Pytorch basics

Adapted from Day 3 Lab 1

by yours truly, farah

Table of contents

why Pytorch?

- Dynamic, changes occur during run-time
- Flexible for fast prototyping
- Often used in research experimentation

- Tensorflow is for old people
- The only reason I would ever use tensorflow:

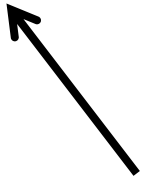
my GOAT Andrew

Pro tip: Use Andrew Ng's course on Neural Networks and Deep Learning offered to you on Coursera

PyTorch is an open-source deep learning framework that allows us to build and train neural networks using:

tensors and **automatic differentiation**.

for data



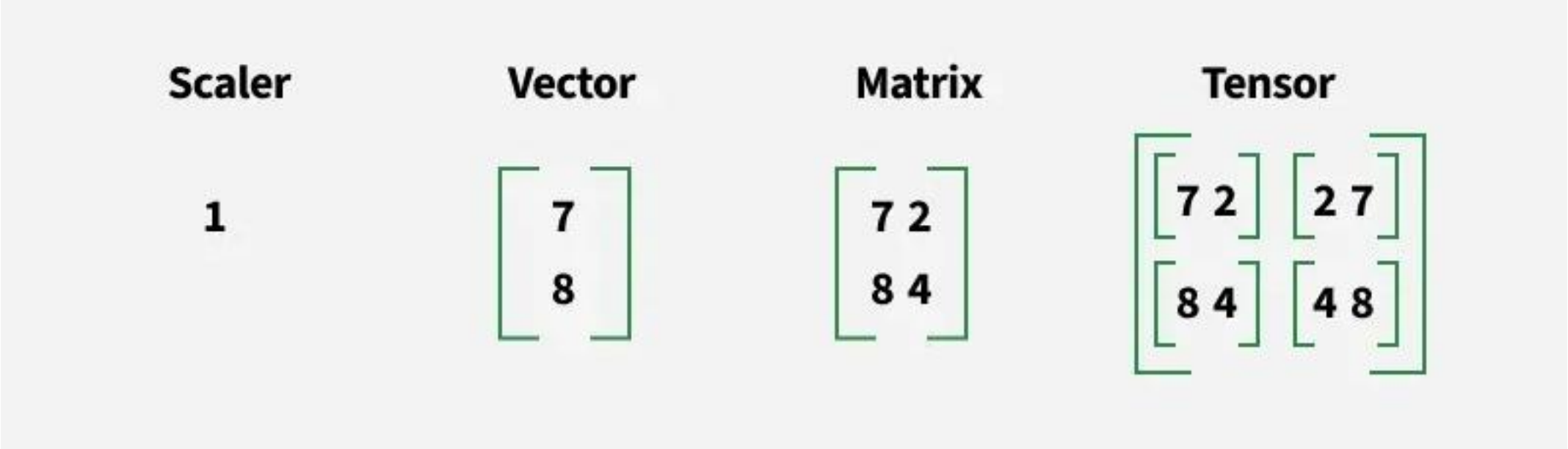
for training (gradient descent)

what even are tensors?

- Tensors are a way of storing the input data
- **In most cases**, they are multi-dimensional matrices
- They contain elements of a SINGLE DATA TYPE, just like:
  - torch.int32
  - torch.float64

- Why tensors?
  - Generalize data of all dimensions (0D, 1D, 2D… 6D)
  - They run on both CPUs and GPUs

what even are tensors?



question: how many tensors are present in the image above?

answer: 4, they're all tensors, just different dimensions

for more info: https://www.youtube.com/watch?v=L35fFDpwIM4

creating tensors

```python
import torch

# Create tensors
x = torch.tensor([1.0, 2.0, 3.0])
y = torch.randn(3)
print("x:", x)
print("y:", y)
```

```
x: tensor([1., 2., 3.])
y: tensor([0.3548, 0.3962, 0.4785])
```

tensor shapes

```
print("Shape of x:", x.shape)
```

```
Shape of x: torch.Size([3])
```

tensor operations

given:

```
a = torch.tensor([1.0, 2.0, 3.0])
b = torch.tensor([4.0, 5.0, 6.0])
```

elementwise operations:

```
print("Addition:", a + b)
```

```
Addition: tensor([5., 7., 9.])
```

```
print("Multiplication:", a * b)
```

```
Multiplication: tensor([ 4., 10., 18.])
```

tensor operations

matrix multiplication:

```
A = torch.randn(2, 3)
B = torch.randn(3, 2)
C = A @ B
```

```
A: tensor([[ 0.0721, -0.9560, -1.2007],
        [-0.1852, -0.4110, -0.9032]])
```

