXTest.transpose(0, 3, 1, 2)

# initial function that runs the common processing, then does what's special to the
# CNN
def __init__(self):
    super().__init__()
    # now, custom stuff done
    self.doCustomProcessing()

```

```

        self.createTensorsAndSplitting()

    def doModelCreation(self, complexity, dataset, sampledParams):
        # first dimension is # of points, so the one after is the number of features
        if(dataset == "mnist"):
            numChannels = 1
        else:
            numChannels = 3

        self.currentModel = CNN(complexity, numChannels,
                               **sampledParams).to(self.device)

    class CNN(nn.Module):
        def __init__(self, complexity, numChannels, learningRate, optimizer, dropoutRate,
                     batchSize):
            super().__init__()
            layers = []

            finalNumFilters = 0

            if(complexity == "low"):
                numFiltersFirstLayer = 32
                convStepSize = 1
                firstKernelSize = 5
                layers.append(nn.Conv2d(numChannels, numFiltersFirstLayer, firstKernelSize,
                                      padding = 1))
                layers.append(nn.ReLU())

        # dimension of each filter after will be 26 x 26 for MNIST, and 30 x 30 for CIFAR.
        # SO, we can use the following pooling fine

            poolKernelSize = 2
            poolStepSize = poolKernelSize
            layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))

```

```

# after this, dimension of each filter will be 13 x 13 for MNIST, and 15 x 15 for
# CIFAR
    numFiltersSecondLayer = 64
        secondKernelSize = 3
layers.append(nn.Conv2d(numFiltersFirstLayer, numFiltersSecondLayer,
secondKernelSize, padding = 1))
    layers.append(nn.ReLU())

    layers.append(nn.AdaptiveAvgPool2d(1))

finalNumFilters = numFiltersSecondLayer
elif(complexity == "medium"):
    numFiltersFirstLayer = 32
        convStepSize = 1
        firstKernelSize = 5
layers.append(nn.Conv2d(numChannels, numFiltersFirstLayer, firstKernelSize,
padding = 1))
    layers.append(nn.BatchNorm2d(numFiltersFirstLayer))
    layers.append(nn.ReLU())

```

# dimension of each filter after will be 26 x 26 for MNIST, and 30 x 30 for CIFAR.  
SO, we can use the following pooling fine

```

    poolKernelSize = 2
    poolStepSize = poolKernelSize
layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))

```

# after this, dimension of each filter will be 12 x 12 for MNIST, and 14 x 14 for
# CIFAR
 numFiltersSecondLayer = 64
 secondKernelSize = 3
layers.append(nn.Conv2d(numFiltersFirstLayer, numFiltersSecondLayer,
secondKernelSize, padding = 1))
 layers.append(nn.BatchNorm2d(numFiltersSecondLayer))
 layers.append(nn.ReLU())



```
        layers.append(nn.BatchNorm2d(numFiltersThirdLayer))
        layers.append(nn.ReLU())

    layers.append(nn.AdaptiveAvgPool2d(1))

    finalNumFilters = numFiltersThirdLayer

    layers.append(nn.flatten())
    layers.append(nn.Linear(finalNumFilters, finalNumFilters * 2))
    layers.append(nn.ReLU())

    # first, remove one hidden layers
    layers.append(nn.Dropout(dropoutRate))

    numClasses = 10
    layers.append(nn.Linear(finalNumFilters * 2, numClasses))

    self.model = nn.Sequential(*layers)

    self.criterion = nn.CrossEntropyLoss()
    self.num_epochs = 30

    if(optimizer == "adam"):
        self.optimizer = optim.Adam(self.parameters(), lr = learningRate)
    else:
        self.optimizer = optim.SGD(self.parameters(), lr = learningRate)

    def forward(self, x):
        return self.model(x)

def trainAndEvaluate(self, trainSet, testSet, finalTrain, device):
    if(finalTrain):
        numEpochs = 50
    else:
        numEpochs = 30
```

```

        for epoch in range(self.num_epochs):
            for batchX, batchY in trainSet:
                batchX, batchY = batchX.to(device), batchY.to(device)

                    # zero out gradients
                    self.optimizer.zero_grad()

                rawOutputs = self(batchX)

                loss = self.criterion(rawOutputs, batchY)

                    # do backward step
                    loss.backward()

                self.optimizer.step()

                numCorrect = 0
                numTotal = 0
                with torch.no_grad():
                    for batchX, batchY in testSet:
                        batchX, batchY = batchX.to(device), batchY.to(device)
                        output = self(batchX)
                        predictedOutputs = torch.argmax(output, dim = 1)

                        for i in range(0, len(predictedOutputs), 1):
                            if(predictedOutputs[i] == batchY[i]):
                                numCorrect += 1
                                numTotal += 1

                return numCorrect / numTotal

from mlp import MLP_Manager
from cnn import CNN_Manager

```

```
complexities = ["low", "medium", "high"]

datasetTypes = ["mnist", "cifar"]

mlpModel = MLP_Manager()

cnnModel = CNN_Manager()

    for dataset in datasetTypes:
        for complexityLevel in complexities:
            mlpModel.createFullModel(complexityLevel,dataset)

            for dataset in datasetTypes:
                for complexityLevel in complexities:
                    cnnModel.createFullModel(complexityLevel,dataset)
```

ChatGPT said:

You're extremely close — this is actually very well organized and clean.

There are just a few small errors or inconsistencies that will cause runtime issues or logic mismatches.

Let's go through them carefully 

---

## 1. Missing `modelName` argument when calling `createFullModel`

In your driver code at the end:

```
for dataset in datasetTypes: for complexityLevel in complexities:
    mlpModel.createFullModel(complexityLevel, dataset)
```

but your function definition is:

```
def createFullModel(self, complexity, dataset, modelName):
```

 Fix:

Pass in the model name:

```
for dataset in datasetTypes: for complexityLevel in complexities:  
    mlpModel.createFullModel(complexityLevel, dataset, "MLP") for dataset in  
        datasetTypes: for complexityLevel in complexities:  
            cnnModel.createFullModel(complexityLevel, dataset, "CNN")
```



## ⚠ 2. `trainAndEvaluate` signatures mismatch

- In your MLP/CNN classes, you have:
- `def trainAndEvaluate(self, trainSet, testSet, finalTrain, device):`
- But you call it like this:
- `accuracy = self.currentModel.trainAndEvaluate(self.trainBatch,  
self.testBatch, False)`  
→ You didn't pass the `device` argument.

✓ Fix: either

- Pass the device:
- `accuracy = self.currentModel.trainAndEvaluate(self.trainBatch,  
self.testBatch, False, self.device)`  
and same for the final training call,  
OR
- Remove the `device` argument from the method and infer it internally:
- `device = next(self.parameters()).device`



## ⚠ 3. Minor logic mismatch in CNN training loop

You have both `self.num_epochs = 30` and this in `trainAndEvaluate`:

```
if(finalTrain): numEpochs = 50 else: numEpochs = 30 for epoch in  
    range(self.num_epochs):
```

→ The loop uses `self.num_epochs`, ignoring the condition.

✓ Fix:

Change it to:

```
numEpochs = 50 if finalTrain else 30 for epoch in range(numEpochs):
```

---

## ⚠ 4. Dropout comment typo

In both MLP and CNN:

```
# first, remove on hidden layers layers.append(nn.Dropout(dropoutRate))
```

— looks like a typo; you meant:

```
# apply dropout before the final layer
```

---

## ⚠ 5. CIFAR reshape issue (in `MLP_Manager.doCustomProcessing`)

You do:

```
self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2) self.cifarXTest  
= self.cifarXTest.transpose(0, 3, 1, 2) self.cifarXTrain =  
    self.cifarXTrain.reshape(...)
```

This is fine, but note that transposing then flattening is equivalent to just flattening the original image; the transpose has no effect here since MLPs don't use channels.

- ✓ Not technically wrong, just redundant — you can skip the transpose in the MLP version.

---

## ⚠ 6. Device transfer consistency

In both MLP and CNN you do:

```
batchX, batchY = batchX.to(device), batchY.to(device)
```

Good — but note you'll also need to ensure your model is on the same device in

```
createFullModel():
```

```
    self.currentModel = MLP(...).to(self.device)
```

(which you already do ✓)

## ⚠️ 7. Slightly misleading variable name in CNN flattening

You do:

```
layers.append(nn.flatten())
```

→ `nn.flatten` should be `nn.Flatten()` (capital F, it's a class).

✓ Fix:

```
layers.append(nn.Flatten())
```

## ✓ Optional improvements

- Add progress feedback inside training:
- `print(f"Epoch [{epoch+1}/{numEpochs}], Loss: {loss.item():.4f}")` just for debugging.
- Use:
- `numMinutes = int((endTime - startTime) // 60)` `numSeconds = int((endTime - startTime) % 60)` to avoid decimals in time output.

## ✓ Final Summary of Fixes

| Issue                              | File    | Fix                                                         |
|------------------------------------|---------|-------------------------------------------------------------|
| Missing <code>modelName</code> arg | main    | Add <code>"MLP"</code> / <code>"CNN"</code>                 |
| Device arg mismatch                | MLP/CNN | Add <code>device</code> when calling or infer it internally |
| Epoch loop                         | MLP/CNN | Use <code>for epoch in range(numEpochs)</code>              |
| Typo in dropout comment            | Both    | Fix comment                                                 |
| Flatten layer                      | CNN     | Use <code>nn.Flatten()</code>                               |

---

|                     |             |                                  |
|---------------------|-------------|----------------------------------|
| Redundant transpose | MLP_Manager | (Optional) remove before flatten |
| Time formatting     | Model       | Use integer mins/secs            |

---

Would you like me to show a cleaned-up, ready-to-run version of all three files

(`model.py`, `mlp.py`, and `cnn.py`) that integrates these fixes?

You said:

anything else to change?

```
from mlp import MLP_Manager
from cnn import CNN_Manager

complexities = ["low", "medium", "high"]

datasetTypes = ["mnist", "cifar"]

mlpModel = MLP_Manager()

cnnModel = CNN_Manager()

for dataset in datasetTypes:
    for complexityLevel in complexities:
        mlpModel.createFullModel(complexityLevel,dataset, "MLP")

        for dataset in datasetTypes:
            for complexityLevel in complexities:
                cnnModel.createFullModel(complexityLevel,dataset, "CNN")

from abc import ABC, abstractmethod
from tensorflow.keras.datasets import mnist, cifar10
import random
import time

from torch.utils.data import TensorDataset, DataLoader, random_split, ConcatDataset
```

```

import torch

class Model(ABC):
    @abstractmethod
    def doCustomProcessing(self):
        pass
    def retrieveDataset(self, setType):
        if setType == "mnist":
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
        else:
            (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

        # normalizing them. Converting pixel values between 0 - 255 to 0 - 1
        xTrain = xTrain.astype('float32') / 255.0
        xTest = xTest.astype('float32') / 255.0

        return (xTrain, yTrain, xTest, yTest)

    # initialize method
    def __init__(self):
        (self.mnistXTrain, self.mnistYTrain, self.mnistXTest, self.mnistYTest) =
            self.retrieveDataset("mnist")
        (self.cifarXTrain, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
            self.retrieveDataset("cifar")

        self.parameters = {
            "learningRate": [0.01, 0.001, 0.0001],
            "batchSize": [32, 64, 128],
            "optimizer": ["sgd", "adam"],
            "dropoutRate": [0.2, 0.5]
        }

self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

```

```

        def sampleParams(self):
            return {key: random.choice(values) for key, values in self.parameters.items()}

        def createTensorsAndSplitting(self):
            self.mnistXTrain = torch.tensor(self.mnistXTrain, dtype=torch.float32)
            self.mnistXTest = torch.tensor(self.mnistXTest, dtype=torch.float32)
            self.mnistYTrain = torch.tensor(self.mnistYTrain, dtype=torch.long).squeeze()
            self.mnistYTest = torch.tensor(self.mnistYTest, dtype=torch.long).squeeze()

            self.cifarXTrain = torch.tensor(self.cifarXTrain, dtype=torch.float32)
            self.cifarXTest = torch.tensor(self.cifarXTest, dtype=torch.float32)
            self.cifarYTrain = torch.tensor(self.cifarYTrain, dtype=torch.long).squeeze()
            self.cifarYTest = torch.tensor(self.cifarYTest, dtype=torch.long).squeeze()

    mnistTrain = TensorDataset(self.mnistXTrain, self.mnistYTrain)
    cifarTrain = TensorDataset(self.cifarXTrain, self.cifarYTrain)

    self.mnistTest = TensorDataset(self.mnistXTest, self.mnistYTest)
    self.cifarTest = TensorDataset(self.cifarXTest, self.cifarYTest)

    self.mnistTrain, self.mnistValid = random_split(mnistTrain, [50000, 10000])

    self.cifarTrain, self.cifarValid = random_split(cifarTrain, [45000, 5000])

    @abstractmethod
    def doModelCreation(self, complexity, dataset, sampledParams):
        pass

    def createBatches(self, batchSize, dataset, doingFinalTrain):

        if(dataset == "mnist"):
            if(doingFinalTrain):
                mergedTrain = ConcatDataset([self.mnistTrain, self.mnistValid])

```

```

self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
                           True)
self.testBatch = DataLoader(self.mnistTest, batch_size = batchSize)
else:
    self.trainBatch = DataLoader(self.mnistTrain, batch_size = batchSize, shuffle
                               = True)
    self.testBatch = DataLoader(self.mnistValid, batch_size = batchSize)
else:
    if(doingFinalTrain):
        mergedTrain = ConcatDataset([self.cifarTrain, self.cifarValid])
self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
                           True)
self.testBatch = DataLoader(self.cifarTest, batch_size = batchSize)
else:
    self.trainBatch = DataLoader(self.cifarTrain, batch_size = batchSize, shuffle =
                               True)
    self.testBatch = DataLoader(self.cifarValid, batch_size = batchSize)

def createFullModel(self, complexity, dataset, modelName):
    startTime = time.time()
    tuneIterations = 1
    self.finalParams = {}
    highestAccuracy = 0

    for _ in range(0, tuneIterations, 1):
        sampledParams = self.sampleParams()

        self.doModelCreation(complexity, dataset, sampledParams)

        self.createBatches(sampledParams["batchSize"], dataset, False)

        accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,
  False, self.device)

        if(accuracy > highestAccuracy):

```

```

        highestAccuracy = accuracy
        self.finalParams = sampledParams

    self.doModelCreation(complexity, dataset, self.finalParams)

    self.createBatches(self.finalParams["batchSize"], dataset, True)
accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,
   True, self.device)

    endTime = time.time()

    numMinutes = (endTime - startTime) / 60
    numSeconds = (endTime - startTime) % 60

print(f"Accuracy for {modelName} model with complexity of {complexity} and
dataset of {dataset} was {accuracy}, while total elapsed time was {numMinutes} minutes
and {numSeconds} seconds. Additionally, the parameters chosen were:
{self.finalParams}")

```

```

from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

```

```

class MLP_Manager(Model):
    def doCustomProcessing(self):
# a mlp just expects a 1 dimensional array as its input, so we must flatten our 2D
array to a 1D one.
        self.mnistXTrain = self.mnistXTrain.reshape(self.mnistXTrain.shape[0],
   self.mnistXTrain.shape[1] * self.mnistXTrain.shape[2])
        self.mnistXTest = self.mnistXTest.reshape(self.mnistXTest.shape[0],
  self.mnistXTest.shape[1] * self.mnistXTest.shape[2])

```



```

        class MLP(nn.Module):
def __init__(self, complexity, numFeatures, learningRate, optimizer, dropoutRate,
            batchSize):
    super().__init__()
    layers = []

# do all the layers except for the final layer
beforeHiddenLayerLength = -1
if(complexity == "low"):
    numPerceptrons = 128
layers.append(nn.Linear(numFeatures, numPerceptrons))
    layers.append(nn.ReLU())
beforeHiddenLayerLength = numPerceptrons
elif(complexity == "medium"):
    numPerceptronsFirst = 512
    numPerceptronsSecond = 256
    numPerceptronsThird = 128
    layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
    layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
    layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
    layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsThird
elif(complexity == "high"):
    numPerceptronsFirst = 1024
    numPerceptronsSecond = 512
    numPerceptronsThird = 256
    numPerceptronsFourth = 128
    numPerceptronsFifth = 64
    numPerceptronsSixth = 32

```

```

        layers.append(nn.Linear(numFeatures, numPerceptronsFirst))
                layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFirst, numPerceptronsSecond))
                layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsSecond, numPerceptronsThird))
                layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsThird, numPerceptronsFourth))
                layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFourth, numPerceptronsFifth))
                layers.append(nn.ReLU())
layers.append(nn.Linear(numPerceptronsFifth, numPerceptronsSixth))
                layers.append(nn.ReLU())

beforeHiddenLayerLength = numPerceptronsSixth

        # apply dropout before the final layer
        layers.append(nn.Dropout(dropoutRate))

        # now do the final layer
        numClasses = 10
layers.append(nn.Linear(beforeHiddenLayerLength, numClasses))

        self.model = nn.Sequential(*layers)

        self.criterion = nn.CrossEntropyLoss()

        if(optimizer == "adam"):
            self.optimizer = optim.Adam(self.parameters(), lr = learningRate)
            else:
            self.optimizer = optim.SGD(self.parameters(), lr = learningRate)

        def forward(self, x):
                return self.model(x)

def trainAndEvaluate(self, trainSet, testSet, finalTrain, device):

```

```

        if(finalTrain):
            numEpochs = 50
        else:
            numEpochs = 30

    for epoch in range(numEpochs):
        for batchX, batchY in trainSet:
            batchX, batchY = batchX.to(device), batchY.to(device)

            # zero out gradients
            self.optimizer.zero_grad()

            rawOutputs = self(batchX)

            loss = self.criterion(rawOutputs, batchY)

            # do backward step
            loss.backward()

            self.optimizer.step()

            numCorrect = 0
            numTotal = 0
            with torch.no_grad():
                for batchX, batchY in testSet:
                    batchX, batchY = batchX.to(device), batchY.to(device)
                    output = self(batchX)
                    predictedOutputs = torch.argmax(output, dim = 1)

                    for i in range(0, len(predictedOutputs), 1):
                        if(predictedOutputs[i] == batchY[i]):
                            numCorrect += 1
                            numTotal += 1

    return numCorrect / numTotal

```

```

from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import numpy as np

class CNN_Manager(Model):
    # this method does custom processing needed for the CNN
    def doCustomProcessing(self):
        # the CNN expects a 3D array; the MNIST is 2D as it has no dimension for color
        # (its monochrome), so we
        # squeeze in another dimension
        self.mnistXTrain = np.expand_dims(self.mnistXTrain, axis = 1)
        self.mnistXTest = np.expand_dims(self.mnistXTest, axis = 1)

        # the cnn expects color dimension first, so just swap the dimensions for this
        self.cifarXTrain = self.cifarXTrain.transpose(0, 3, 1, 2)
        self.cifarXTest = self.cifarXTest.transpose(0, 3, 1, 2)

    # initial function that runs the common processing, then does what's special to the
    # CNN
    def __init__(self):
        super().__init__()
        # now, custom stuff done
        self.doCustomProcessing()

        self.createTensorsAndSplitting()

    def doModelCreation(self, complexity, dataset, sampledParams):
        # first dimension is # of points, so the one after is the number of features
        if(dataset == "mnist"):
            numChannels = 1
        else:

```

```

        numChannels = 3

    self.currentModel = CNN(complexity, numChannels,
                           **sampledParams).to(self.device)

    class CNN(nn.Module):
def __init__(self, complexity, numChannels, learningRate, optimizer, dropoutRate,
            batchSize):
    super().__init__()
    layers = []

    finalNumFilters = 0

    if(complexity == "low"):
        numFiltersFirstLayer = 32
        convStepSize = 1
        firstKernelSize = 5
        layers.append(nn.Conv2d(numChannels, numFiltersFirstLayer, firstKernelSize,
                             padding = 1))
        layers.append(nn.ReLU())

# dimension of each filter after will be 26 x 26 for MNIST, and 30 x 30 for CIFAR.
# SO, we can use the following pooling fine

        poolKernelSize = 2
        poolStepSize = poolKernelSize
        layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))

# after this, dimension of each filter will be 13 x 13 for MNIST, and 15 x 15 for
# CIFAR
        numFiltersSecondLayer = 64
        secondKernelSize = 3
        layers.append(nn.Conv2d(numFiltersFirstLayer, numFiltersSecondLayer,
                             secondKernelSize, padding = 1))
        layers.append(nn.ReLU())

```

```

        layers.append(nn.AdaptiveAvgPool2d(1))

        finalNumFilters = numFiltersSecondLayer
        elif(complexity == "medium"):
            numFiltersFirstLayer = 32
            convStepSize = 1
            firstKernelSize = 5
        layers.append(nn.Conv2d(numChannels, numFiltersFirstLayer, firstKernelSize,
                             padding = 1))
        layers.append(nn.BatchNorm2d(numFiltersFirstLayer))
        layers.append(nn.ReLU())
    
```

# dimension of each filter after will be 26 x 26 for MNIST, and 30 x 30 for CIFAR.  
 SO, we can use the following pooling fine

```

        poolKernelSize = 2
        poolStepSize = poolKernelSize
        layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))
    