×

```
B: tensor([[ 0.0721, -0.9560, -1.2007],
        [-0.1852, -0.4110, -0.9032]])
```

```
C: tensor([[-1.0382, -0.5640],
        [-0.0658,  0.5422]])

Matrix multiplication result shape: torch.Size([2, 2])
```

data representation

The way data is represented in tensors depends on whether it is:

structured

Tabular Data

columns = attributes for those observations

Player	Minutes	Points	Rebounds	Assists
A	41	20	6	5
B	30	29	7	6
C	22	7	7	2
D	26	3	3	9
E	20	19	8	0
F	9	6	14	14
G	14	22	8	3
I	22	36	0	9
J	34	8	1	3

Rows = observations

unstructured

structured data

- represented as a 2D tensor
- Shape: (batch_size, number_of_features)
- Commonly used for tasks like regression and classification

Tabular Data

columns = attributes for those observations

Rows = observations

Player	Minutes	Points	Rebounds	Assists
A	41	20	6	5
B	30	29	7	6
C	22	7	7	2
D	26	3	3	9
E	20	19	8	0
F	9	6	14	14
G	14	22	8	3
I	22	36	0	9
J	34	8	1	3

question: what is the tensor shape of the table on the right?

answer: (9, 5)

unstructured data

- represented as a 4D tensor
- Shape: (batch_size, channels, height, width)
- Channels represent color information:
  - Grayscale ⟶ 1 channel
  - RGB (Red, Green, Blue) ⟶ 3 channels

question: assume the image on the right is composed of 1024x1024 pixels, and we have 120 images in total, what is the tensor shape?

answer: (120, 3, 1024, 1024)

tensor shape manipulation

- Manipulating tensor shapes is essential in deep learning
- I want you to know these functions better than you know your own family:
  - .flatten()
  - .squeeze()
  - .unsqueeze()
  - .view()

.flatten()

- Flattens input by reshaping it into a 1D tensor
- Converts **any shape** to (batch_size, features)

.flatten()

```python
import torch

x = torch.randn(32, 3, 28, 28)
x_flat = x.flatten(start_dim=1)
print("Flatten:", x_flat.shape)
```

```
Flatten: torch.Size([32, 2352])
```

***note**: you can use both x.flatten() or torch.flatten(x)

.flatten()

torch.flatten

`torch.flatten(input, start_dim=0, end_dim=-1) → Tensor`

Flattens `input` by reshaping it into a one-dimensional tensor. If `start_dim` or `end_dim` are passed, only dimensions starting with `start_dim` and ending with `end_dim` are flattened. The order of elements in `input` is unchanged.

Unlike NumPy's flatten, which always copies input's data, this function may return the original object, a view, or copy. If no dimensions are flattened, then the original object `input` is returned. Otherwise, if input can be viewed as the flattened shape, then that view is returned. Finally, only if the input cannot be viewed as the flattened shape is input's data copied. See `torch.Tensor.view()` for details on when a view will be returned.

Note

Flattening a zero-dimensional tensor will return a one-dimensional view.

Parameters:

- **input** (*Tensor*) – the input tensor.
- **start_dim** (*int*) – the first dim to flatten
- **end_dim** (*int*) – the last dim to flatten

.flatten()

question: given a tensor shape of (32, 3, 28, 28), what is the output shape if we flatten the tensor with the start_dim = 0?

answer: 75264

question: with the same tensor, what if the start_dim = 2?

answer: (32, 3, 784)

.flatten() common enemies (1)

RuntimeError: mat1 and mat2 shapes cannot be multiplied

why does this happen?

- Fully connected layers expect 2D tensors: (batch_size, features)
- But if you forget to flatten, you might pass something like:

(batch_size, channels, height, width)

Fix: x.flatten(start_dim=1)

.flatten() common enemies (2)

RuntimeError: Expected input size (N, C) but got (N, d1, d2)

why does this happen?

Predictions are multi-dimensional (1D or 2D targets)

Fix: x.flatten(start_dim=1)

.squeeze()

- Removes dimensions with size 1
- For example, if `input` is of shape: $(A \times 1 \times B \times C \times 1 \times D)$ then `input.squeeze()` will be of shape: $(A \times B \times C \times D)$

.squeeze()

```
import torch

x = torch.randn(1, 3, 28, 28)
x_sq = x.squeeze()
print("Squeeze:", x_sq.shape)
```

```
Squeeze: torch.Size([3, 28, 28])
```

***note**: you can use both x.squeeze() or torch.squeeze(x)

.squeeze()

torch.squeeze

`torch.squeeze(input: Tensor, dim: Optional[Union[int, List[int]]]) → Tensor`

Returns a tensor with all specified dimensions of `input` of size *1* removed.

For example, if *input* is of shape: $(A \times 1 \times B \times C \times 1 \times D)$ then the *input.squeeze()* will be of shape: $(A \times B \times C \times D)$.