```

# after this, dimension of each filter will be 12 x 12 for MNIST, and 14 x 14 for  
 CIFAR  
 numFiltersSecondLayer = 64  
 secondKernelSize = 3  
 layers.append(nn.Conv2d(numFiltersFirstLayer, numFiltersSecondLayer,
 secondKernelSize, padding = 1))
 layers.append(nn.BatchNorm2d(numFiltersSecondLayer))
 layers.append(nn.ReLU())

```
        layers.append(nn.AdaptiveAvgPool2d(1))
```

```

        finalNumFilters = numFiltersSecondLayer
        elif(complexity == "high"):
            numFiltersFirstLayer = 32
            convStepSize = 1
    
```

```
    firstKernelSize = 3  
layers.append(nn.Conv2d(numChannels, numFiltersFirstLayer, firstKernelSize,  
padding = 1))
```

```
    layers.append(nn.BatchNorm2d(numFiltersFirstLayer))
```

```
    layers.append(nn.ReLU())
```

# dimension of each filter after will be 28 x 28 for MNIST, and 32 x 32 for CIFAR.  
SO, we can use the following pooling fine

```
    poolKernelSize = 2
```

```
    poolStepSize = poolKernelSize
```

```
layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))
```

# after this, dimension of each filter will be 14 x 14 for MNIST, and 16 x 16 for  
CIFAR

```
    numFiltersSecondLayer = 64
```

```
    secondKernelSize = 3
```

```
layers.append(nn.Conv2d(numFiltersFirstLayer, numFiltersSecondLayer,  
secondKernelSize, padding = 1))
```

```
    layers.append(nn.BatchNorm2d(numFiltersSecondLayer))
```

```
    layers.append(nn.ReLU())
```

```
layers.append(nn.MaxPool2d(poolKernelSize, poolStepSize))
```

```
    numFiltersThirdLayer = 128
```

```
    secondKernelSize = 2
```

```
layers.append(nn.Conv2d(numFiltersSecondLayer, numFiltersThirdLayer,  
secondKernelSize, padding = 1))
```

```
    layers.append(nn.BatchNorm2d(numFiltersThirdLayer))
```

```
    layers.append(nn.ReLU())
```

```
layers.append(nn.AdaptiveAvgPool2d(1))
```

```
finalNumFilters = numFiltersThirdLayer
```

```

        layers.append(nn.Flatten())
layers.append(nn.Linear(finalNumFilters, finalNumFilters * 2))
        layers.append(nn.ReLU())

        if(complexity != "low"):
# apply dropout before the final layer
layers.append(nn.Dropout(dropoutRate))

        numClasses = 10
layers.append(nn.Linear(finalNumFilters * 2, numClasses))

        self.model = nn.Sequential(*layers)

        self.criterion = nn.CrossEntropyLoss()

        if(optimizer == "adam"):
self.optimizer = optim.Adam(self.parameters(), lr = learningRate)
        else:
self.optimizer = optim.SGD(self.parameters(), lr = learningRate)

        def forward(self, x):
            return self.model(x)

def trainAndEvaluate(self, trainSet, testSet, finalTrain, device):
        if(finalTrain):
            numEpochs = 50
        else:
            numEpochs = 30

        for epoch in range(numEpochs):
            for batchX, batchY in trainSet:
batchX, batchY = batchX.to(device), batchY.to(device)

            # zero out gradients
            self.optimizer.zero_grad()

```

```
rawOutputs = self(batchX)

loss = self.criterion(rawOutputs, batchY)

# do backward step
loss.backward()

self.optimizer.step()

numCorrect = 0
numTotal = 0
with torch.no_grad():
    for batchX, batchY in testSet:
        batchX, batchY = batchX.to(device), batchY.to(device)
        output = self(batchX)
        predictedOutputs = torch.argmax(output, dim = 1)

        for i in range(0, len(predictedOutputs), 1):
            if(predictedOutputs[i] == batchY[i]):
                numCorrect += 1
                numTotal += 1

return numCorrect / numTotal
```

ChatGPT said:

Excellent — this is a very clean and complete setup, and your structure is actually quite close to what a production ML experiment manager would look like.

At this point, there are no major breaking issues left. However, I'll list a few optional refinements and subtle correctness/performance improvements that would make your code both cleaner and more PyTorch-idiomatic.

---

 **High-Priority Tweaks (Recommended)**

## 1. Progress printing inside `trainAndEvaluate()`

Right now, the model trains silently for all epochs.

Add a quick print (especially for Colab) to track training:

```
for epoch in range(numEpochs): runningLoss = 0.0 for batchX, batchY in trainSet: batchX, batchY = batchX.to(device), batchY.to(device) self.optimizer.zero_grad() rawOutputs = self(batchX) loss = self.criterion(rawOutputs, batchY) loss.backward() self.optimizer.step() runningLoss += loss.item() if (epoch + 1) % 5 == 0 or epoch == numEpochs - 1: print(f"Epoch [{epoch+1}/{numEpochs}], Loss: {runningLoss / len(trainSet):.4f}")
```

---

## 2. Accuracy calculation efficiency

Your accuracy loop currently checks predictions one by one:

```
for i in range(0, len(predictedOutputs), 1): if(predictedOutputs[i] == batchY[i]): numCorrect += 1 numTotal += 1
```

This is O(n) Python-level looping.

In PyTorch, do it vectorized for a ~100× speedup:

✓ Replace with:

```
numCorrect += (predictedOutputs == batchY).sum().item() numTotal += batchY.size(0)
```

---

## 3. Use `torch.utils.data.ConcatDataset` for merging validation and training

You're already doing this correctly — good job — but note that you could simplify:

```
mergedTrain = self.mnistTrain + self.mnistValid
```

PyTorch's `ConcatDataset` supports `+` operator overloading as of 2024+, so this is slightly cleaner but optional.

---

## 4. More precise time formatting

Instead of showing float minutes and seconds:

```
elapsed = endTime - startTime minutes = int(elapsed // 60) seconds =
int(elapsed % 60) print(f"... elapsed time: {minutes}m {seconds}s")
```

---

## 5. Avoid `torch.flatten` confusion

In your CNN:

```
layers.append(nn.Flatten())
```

 That's correct — earlier you fixed it from `nn.flatten()` (lowercase). Just confirm all CNNs use the capitalized version.

---

## 6. AdaptiveAvgPool2d shape

You're using `nn.AdaptiveAvgPool2d(1)` correctly — it'll output `(batch, channels, 1, 1)`. Flattening after that will produce a tensor of shape `(batch, channels)`.

 No fix needed, but note that this is exactly why your final linear input equals `finalNumFilters`.

---

## Optional / Quality-of-Life Improvements

### 7. Save best model automatically

You could store best weights after hyperparameter tuning:

```
best_state_dict = None highestAccuracy = 0 ... if accuracy >
highestAccuracy: highestAccuracy = accuracy best_state_dict =
```

```
self.currentModel.state_dict().copy() ...  
self.currentModel.load_state_dict(best_state_dict)
```

This ensures the final retrain starts from the best configuration rather than random init again.

---

## 8. Add evaluation mode for testing

Wrap testing with:

```
self.eval() with torch.no_grad(): ... self.train()
```

so dropout/batchnorm behave deterministically during testing.

---

## 9. GPU sanity check print

Inside `Model.__init__()`:

```
print(f"Using device: {self.device}")
```

This helps you confirm Colab is on GPU.

---

## 10. For CIFAR labels

CIFAR10 labels are shape `(N, 1)`, and you do `.squeeze()` — good.

- ✓ Just make sure you never accidentally squeeze the wrong dimension (MNIST is already `(N, )`).
- 

## ✓ Summary of Key Fixes/Improvements

| Priority | Issue                | Fix                                                  |
|----------|----------------------|------------------------------------------------------|
| ★        | Epoch loop printing  | Add progress display                                 |
| ★        | Accuracy computation | Vectorize using <code>(preds == labels).sum()</code> |

---

|    |                          |                                                              |
|----|--------------------------|--------------------------------------------------------------|
| ★  | Eval mode during testing | Use <code>model.eval()</code> and <code>model.train()</code> |
| ★  | Time formatting          | Use integer minutes/seconds                                  |
| ✓  | Flatten layer            | Use <code>nn.Flatten()</code> (correct now)                  |
| ✓  | Dropout placement        | Done correctly before final layer                            |
| ⚙️ | Auto-save best weights   | Optional improvement                                         |
| ⚙️ | Device printout          | Optional debug info                                          |

---

Would you like me to show what your final cleaned version of `trainAndEvaluate()` (for both CNN and MLP) would look like with all of these improvements baked in? It's only about 15 lines longer but much cleaner and faster.

You said:  
is there any way to get around this?

Cannot connect to GPU backend  
You cannot currently connect to a GPU due to usage limits in Colab. Learn more To get more access to GPUs, consider purchasing Colab compute units with Pay As You Go.

ChatGPT said:  
Yeah — that message means your free GPU quota on Google Colab has run out (it's based on recent GPU usage). Unfortunately, Google limits how much GPU time free-tier users get per day/week.

Here are a few ways to get around it:

---

## Option 1: Use a different Colab account

- Google tracks GPU usage per Google account.
  - If you log in with another account (e.g., a school or secondary Gmail), you'll usually get fresh GPU quota immediately.
  - You can upload your notebook and re-run it there.
- 

## Option 2: Use Colab Pay-as-you-go

- Colab now has a “Compute Units” system.
  - You can buy a few dollars’ worth of compute units and get consistent GPU access (often A100 or T4 GPUs).
  - To activate:  
Colab menu → Runtime → Change runtime type → GPU → Pay-as-you-go
- 



## Option 3: Use a local GPU runtime

If your computer (or another machine you can SSH into) has an NVIDIA GPU:

1. Install the Colab runtime locally:
2. 

```
pip install jupyterlab jupyter_http_over_ws jupyter_serverextension enable --py jupyter_http_over_ws
```
3. Start Jupyter:
4. 

```
jupyter notebook  
--NotebookApp.allow_origin='https://colab.research.google.com'  
--port=8888
```
5. In Colab:  
→ Connect → Connect to local runtime  
→ Enter: `http://localhost:8888`

Now Colab will use your local GPU instead of Google's.

---

## Option 4: Use a free cloud alternative

You can also switch to other platforms that often have *fewer GPU restrictions*:

- Kaggle Notebooks → free T4 GPU, resets every session
  - Paperspace Gradient → free GPU tier (with daily reset)
  - RunPod, Lambda Labs, or Google Cloud free trial if you want more power.
- 

Would you like me to show you how to move your current Colab notebook to Kaggle  
(which gives a free GPU instantly)?

You said:

if i do pay as you go, how much does it cost to run it for about 2-3 hours?

ChatGPT said:

Good question! The cost for using Google Colab on a Pay As You Go basis depends on which GPU you get and how quickly you burn through “compute units” (CUs). Here’s a breakdown:

---

## \$ Pricing overview

- 100 compute units cost US \$9.99.
  - [Google Colab](#)
  - [+2](#)
  - [TechCrunch](#)
  - [+2](#)
  - Each GPU uses up a certain number of compute units per hour. For example:
    - A T4 GPU ~ 1.8 units/hour → ≈ US \$0.18/hour.
      - [McCormickML](#)
    - A V100 GPU ~ 4.9 units/hour → ≈ US \$0.49/hour.
      - [McCormickML](#)
    - An A100 GPU ~ 11.8 units/hour → ≈ US \$1.18/hour.
      - [McCormickML](#)
- 

## 🔍 Estimated cost for 2-3 hours

Based on the typical GPU usage above:

- If you get a T4 GPU, 2–3 hours would cost roughly US \$0.36 to US \$0.54 (at ~\$0.18/hr).
  - If you get a V100 GPU, 2–3 hours would cost roughly US \$0.98 to US \$1.47 (at ~\$0.49/hr).
  - If you get an A100 GPU, 2–3 hours would cost roughly US \$2.36 to US \$3.55 (at ~\$1.18/hr).
- 

## ⚠ Important caveats

- The exact GPU you get is not guaranteed — Colab allocates GPU types dynamically based on availability and plan.
  - [TechCrunch](#)
  - [+1](#)
  - The “units/hour” rate may vary by region, demand, and the exact hardware instance.
  - Even if you pay for units, Colab may pre-empt sessions or impose other usage caps (especially under heavy load).
  - When you exhaust the units, you will fall back to free-tier limits.
  - [Stack Overflow](#)
- 

If you like, I can check current live rates for your region (US, Europe, etc.) as of this week — that way you’ll know exactly what to expect.

You said:  
do i need to apply a softmax here?

```
# loop through epochs
for epoch in range(numEpochs):
    for batchX, batchY in trainSet:
        # convert them go GPU
        batchX, batchY = batchX.to(device), batchY.to(device)

        # zero out gradients
        self.optimizer.zero_grad()
```

```
# get the results  
rawOutputs = self(batchX)  
  
loss = self.criterion(rawOutputs, batchY)  
  
# do backward step  
loss.backward()  
  
self.optimizer.step()
```

ChatGPT said:

Short answer: No — you should NOT apply softmax there.

And you are currently doing it correctly.

---

## ✓ Why you **should not** apply softmax before CrossEntropyLoss

PyTorch's:

```
nn.CrossEntropyLoss()
```

already includes a softmax inside it, along with a negative log-likelihood (NLL) step.

So if you apply a softmax manually, you'd be doing:

```
softmax → log → NLL
```

on top of the loss's own internal:

```
(softmax → log → NLL)
```

This causes:

- double-softmax, which distorts class probabilities
  - vanishing gradients
  - worse accuracy
-

## ✓ What you *should* give to the loss function

`rawOutputs` should be raw logits (the direct linear layer output).

And `batchY` should be the integer class labels.

You are already doing exactly that:

```
rawOutputs = self(batchX) loss = self.criterion(rawOutputs, batchY)
```

This is perfect.

---

## ? When DO you apply softmax?

Only when you want human-readable probabilities, e.g., during evaluation *after* computing accuracy:

```
probs = torch.softmax(output, dim=1)
```

But not during training and not for the loss input.

---

## ★ Final answer:

No softmax is needed — do NOT add it.

Your training loop is correct as written.

You said:

what's the difference bewtween thse two lines?

```
# do backward step to tweak the weights  
loss.backward()
```

```
self.optimizer.step()
```

ChatGPT said:

These two lines do *different* parts of the training process. They're both required, and they don't do the same thing.

---

## 1. `loss.backward()`

This computes the gradients of the loss with respect to every parameter in your model.

- PyTorch builds a computational graph during the forward pass
- Calling `loss.backward()`:
  - Traverses this graph
  - Computes  $\partial \text{loss} / \partial \text{weights}$  for *every weight*
  - Stores these gradients in `param.grad` for each parameter

It does NOT update the weights.

It only *calculates* how much they should change.

---

## 2. `self.optimizer.step()`

This uses the stored gradients to update the model weights.

- If using SGD, it performs:
  - $w = w - \eta \cdot \text{grad}$
  - $w = w - \eta \cdot \text{grad}$
- If using Adam:
  - It uses running averages, momentum terms, bias correction, etc.
  - But still: it updates weights using the gradients stored by `.backward()`

After `.step()`, the weights are officially changed.

---

## Summary Table

| Line                          | What it does                                           | Does it change weights?                                                                   |
|-------------------------------|--------------------------------------------------------|-------------------------------------------------------------------------------------------|
| <code>loss.backward()</code>  | Computes gradients & stores them in <code>.grad</code> |  No  |
| <code>optimizer.step()</code> | Updates weights based on gradients                     |  Yes |

---

**This is why you need both**

Typical order:

1. zero\_grad() — clear old gradients
2. forward pass — compute predictions
3. loss = criterion(...)
4. loss.backward() — compute gradients
5. optimizer.step() — update weights

If you skip `backward()`: the step has no gradients → no learning

If you skip `step()`: gradients accumulate but weights never change → no learning

If you skip `zero_grad()`: gradients accumulate from previous batches → model blows up

---

If you want, I can draw a diagram or show an example printing out `.grad` before and after each step!

You said:

what libraries need to be installed here

```
from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import numpy as np

from abc import ABC, abstractmethod
from tensorflow.keras.datasets import mnist, cifar10
import random
import time
from torch.utils.data import TensorDataset, DataLoader, random_split, ConcatDataset
import torch

from mlp import MLP_Manager
from cnn import CNN_Manager
```

```
from model import Model
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
```

ChatGPT said:

Here is the complete list of external libraries you must install based on the imports you showed.

I'll group them by whether they are external pip installs or Python built-ins.

---

## ✓ External libraries you must install with pip

### 1. PyTorch

This covers:

```
import torch import torch.nn as nn import torch.optim as optim from
torch.utils.data import DataLoader, TensorDataset, random_split,
ConcatDataset
```

Install (CPU-only):

```
pip install torch
```

Or for GPU (CUDA 12 example):

```
pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu121
```

---

### 2. TensorFlow (because you use keras.datasets)

For:

```
from tensorflow.keras.datasets import mnist, cifar10
```

Install:

```
pip install tensorflow
```

If you do NOT want full TensorFlow, you can install only the dataset loader via `keras`:

```
pip install keras
```

But typically users install full TensorFlow.

---

### 3. NumPy

For:

```
import numpy as np
```

Install:

```
pip install numpy
```

---

## ■ Local project modules (no pip install needed)

These are files in your own project directory:

```
from model import Model from mlp import MLP_Manager from cnn import  
CNN_Manager
```

These must exist as:

```
model.py  
mlp.py  
cnn.py
```

in the same folder or in your PYTHONPATH.

---

## ■ Built-in Python modules (no installation needed)

These come with Python by default:

```
from abc import ABC, abstractmethod import random import time
```

No pip install required.

---

## ✓ Final pip install list

If you want everything working:

```
pip install torch tensorflow numpy
```

Optional if you want GPU acceleration:

```
pip install torch torchvision torchaudio --index-url  
https://download.pytorch.org/whl/cu121
```

---

If you want, I can also help you verify your environment, CUDA version, or fix any import errors.

You said:

will this work?

```
from abc import ABC, abstractmethod  
from tensorflow.keras.datasets import mnist, cifar10  
import random  
import time  
from torch.utils.data import TensorDataset, DataLoader, random_split, ConcatDataset  
import torch  
import numpy as np
```

```
class Model(ABC):  
    @abstractmethod  
    def doCustomProcessing(self):  
        pass  
    def retrieveDataset(self, setType):  
        if setType == "mnist":  
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()  
        else:
```

```

(xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

# normalizing them. Converting pixel values between 0 - 255 to 0 - 1
xTrain = xTrain.astype('float32') / 255.0
xTest = xTest.astype('float32') / 255.0

return (xTrain, yTrain, xTest, yTest)

# initialize method
def __init__(self):
    (self.mnistXTrain, self.mnistYTrain, self.mnistXTest, self.mnistYTest) =
        self.retrieveDataset("mnist")
    (self.cifarXTrain, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
        self.retrieveDataset("cifar")

# all the parameters we can choose from
    self.parameters = {
        "learningRate": [0.01, 0.001, 0.0001],
        "batchSize": [32, 64, 128],
        "optimizer": ["sgd", "adam"],
        "dropoutRate": [0.2, 0.5]
    }

# trying to use the GPU if possible, depending on where we run it
self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# this allows us to randomly sample from each parameter
    def sampleParams(self):
        return {key: random.choice(values) for key, values in self.parameters.items()}

# create the tensors for each array, as well as TensorDatasets
    def createTensorsAndSplitting(self):
        self.mnistXTrain = torch.tensor(self.mnistXTrain, dtype=torch.float32)
        self.mnistXTest = torch.tensor(self.mnistXTest, dtype=torch.float32)
        self.mnistYTrain = torch.tensor(self.mnistYTrain, dtype=torch.long).squeeze()

```

```

        self.mnistYTest = torch.tensor(self.mnistYTest, dtype=torch.long).squeeze()

        self.cifarXTrain = torch.tensor(self.cifarXTrain, dtype=torch.float32)
        self.cifarXTest = torch.tensor(self.cifarXTest, dtype=torch.float32)
        self.cifarYTrain = torch.tensor(self.cifarYTrain, dtype=torch.long).squeeze()
        self.cifarYTest = torch.tensor(self.cifarYTest, dtype=torch.long).squeeze()

mnistTrain = TensorDataset(self.mnistXTrain, self.mnistYTrain)
cifarTrain = TensorDataset(self.cifarXTrain, self.cifarYTrain)

self.mnistTest = TensorDataset(self.mnistXTest, self.mnistYTest)
self.cifarTest = TensorDataset(self.cifarXTest, self.cifarYTest)

# after we create the tensors we can do this
self.mnistTrain, self.mnistValid = random_split(mnistTrain, [50000, 10000])

self.cifarTrain, self.cifarValid = random_split(cifarTrain, [45000, 5000])

# abstract method the subclasses implement
@abstractmethod
def doModelCreation(self, complexity, dataset, sampledParams):
    pass

# We need to do this each hypertuning because the batch size is a hyeprparameter,
# so we cneed to create batches dynamically
def createBatches(self, batchSize, dataset, doingFinalTrain):
    if(dataset == "mnist"):
        if(doingFinalTrain):
            mergedTrain = ConcatDataset([self.mnistTrain, self.mnistValid])
            self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
   True)
            self.testBatch = DataLoader(self.mnistTest, batch_size = batchSize)
            else:

```

```

self.trainBatch = DataLoader(self.mnistTrain, batch_size = batchSize, shuffle
                           = True)
self.testBatch = DataLoader(self.mnistValid, batch_size = batchSize)
else:
    if(doingFinalTrain):
        mergedTrain = ConcatDataset([self.cifarTrain, self.cifarValid])
self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
                           True)
self.testBatch = DataLoader(self.cifarTest, batch_size = batchSize)
else:
self.trainBatch = DataLoader(self.cifarTrain, batch_size = batchSize, shuffle =
                           True)
self.testBatch = DataLoader(self.cifarValid, batch_size = batchSize)

# this creates the model, calls the function for tuning, and does evaluation and
# returns the runtime.
def doFullTuning(self, dataset, modelName):
    complexities = ["low", "medium", "high"]
    complexityAccuracies = np.zeros(3)
    complexityParams = []

    # get the best complexity too
    for complexity in complexities:
        startTime = time.time()
        tunelIterations = 11
        finalParams = {}
        highestAccuracy = 0

        allTuneAccuracies = np.zeros(tunelIterations)

        # hypertune using random search with about 11 iterations
        for i in range(0, tunelIterations, 1):
            #get a random search sample
            sampledParams = self.sampleParams()

```

```

        # crate our model according to sampled parameters
        self.doModelCreation(complexity, dataset, sampledParams)

        # create the batches
        self.createBatches(sampledParams["batchSize"], dataset, False)

        # train and valuate this current model, get accuracy
        accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,
   False, self.device)

        # keep the parameter set with the highest scores
        if(accuracy > highestAccuracy):
            highestAccuracy = accuracy
            finalParams = sampledParams

        allTuneAccuracies[i] = accuracy

# get end time from start of tuning to end of final model's training
endTime = time.time()
numMinutes = (endTime - startTime) / 60
numSeconds = (endTime - startTime) % 60

meanAccForComplexity = np.mean(allTuneAccuracies)
deviationForComplexity = np.std(allTuneAccuracies)

print(f"Validation accuracy for {modelName} model with complexity of
{complexity} and dataset of {dataset} was {meanAccForComplexity} with standard
deviation of {deviationForComplexity}, while total elapsed time was {numMinutes}
minutes and {numSeconds} seconds. Additionally, the parameters chosen were:
{self.finalParams}")
complexityAccuracies = meanAccForComplexity
complexityParams.append(finalParams)

bestComplexityIndex, _ = np.argmax(complexityAccuracies, axis = 0)

```

```

        self.finalComplexity = complexities[bestComplexityIndex]
        self.trueFinalParams = finalParams[bestComplexityIndex]

    def doFinalEvaluation(self, dataset, modelName):
        # do model creation again with the final parameters
        self.doModelCreation(self.finalComplexity, dataset, self.trueFinalParams)

# create batches again, but this time pass in doingFinalTrain = True to concatenate
# train and validation sets together
        self.createBatches(self.trueFinalParams["batchSize"], dataset, True)
            # get accuracy to print
accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,
  True, self.device)
        print(f"Test accuracy for {modelName} model with complexity of
{self.finalComplexity} and dataset of {dataset} was {accuracy}. As a reminder, the
parameters chosen were: {self.trueFinalParams}")

    ChatGPT said:
You're very close, but as written this code will NOT run correctly.

    Here are the major problems and how to fix them.

```

## **✗ 1. You call `np.std()` — but NumPy has no function named `stdev`**

You want:

```
np.std(allTuneAccuracies)
```

Fix:

```
deviationForComplexity = np.std(allTuneAccuracies)
```

## **✗ 2. `self.finalParams` is never defined — but you print it**

You wrote:

```
print(... while total elapsed time was ... Additionally, the parameters  
chosen were: {self.finalParams}")
```

There is no `self.finalParams` attribute anywhere in the class.

You probably meant:

```
{finalParams}
```

or later use `complexityParams`.

---

## ✗ 3. These lines overwrite the array instead of assigning into it

You wrote:

```
complexityAccuracies = meanAccForComplexity
```

This destroys your array — now it's a float.

You meant:

```
complexityAccuracies[i] = meanAccForComplexity
```

But there is no index `i` at that loop level.

Correct structure:

```
for idx, complexity in enumerate(complexities): ...  
    complexityAccuracies[idx] = meanAccForComplexity  
    complexityParams.append(finalParams)
```

---

## ✗ 4. You incorrectly compute the best complexity

You wrote:

```
bestComplexityIndex, _ = np.argmax(complexityAccuracies, axis = 0)
```

Inside a 1D array `axis=0` is fine but `np.argmax()` returns a single integer, not a tuple.

Correct:

```
bestComplexityIndex = np.argmax(complexityAccuracies)
```

## ✖ 5. You assign the wrong variable for final params

You wrote:

```
self.trueFinalParams = finalParams[bestComplexityIndex]
```

But `finalParams` is a dict from the LAST LOOP ONLY.

You want to retrieve the params you saved earlier:

```
self.trueFinalParams = complexityParams[bestComplexityIndex]
```

## ✖ 6. CIFAR labels need flattening

This line works:

```
self.cifarYTrain = torch.tensor(self.cifarYTrain,  
                                 dtype=torch.long).squeeze()
```

But note: CIFAR labels have shape `(n, 1)` so this is correct — but if the batch size is 1 you may accidentally squeeze away the batch dimension.

Better:

```
self.cifarYTrain = torch.tensor(self.cifarYTrain,  
                                 dtype=torch.long).view(-1)
```

## ⚠ 7. Missing: moving tensors to GPU

You initialize:

```
self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

BUT your tensors remain on CPU unless moved in your model training code.

If your `trainAndEvaluate` does not do `.to(device)`, you need to update it.

---

## ✖ 8. `retrieveDataset("cifar")` doesn't work — should be "cifar10"

Right now this line fails:

```
(self.cifarXTrain, self.cifarYTrain, ...) = self.retrieveDataset("cifar")
```

Because your function checks:

```
if setType == "mnist": ... else: (xTrain, yTrain), (xTest, yTest) =  
    cifar10.load_data()
```

So "cifar" is ok because your logic treats it as "not mnist" — but that's dangerous.

Better:

```
elif setType == "cifar": ... else: raise ValueError("Unknown dataset!")
```

---

## 🔧 Here is a corrected version of the problematic part

```
complexityAccuracies = np.zeros(3) complexityParams = [] for idx,  
    complexity in enumerate(complexities): startTime = time.time()  
tuneIterations = 11 finalParams = {} highestAccuracy = 0 allTuneAccuracies  
= np.zeros(tuneIterations) for i in range(tuneIterations): sampledParams =  
    self.sampleParams() self.doModelCreation(complexity, dataset,  
    sampledParams) self.createBatches(sampledParams["batchSize"], dataset,  
    False) accuracy = self.currentModel.trainAndEvaluate( self.trainBatch,  
    self.testBatch, False, self.device ) if accuracy > highestAccuracy:  
    highestAccuracy = accuracy finalParams = sampledParams  
    allTuneAccuracies[i] = accuracy endTime = time.time() numMinutes =  
(endTime - startTime) / 60 numSeconds = (endTime - startTime) % 60 meanAcc
```

```

= np.mean(allTuneAccuracies) deviation = np.std(allTuneAccuracies) print(
f"Validation accuracy for {modelName} model with complexity {complexity} "
    f"and dataset {dataset}: mean={meanAcc}, std={deviation}, "
    f"time={numMinutes}m {numSeconds}s, params={finalParams}" )

complexityAccuracies[idx] = meanAcc complexityParams.append(finalParams)
bestComplexityIndex = np.argmax(complexityAccuracies) self.finalComplexity
    = complexities[bestComplexityIndex] self.trueFinalParams =
        complexityParams[bestComplexityIndex]

```

---

## Summary: Will your code work?

No. Not yet.

It has 5–6 critical bugs that will cause:

- runtime errors
- incorrect parameter selection
- incorrect accuracy tracking
- crashes during printing
- wrong complexity choice

If you want, I can fully rewrite the class in a clean, correct, well-structured version.

You said:

 \*\*4\. You incorrectly compute the best complexity\*\*

===== You wrote: python Copy code  
`bestComplexityIndex, \_ = np.argmax(complexityAccuracies, axis = 0)` Inside a 1D array `axis=0` is fine but `np.argmax()` returns a single integer, not a tuple. Correct: python Copy code  
`bestComplexityIndex = np.argmax(complexityAccuracies)`  
does it not return bot hthe index and the max value?

ChatGPT said:

No — `np.argmax()` does NOT return both the index and the value.

It only returns one thing: the index of the maximum value.

---

## Example

```
arr = np.array([10, 20, 15]) idx = np.argmax(arr) print(idx) # 1
```

That's all it returns.

---

## This will NOT work

```
bestIndex, bestValue = np.argmax(arr)
```

This errors because `np.argmax(arr)` returns just 1, not (1, 20).

---

## If you want both index and value, you must get the value yourself

```
idx = np.argmax(arr) value = arr[idx]
```

Or in one line:

```
idx, value = np.argmax(arr), arr[np.argmax(arr)]
```

---

## Why your original code fails

You wrote:

```
bestComplexityIndex, _ = np.argmax(complexityAccuracies, axis = 0)
```

Since `np.argmax(...)` returns a single integer, Python tries to unpack it into two variables, causing:

```
ValueError: cannot unpack non-iterable int object
```

---

If you want, I can help rewrite the selection code cleanly.

You said:

will this work?

```
from abc import ABC, abstractmethod
```

```

from tensorflow.keras.datasets import mnist, cifar10
import random
import time
from torch.utils.data import TensorDataset, DataLoader, random_split, ConcatDataset
import torch
import numpy as np

class Model(ABC):
    @abstractmethod
    def doCustomProcessing(self):
        pass
    def retrieveDataset(self, setType):
        if setType == "mnist":
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
        else:
            (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

        # normalizing them. Converting pixel values between 0 - 255 to 0 - 1
        xTrain = xTrain.astype('float32') / 255.0
        xTest = xTest.astype('float32') / 255.0

        return (xTrain, yTrain, xTest, yTest)

    # initialize method
    def __init__(self):
        (self.mnistXTrain, self.mnistYTrain, self.mnistXTest, self.mnistYTest) =
            self.retrieveDataset("mnist")
        (self.cifarXTrain, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
            self.retrieveDataset("cifar")

    # all the parameters we can choose from
    self.parameters = {
        "learningRate": [0.01, 0.001, 0.0001],
        "batchSize": [32, 64, 128],
    }

```

```

        "optimizer": ["sgd", "adam"],
        "dropoutRate": [0.2, 0.5]
    }

# trying to use the GPU if possible, depending on where we run it
self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# this allows us to randomly sample from each parameter
def sampleParams(self):
    return {key: random.choice(values) for key, values in self.parameters.items()}

# create the tensors for each array, as well as TensorDatasets
def createTensorsAndSplitting(self):
    self.mnistXTrain = torch.tensor(self.mnistXTrain, dtype=torch.float32)
    self.mnistXTest = torch.tensor(self.mnistXTest, dtype=torch.float32)
    self.mnistYTrain = torch.tensor(self.mnistYTrain, dtype=torch.long).squeeze()
    self.mnistYTest = torch.tensor(self.mnistYTest, dtype=torch.long).squeeze()

    self.cifarXTrain = torch.tensor(self.cifarXTrain, dtype=torch.float32)
    self.cifarXTest = torch.tensor(self.cifarXTest, dtype=torch.float32)
    self.cifarYTrain = torch.tensor(self.cifarYTrain, dtype=torch.long).squeeze()
    self.cifarYTest = torch.tensor(self.cifarYTest, dtype=torch.long).squeeze()

mnistTrain = TensorDataset(self.mnistXTrain, self.mnistYTrain)
cifarTrain = TensorDataset(self.cifarXTrain, self.cifarYTrain)

self.mnistTest = TensorDataset(self.mnistXTest, self.mnistYTest)
self.cifarTest = TensorDataset(self.cifarXTest, self.cifarYTest)

# after we create the tensors we can do this
self.mnistTrain, self.mnistValid = random_split(mnistTrain, [50000, 10000])

self.cifarTrain, self.cifarValid = random_split(cifarTrain, [45000, 5000])

```

```

# abstract method the subclasses implement
    @abstractmethod

def doModelCreation(self, complexity, dataset, sampledParams):
    pass

# We need to do this each hypertuning because the batch size is a hyeprparameter,
# so we cneed to create batches dynamically
def createBatches(self, batchSize, dataset, doingFinalTrain):
    if(dataset == "mnist"):
        if(doingFinalTrain):
            mergedTrain = ConcatDataset([self.mnistTrain, self.mnistValid])
            self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
                True)
            self.testBatch = DataLoader(self.mnistTest, batch_size = batchSize)
            else:
                self.trainBatch = DataLoader(self.mnistTrain, batch_size = batchSize, shuffle
                    = True)
                self.testBatch = DataLoader(self.mnistValid, batch_size = batchSize)
                else:
                    if(doingFinalTrain):
                        mergedTrain = ConcatDataset([self.cifarTrain, self.cifarValid])
                        self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
                            True)
                        self.testBatch = DataLoader(self.cifarTest, batch_size = batchSize)
                        else:
                            self.trainBatch = DataLoader(self.cifarTrain, batch_size = batchSize, shuffle =
                                True)
                            self.testBatch = DataLoader(self.cifarValid, batch_size = batchSize)

# this creates the model, calls the function for tuning, and does evaluation and
# returns the rutnime.
def doFullTuning(self, dataset, modelName):
    complexities = ["low", "medium", "high"]
    complexityAccuracies = np.zeros(3)
    complexityParams = [{} for _ in range(3)]

```

```

        # get the best complexity too
        for complexity in complexities:
            startTime = time.time()
            tunelIterations = 11
            finalParams = {}
            highestAccuracy = 0

            allTuneAccuracies = np.zeros(tunelIterations)

            # hypertune using random search with about 11 iterations
            for i in range(0, tunelIterations, 1):
                #get a random search sample
                sampledParams = self.sampleParams()

                # crate our model according to sampled parameters
                self.doModelCreation(complexity, dataset, sampledParams)

                # create the batches
                self.createBatches(sampledParams["batchSize"], dataset, False)

                # train and valuate this current model, get accuracy
                accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,
   False, self.device)

                # keep the parameter set with the highest scores
                if(accuracy > highestAccuracy):
                    highestAccuracy = accuracy
                    finalParams = sampledParams

                allTuneAccuracies[i] = accuracy

        # get end time from start of tuning to end of final model's training
        endTime = time.time()
    
```

```

        numMinutes = (endTime - startTime) / 60
        numSeconds = (endTime - startTime) % 60

        meanAccForComplexity = np.mean(allTuneAccuracies)
        deviationForComplexity = np.std(allTuneAccuracies)

        print(f"Validation accuracy for {modelName} model with complexity of
{complexity} and dataset of {dataset} was {meanAccForComplexity} with standard
deviation of {deviationForComplexity}, while total elapsed time was {numMinutes}
minutes and {numSeconds} seconds. Additionally, the parameters chosen were:
{finalParams}")

        complexityAccuracies[i] = meanAccForComplexity
        complexityParams[i] = finalParams

bestComplexityIndex = np.argmax(complexityAccuracies, axis = 0)

        self.finalComplexity = complexities[bestComplexityIndex]
        self.trueFinalParams = complexityParams[bestComplexityIndex]

def doFinalEvaluation(self, dataset, modelName):
    # do model creation again with the final parameters
    self.doModelCreation(self.finalComplexity, dataset, self.trueFinalParams)

# create batches again, but this time pass in doingFinalTrain = True to concatenate
    train and validation sets together
    self.createBatches(self.trueFinalParams["batchSize"], dataset, True)
        # get accuracy to print
accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,
    True, self.device)
    print(f"Test accuracy for {modelName} model with complexity of
{self.finalComplexity} and dataset of {dataset} was {accuracy}. As a reminder, the
parameters chosen were: {self.trueFinalParams}")

```

ChatGPT said:

You're very close, but a few bugs will prevent this from working correctly.

Here are the issues and how to fix them.

---

## ! CRITICAL BUGS

### ✓ Bug #1 — `complexityAccuracies[i]` is wrong

Inside the loop:

```
for complexity in complexities: ... complexityAccuracies[i] =  
    meanAccForComplexity
```

But `i` is from the inner loop (0–10), not the complexity index!

Meaning you're writing accuracy values into the wrong places.

### ✓ FIX:

Use an index for the outer loop:

```
for cIndex, complexity in enumerate(complexities): ...  
    complexityAccuracies[cIndex] = meanAccForComplexity  
    complexityParams[cIndex] = finalParams
```

---

### ! Bug #2 — CIFAR images need a channel dimension

MNIST shape is `(batch, 28, 28)` — PyTorch expects `(batch, 1, 28, 28)`

CIFAR shape is `(batch, 32, 32, 3)` — PyTorch expects `(batch, 3, 32, 32)`

Your code does not reorder or add channels.

This will break your CNN.

### ✓ FIX:

Inside `createTensorsAndSplitting()` add:

```
self.mnistXTrain = self.mnistXTrain.unsqueeze(1) # (N,1,28,28)
self.mnistXTest = self.mnistXTest.unsqueeze(1) self.cifarXTrain =
self.cifarXTrain.permute(0, 3, 1, 2) # (N,3,32,32) self.cifarXTest =
self.cifarXTest.permute(0, 3, 1, 2)
```

---

## ! Bug #3 — You never store the model in `self.currentModel`

Your code calls:

```
self.currentModel.trainAndEvaluate(...)
```

But `self.doModelCreation()` must assign:

```
self.currentModel = createdModel
```

If your subclass does not set `self.currentModel`, your code will crash.

---

## ! Bug #4 — `doModelCreation()` must move model to GPU

Right now you only move data to GPU. The model must be on the same device:

```
self.currentModel.to(self.device)
```

---

## ! Bug #5 — `mnist.load_data()` and `cifar10.load_data()` produce labels of shape `(N, 1)`

You fix this by `.squeeze()`, which is correct — but note that if the dataset ever contained `0`, `.squeeze()` won't remove only the last dimension.

This part is okay, just pointing it out.

---

## ! Bug #6 — Missing imports

You use `np.stdev()` but correct function is:

```
np.std()
```

You fixed it in the last version — good.

---

## ● Corrected Version of the Problem Sections

### ✓ Fixing the complexity loop:

```
for cIndex, complexity in enumerate(complexities): startTime = time.time()
tuneIterations = 11 finalParams = {} highestAccuracy = 0 allTuneAccuracies
= np.zeros(tuneIterations) for i in range(tuneIterations): sampledParams =
    self.sampleParams() self.doModelCreation(complexity, dataset,
        sampledParams) self.currentModel.to(self.device)
self.createBatches(sampledParams["batchSize"], dataset, False) accuracy =
    self.currentModel.trainAndEvaluate( self.trainBatch, self.testBatch,
    False, self.device ) if accuracy > highestAccuracy: highestAccuracy =
        accuracy finalParams = sampledParams allTuneAccuracies[i] = accuracy
meanAcc = np.mean(allTuneAccuracies) stdAcc = np.std(allTuneAccuracies)
print(f"...") complexityAccuracies[cIndex] = meanAcc
complexityParams[cIndex] = finalParams
```

---

### ★ Answer: Will your code work?

**After the 6 fixes above — YES, it will work.**

Right now, it will NOT work due to:

- incorrect indexing
  - incorrect tensor image shape
  - missing `.to(device)` on model
  - missing `self.currentModel` assignment
  - CIFAR channel issue
  - minor statistical & print bugs
-

If you want, I can also review your `MLP_Manager` and `CNN_Manager` files to ensure they match this structure correctly.

You said:

how about this?

```
from abc import ABC, abstractmethod
from tensorflow.keras.datasets import mnist, cifar10
import random
import time
from torch.utils.data import TensorDataset, DataLoader, random_split, ConcatDataset
import torch
import numpy as np

class Model(ABC):
    @abstractmethod
    def doCustomProcessing(self):
        pass
    def retrieveDataset(self, setType):
        if setType == "mnist":
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
        else:
            (xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

        # normalizing them. Converting pixel values between 0 - 255 to 0 - 1
        xTrain = xTrain.astype('float32') / 255.0
        xTest = xTest.astype('float32') / 255.0

        return (xTrain, yTrain, xTest, yTest)

    # initialize method
    def __init__(self):
        (self.mnistXTrain, self.mnistYTrain, self.mnistXTest, self.mnistYTest) =
            self.retrieveDataset("mnist")
```

```

(self.cifarXTrain, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
    self.retrieveDataset("cifar")

# all the parameters we can choose from
self.parameters = {
    "learningRate": [0.01, 0.001, 0.0001],
    "batchSize": [32, 64, 128],
    "optimizer": ["sgd", "adam"],
    "dropoutRate": [0.2, 0.5]
}

# trying to use the GPU if possible, depending on where we run it
self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# this allows us to randomly sample from each parameter
def sampleParams(self):
    return {key: random.choice(values) for key, values in self.parameters.items()}

# create the tensors for each array, as well as TensorDatasets
def createTensorsAndSplitting(self):
    self.mnistXTrain = torch.tensor(self.mnistXTrain, dtype=torch.float32)
    self.mnistXTest = torch.tensor(self.mnistXTest, dtype=torch.float32)
    self.mnistYTrain = torch.tensor(self.mnistYTrain, dtype=torch.long).squeeze()
    self.mnistYTest = torch.tensor(self.mnistYTest, dtype=torch.long).squeeze()

    self.cifarXTrain = torch.tensor(self.cifarXTrain, dtype=torch.float32)
    self.cifarXTest = torch.tensor(self.cifarXTest, dtype=torch.float32)
    self.cifarYTrain = torch.tensor(self.cifarYTrain, dtype=torch.long).squeeze()
    self.cifarYTest = torch.tensor(self.cifarYTest, dtype=torch.long).squeeze()

mnistTrain = TensorDataset(self.mnistXTrain, self.mnistYTrain)
cifarTrain = TensorDataset(self.cifarXTrain, self.cifarYTrain)

self.mnistTest = TensorDataset(self.mnistXTest, self.mnistYTest)

```

```

        self.cifarTest = TensorDataset(self.cifarXTest, self.cifarYTest)

            # after we create the tensors we can do this
        self.mnistTrain, self.mnistValid = random_split(mnistTrain, [50000, 10000])

        self.cifarTrain, self.cifarValid = random_split(cifarTrain, [45000, 5000])

            # abstract method the subclasses implement
            @abstractmethod
        def doModelCreation(self, complexity, dataset, sampledParams):
            pass

# We need to do this each hypertuning because the batch size is a hyeprparameter,
    so we cneed to create batches dynamically
    def createBatches(self, batchSize, dataset, doingFinalTrain):
        if(dataset == "mnist"):
            if(doingFinalTrain):
                mergedTrain = ConcatDataset([self.mnistTrain, self.mnistValid])
            self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
                True)
            self.testBatch = DataLoader(self.mnistTest, batch_size = batchSize)
            else:
                self.trainBatch = DataLoader(self.mnistTrain, batch_size = batchSize, shuffle
                    = True)
                self.testBatch = DataLoader(self.mnistValid, batch_size = batchSize)
            else:
                if(doingFinalTrain):
                    mergedTrain = ConcatDataset([self.cifarTrain, self.cifarValid])
                self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
                    True)
                self.testBatch = DataLoader(self.cifarTest, batch_size = batchSize)
            else:
                self.trainBatch = DataLoader(self.cifarTrain, batch_size = batchSize, shuffle =
                    True)
                self.testBatch = DataLoader(self.cifarValid, batch_size = batchSize)

```

```

# this creates the model, calls the function for tuning, and does evaluation and
# returns the rutnime.

def doFullTuning(self, dataset, modelName):
    complexities = ["low", "medium", "high"]
    complexityAccuracies = np.zeros(3)
    complexityParams = [{} for _ in range(3)]

        # get the best complexity too
    for complexity in complexities:
        startTime = time.time()
        tunelIterations = 11
        finalParams = {}
        highestAccuracy = 0

        allTuneAccuracies = np.zeros(tunelIterations)

        # hypertune using random search with about 11 iterations
        for i in range(0, tunelIterations, 1):
            #get a random search sample
            sampledParams = self.sampleParams()

            # crate our model according to sampled parameters
            self.doModelCreation(complexity, dataset, sampledParams)

            # create the batches
            self.createBatches(sampledParams["batchSize"], dataset, False)

            # train and valuate this current model, get accuracy
            accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,
                False, self.device)

            # keep the parameter set with the highest scores
            if(accuracy > highestAccuracy):
                highestAccuracy = accuracy

```

```

        finalParams = sampledParams

        allTuneAccuracies[i] = accuracy

# get end time from start of tuning to end of final model's training
        endTime = time.time()
        numMinutes = (endTime - startTime) / 60
        numSeconds = (endTime - startTime) % 60

meanAccForComplexity = np.mean(allTuneAccuracies)
deviationForComplexity = np.std(allTuneAccuracies)

print(f"Validation accuracy for {modelName} model with complexity of
{complexity} and dataset of {dataset} was {meanAccForComplexity} with standard
deviation of {deviationForComplexity}, while total elapsed time was {numMinutes}
minutes and {numSeconds} seconds. Additionally, the parameters chosen were:
{finalParams}")

complexityAccuracies[complexity] = meanAccForComplexity
complexityParams[complexity] = finalParams

bestComplexityIndex = np.argmax(complexityAccuracies, axis = 0)

self.finalComplexity = complexities[bestComplexityIndex]
self.trueFinalParams = complexityParams[bestComplexityIndex]

def doFinalEvaluation(self, dataset, modelName):
    # do model creation again with the final parameters
    self.doModelCreation(self.finalComplexity, dataset, self.trueFinalParams)

# create batches again, but this time pass in doingFinalTrain = True to concatenate
    train and validation sets together
    self.createBatches(self.trueFinalParams["batchSize"], dataset, True)
        # get accuracy to print

```

```
accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,  
  True, self.device)  
print(f"Test accuracy for {modelName} model with complexity of  
{self.finalComplexity} and dataset of {dataset} was {accuracy}. As a reminder, the  
parameters chosen were: {self.trueFinalParams}")
```

ChatGPT said:

Here's a clean breakdown of what you need to install for the code you just posted, and where potential issues are.

---

## Python Libraries Required

Your code references the following external libraries (i.e., things you must install with pip):

### ### Must install

| Library                                            | Reason                                         |
|----------------------------------------------------|------------------------------------------------|
| torch                                              | PyTorch tensors, DataLoader, training          |
| tensorflow                                         | For tensorflow.keras.datasets MNIST + CIFAR-10 |
| numpy                                              | Arrays + mean/std                              |
| absl-py ( <i>maybe already installed with TF</i> ) | Comes with TensorFlow                          |
| mlp ( <i>local file</i> )                          | Your custom module ( <code>mlp.py</code> )     |
| cnn ( <i>local file</i> )                          | Your custom module ( <code>cnn.py</code> )     |
| model ( <i>local file</i> )                        | Your custom module ( <code>model.py</code> )   |

Install command:

```
pip install torch tensorflow numpy
```

That's it — the rest (like `random`, `time`, `abc`) are part of Python.

---



## Important Problems in Your Code

Here are issues you'll want to fix:

---

### ✗ 1. You're importing TensorFlow dataset loaders AND using PyTorch dataset loading

This:

```
from tensorflow.keras.datasets import mnist, cifar10
```

means MNIST/CIFAR are loaded as NumPy arrays from TF.

You later convert them manually into PyTorch tensors:

```
self.mnistXTrain = torch.tensor(...)
```

This is *fine*, but you do not need TensorFlow at all unless you specifically want its dataset loader.

### ✓ PyTorch has MNIST / CIFAR loaders built-in:

```
import torchvision.datasets as datasets import torchvision.transforms as transforms
```

If you want to simplify, I can rewrite your loader in PyTorch.

---

### ✗ 2. You call `random_split(mnistTrain, [50000, 10000])`

This only works if your MNIST train set actually has exactly 60,000 samples *after conversion*.

But CIFAR is 50,000 → your code tries:

```
random_split(cifarTrain, [45000, 5000])
```

That's fine.

MNIST is 60k → also fine.

But if you ever change datasets, this breaks.

---

### ✗ 3. You're indexing arrays incorrectly

You wrote:

```
complexityAccuracies[complexity] = ...
```

But `complexityAccuracies` is a NumPy array of size 3, and `complexity` is a string ("low", "medium", "high").

Meaning:

#### ✗ This line will crash:

```
complexityAccuracies[complexity] = meanAccForComplexity
```

You cannot do:

```
arr["medium"]
```

You must map complexities → indices:

```
for idx, complexity in enumerate(complexities): complexityAccuracies[idx] = ... complexityParams[idx] = ...
```

---

### ✗ 4. CIFAR10 labels from TF load as shape (50000, 1)

That's why you had to `.squeeze()`.

Fine — but if a dataset ever has labels of shape (n,) you'll break.

---

## 👉 Summary

## You need to install only:

```
pip install torch tensorflow numpy
```

Everything else is built-in or your own modules.

---

If you want, I can:

- ✓ rewrite your dataset loader using only PyTorch
  - ✓ fix the complexity indexing bug
  - ✓ simplify your hyperparameter tuning loop
  - ✓ check your MLP and CNN classes too

Just tell me!

You said:

how about this, any errors?

```
from abc import ABC, abstractmethod
from tensorflow.keras.datasets import mnist, cifar10
import random
import time
from torch.utils.data import TensorDataset, DataLoader, random_split, ConcatDataset
import torch
import numpy as np
```

```
class Model(ABC):
    @abstractmethod
    def doCustomProcessing(self):
        pass
    def retrieveDataset(self, setType):
        if setType == "mnist":
            (xTrain, yTrain), (xTest, yTest) = mnist.load_data()
        else:
```

```

(xTrain, yTrain), (xTest, yTest) = cifar10.load_data()

# normalizing them. Converting pixel values between 0 - 255 to 0 - 1
xTrain = xTrain.astype('float32') / 255.0
xTest = xTest.astype('float32') / 255.0

return (xTrain, yTrain, xTest, yTest)

# initialize method
def __init__(self):
    (self.mnistXTrain, self.mnistYTrain, self.mnistXTest, self.mnistYTest) =
        self.retrieveDataset("mnist")
    (self.cifarXTrain, self.cifarYTrain, self.cifarXTest, self.cifarYTest) =
        self.retrieveDataset("cifar")

# all the parameters we can choose from
    self.parameters = {
        "learningRate": [0.01, 0.001, 0.0001],
        "batchSize": [32, 64, 128],
        "optimizer": ["sgd", "adam"],
        "dropoutRate": [0.2, 0.5]
    }

# trying to use the GPU if possible, depending on where we run it
self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# this allows us to randomly sample from each parameter
    def sampleParams(self):
        return {key: random.choice(values) for key, values in self.parameters.items()}

# create the tensors for each array, as well as TensorDatasets
    def createTensorsAndSplitting(self):
        self.mnistXTrain = torch.tensor(self.mnistXTrain, dtype=torch.float32)
        self.mnistXTest = torch.tensor(self.mnistXTest, dtype=torch.float32)
        self.mnistYTrain = torch.tensor(self.mnistYTrain, dtype=torch.long).squeeze()

```

```

self.mnistYTest = torch.tensor(self.mnistYTest, dtype=torch.long).squeeze()

    self.cifarXTrain = torch.tensor(self.cifarXTrain, dtype=torch.float32)
    self.cifarXTest = torch.tensor(self.cifarXTest, dtype=torch.float32)
self.cifarYTrain = torch.tensor(self.cifarYTrain, dtype=torch.long).squeeze()
    self.cifarYTest = torch.tensor(self.cifarYTest, dtype=torch.long).squeeze()

mnistTrain = TensorDataset(self.mnistXTrain, self.mnistYTrain)
cifarTrain = TensorDataset(self.cifarXTrain, self.cifarYTrain)

self.mnistTest = TensorDataset(self.mnistXTest, self.mnistYTest)
    self.cifarTest = TensorDataset(self.cifarXTest, self.cifarYTest)

        # after we create the tensors we can do this
self.mnistTrain, self.mnistValid = random_split(mnistTrain, [50000, 10000])

self.cifarTrain, self.cifarValid = random_split(cifarTrain, [45000, 5000])

        # abstract method the subclasses implement
        @abstractmethod
def doModelCreation(self, complexity, dataset, sampledParams):
        pass

# We need to do this each hypertuning because the batch size is a hyeprparameter,
        so we cneed to create batches dynamically
def createBatches(self, batchSize, dataset, doingFinalTrain):
        if(dataset == "mnist"):
            if(doingFinalTrain):
                mergedTrain = ConcatDataset([self.mnistTrain, self.mnistValid])
self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
True)
        self.testBatch = DataLoader(self.mnistTest, batch_size = batchSize)
        else:

```

```

self.trainBatch = DataLoader(self.mnistTrain, batch_size = batchSize, shuffle
                           = True)
self.testBatch = DataLoader(self.mnistValid, batch_size = batchSize)
else:
    if(doingFinalTrain):
        mergedTrain = ConcatDataset([self.cifarTrain, self.cifarValid])
self.trainBatch = DataLoader(mergedTrain, batch_size = batchSize, shuffle =
                           True)
self.testBatch = DataLoader(self.cifarTest, batch_size = batchSize)
else:
self.trainBatch = DataLoader(self.cifarTrain, batch_size = batchSize, shuffle =
                           True)
self.testBatch = DataLoader(self.cifarValid, batch_size = batchSize)

# this creates the model, calls the function for tuning, and does evaluation and
# returns the runtime.

def doFullTuning(self, dataset, modelName):
    complexities = ["low", "medium", "high"]
    complexityAccuracies = np.zeros(3)
    complexityParams = [{} for _ in range(3)]

    # get the best complexity too
    for complexityIndex, complexity in complexities:
        startTime = time.time()
        tunelIterations = 11
        finalParams = {}
        highestAccuracy = 0

        allTuneAccuracies = np.zeros(tunelIterations)

        # hypertune using random search with about 11 iterations
        for i in range(0, tunelIterations, 1):
            #get a random search sample
            sampledParams = self.sampleParams()

```

```

        # crate our model according to sampled parameters
        self.doModelCreation(complexity, dataset, sampledParams)

        # create the batches
        self.createBatches(sampledParams["batchSize"], dataset, False)

        # train and valuate this current model, get accuracy
        accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,
   False, self.device)

        # keep the parameter set with the highest scores
        if(accuracy > highestAccuracy):
            highestAccuracy = accuracy
            finalParams = sampledParams

        allTuneAccuracies[i] = accuracy

# get end time from start of tuning to end of final model's training
endTime = time.time()
numMinutes = (endTime - startTime) / 60
numSeconds = (endTime - startTime) % 60

meanAccForComplexity = np.mean(allTuneAccuracies)
deviationForComplexity = np.std(allTuneAccuracies)

print(f"Validation accuracy for {modelName} model with complexity of
{complexity} and dataset of {dataset} was {meanAccForComplexity} with standard
deviation of {deviationForComplexity}, while total elapsed time was {numMinutes}
minutes and {numSeconds} seconds. Additionally, the parameters chosen were:
{finalParams}")

complexityAccuracies[complexityIndex] = meanAccForComplexity
complexityParams[complexityIndex] = finalParams

bestComplexityIndex = np.argmax(complexityAccuracies, axis = 0)

```

```

        self.finalComplexity = complexities[bestComplexityIndex]
        self.trueFinalParams = complexityParams[bestComplexityIndex]

    def doFinalEvaluation(self, dataset, modelName):
        # do model creation again with the final parameters
        self.doModelCreation(self.finalComplexity, dataset, self.trueFinalParams)

# create batches again, but this time pass in doingFinalTrain = True to concatenate
# train and validation sets together
        self.createBatches(self.trueFinalParams["batchSize"], dataset, True)
            # get accuracy to print
accuracy = self.currentModel.trainAndEvaluate(self.trainBatch, self.testBatch,
  True, self.device)
        print(f"Test accuracy for {modelName} model with complexity of
{self.finalComplexity} and dataset of {dataset} was {accuracy}. As a reminder, the
parameters chosen were: {self.trueFinalParams}")

```

**ChatGPT said:**

Here is a complete, precise review of your updated code — including all errors that will cause crashes, logic bugs, and small issues that will bite you later.

---

## ! Major Error #1 — Wrong for-loop:

You wrote:

```
for complexityIndex, complexity in complexities:
```

This is invalid Python.

complexities is:

```
["low", "medium", "high"]
```

Iterating over it yields one string at a time, not index + value.

This will raise:

```
ValueError: too many values to unpack (expected 2)
```

### ✓ Fix:

Use `enumerate`:

```
for complexityIndex, complexity in enumerate(complexities):
```

---

## ! Major Error #2 — CIFAR10 shape mismatch for splitting

CIFAR-10 labels loaded from TensorFlow have shape:

```
(50000, 1)
```

But splitting like this:

```
self.cifarTrain, self.cifarValid = random_split(cifarTrain, [45000, 5000])
```

is fine ONLY IF CIFAR training set has exactly 50,000 examples.

Just be aware this will break if you ever use a dataset size that differs.

---

## ! Major Error #3 — MNIST shape is wrong for CNNs

MNIST from TensorFlow loads as:

```
(60000, 28, 28) — missing channel dimension.
```

CNN code expects:

```
(N, 1, 28, 28)
```

You need:

```
self.mnistXTrain = self.mnistXTrain.unsqueeze(1) # add channel dim
```

Same for test.

---

## ! Major Error #4 — CIFAR images need channel reorder

TensorFlow CIFAR is loaded as:

```
(N, 32, 32, 3) ← channels last
```

PyTorch expects:

```
(N, 3, 32, 32) ← channels first
```

Missing transformation:

```
self.cifarXTrain = self.cifarXTrain.permute(0, 3, 1, 2) self.cifarXTest =  
    self.cifarXTest.permute(0, 3, 1, 2)
```

Without this, your CNN will crash.

---

## ! Major Error #5 — Accuracy accumulation logic is correct but final model creation missing

During tuning, you keep:

```
finalParams = sampledParams
```

But after tuning each complexity, you do NOT train a final model using `finalParams` for that complexity.

This is okay if intended, but note:

- You record mean accuracy of all runs, not the best model.
- `finalParams` does not correspond to that mean.

Just making sure you're aware.

---

## ! Major Error #6 — Misleading time display

You compute:

```
numMinutes = (endTime - startTime) / 60 numSeconds = (endTime - startTime)  
            % 60
```

This prints something like:

```
2.73 minutes and 14.52 seconds
```

Which doesn't make sense (because both are computed from the total time).

**Suggest:**

```
elapsed = endTime - startTime minutes = int(elapsed // 60) seconds =  
        elapsed % 60
```

---

## ! Major Error #7 — doCustomProcessing is unused

Not harmful but unclear why the abstract method exists.

---

## ! Major Error #8 — TensorDataset order might matter

MNIST inputs are shape:

```
(60000, 28, 28)
```

and NOT reshaped to:

```
(60000, 784)
```

If your MLP expects flat vectors, you must add:

```
self.mnistXTrain = self.mnistXTrain.reshape(-1, 28*28) self.mnistXTest =  
    self.mnistXTest.reshape(-1, 28*28)
```

So depending on architecture, this might break.

---

## ! Potential Issue — Device handling missing

You set:

```
self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

But you never move the model to device inside `doModelCreation()`.

Example fix:

```
self.currentModel.to(self.device)
```

## ✓ Summary of Needed Fixes (copy/paste version)

### Fix the for loop:

```
for complexityIndex, complexity in enumerate(complexities):
```

### Fix MNIST for CNNs:

```
self.mnistXTrain = self.mnistXTrain.unsqueeze(1) self.mnistXTest =  
    self.mnistXTest.unsqueeze(1)
```

### Fix CIFAR channel order:

```
self.cifarXTrain = self.cifarXTrain.permute(0, 3, 1, 2) self.cifarXTest =  
    self.cifarXTest.permute(0, 3, 1, 2)
```

### Fix time formatting:

```
elapsed = endTime - startTime minutes = int(elapsed // 60) seconds =  
    elapsed % 60
```

### Move model to GPU:

Inside `doModelCreation` (after creating model):

```
self.currentModel.to(self.device)
```

---

If you'd like, I can:

Automatically patch all fixes into your full code

or

Give you a polished, corrected version ready to run

Just tell me.

You said:

why did this adam optimizer perform better than SGD for MLP and CNNs?

ChatGPT said:

The reason Adam outperforms SGD in most MLPs and CNNs comes down to how the two optimizers update weights and handle the challenges of modern deep learning:

---

## 1. SGD (Stochastic Gradient Descent)

- Update rule:  
 $\theta = \theta - lr * \nabla L(\theta)$   
where `lr` is the learning rate.
- Pros:
  - Simple, memory-efficient.
  - Works well if you carefully tune learning rate and decay schedule.
- Cons:
  - Uses the same learning rate for all parameters.
  - Sensitive to gradients that are sparse or have varying magnitudes.
  - Converges slower if the surface has ravines or steep slopes (common in deep networks).
  - Can oscillate along steep directions, especially in deep CNNs or MLPs with many layers.

---

## 2. Adam (Adaptive Moment Estimation)

- Update rule:

Adam computes:

1. First moment: running average of gradients ( $\hat{m}$ )
2. Second moment: running average of squared gradients ( $\hat{v}$ )

Then updates:

- $\theta = \theta - lr * \hat{m} / (\sqrt{\hat{v}} + \epsilon)$
  - Key advantages over SGD:
    1. Adaptive learning rate per parameter: each weight gets its own step size.
      - Helps when gradients differ in scale (common in CNNs, especially with ReLU activations).
    2. Momentum built-in: smooths updates, reduces oscillations in narrow valleys.
    3. Bias correction: works better in early training steps when  $\hat{m}$  and  $\hat{v}$  are small.
    4. Less hyperparameter tuning required: learning rate can often be used “as-is” (0.001 typical).
- 

### 3. Why Adam beats SGD in your experiments

- MLPs:
    - Often have flat regions or plateaus in the loss landscape.
    - SGD can get “stuck” or move very slowly without careful LR scheduling.
    - Adam adapts step size dynamically and escapes plateaus faster.
  - CNNs:
    - Different layers have gradients of very different magnitudes (especially deeper layers).
    - SGD with a single global LR struggles.
    - Adam automatically scales updates per parameter, so all layers learn efficiently.
  - General observation:  
In most small to medium datasets (MNIST, CIFAR-10) Adam converges faster and often reaches a slightly better final validation accuracy, because it navigates the loss landscape more efficiently.
-

In short:

Adam combines the benefits of momentum + adaptive learning rate, whereas vanilla SGD has neither by default. That makes Adam more robust and faster for deep networks like MLPs and CNNs.

---

If you want, I can also explain why sometimes SGD can eventually beat Adam, especially with large datasets and very long training, and how learning rate schedules play a role. Do you want me to go into that?

ChatGPT can make mistakes. Check important info.