When `dim` is given, a squeeze operation is done only in the given dimension(s). If *input* is of shape: $(A \times 1 \times B)$, `squeeze(input, 0)` leaves the tensor unchanged, but `squeeze(input, 1)` will squeeze the tensor to the shape $(A \times B)$.

Note

The returned tensor shares the storage with the input tensor, so changing the contents of one will change the contents of the other.

Warning

If the tensor has a batch dimension of size 1, then *squeeze(input)* will also remove the batch dimension, which can lead to unexpected errors. Consider specifying only the dims you wish to be squeezed.

.squeeze()

question: given a tensor x of shape (1, 3, 28, 28), what is the output shape when we apply `x.squeeze()`?

answer: (3, 28, 28)

question: given a tensor x of shape (2, 1, 2, 1, 2, 1), what is the output shape when we apply `x.squeeze()`?

answer: (2, 2, 2)

.squeeze() common enemy

RuntimeError: Expected 2D input, got 1D tensor

This usually happens during:

- Validation
- Inference
- Debugging with a single sample

Fix: x.squeeze(dim=1)

.unsqueeze()

- Adds a dimension with size 1 at a specified position.
- The position is controlled through the `dim` index

.unsqueeze()

```
import torch

x = torch.randn(3, 28, 28)
x_unsq = x.unsqueeze(0)
print("Unsqueeze:", x_unsq.shape)
```

```
Unsqueeze: torch.Size([1, 3, 28, 28])
```

***note**: you can use both x.unsqueeze() or torch.unsqueeze(x)

.unsqueeze()

torch.unsqueeze

`torch.unsqueeze(input, dim) → Tensor`

Returns a new tensor with a dimension of size one inserted at the specified position.

The returned tensor shares the same underlying data with this tensor.

A `dim` value within the range `[-input.dim() - 1, input.dim() + 1)` can be used. Negative `dim` will correspond to `unsqueeze()` applied at `dim` = `dim + input.dim() + 1`.

Parameters:

- **input** (*Tensor*) – the input tensor.
- **dim** (*int*) – the index at which to insert the singleton dimension

.unsqueeze()

question: given the tensor x = torch.tensor([1, 2, 3, 4]), what is the output tensor when we perform `x.unsqueeze(dim=0)`?

answer: tensor([[1, 2, 3, 4]])

question: what is the output tensor if we instead apply `x.unsqueeze(dim=1)`?

answer: tensor([[1], [2], [3], [4]])

.unsqueeze() common enemies

RuntimeError: Expected input batch_size (N) to match target batch_size (M)

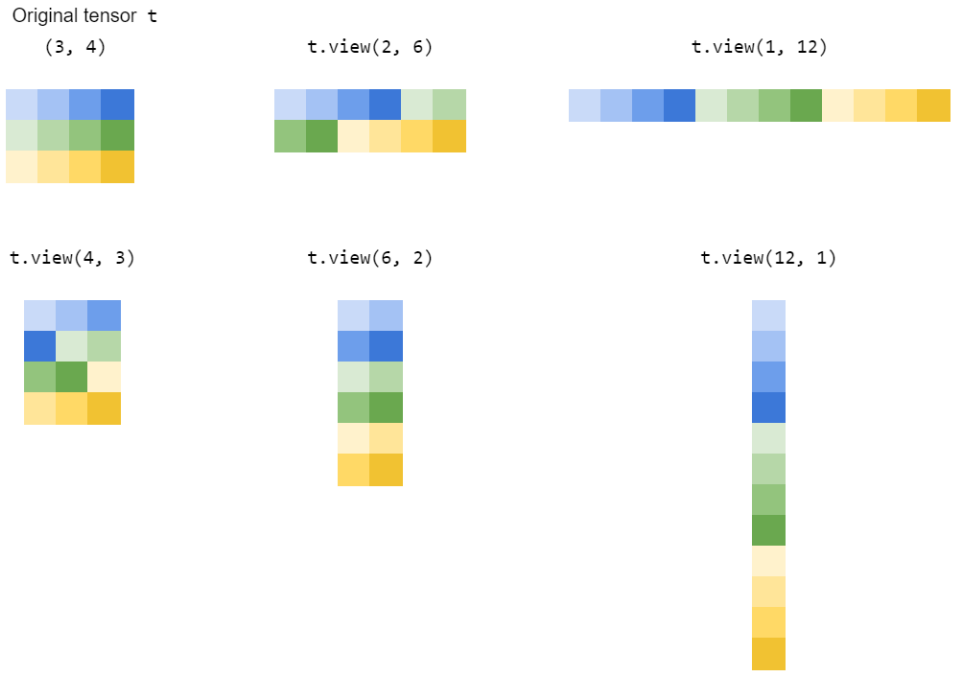
or

RuntimeError: mat1 and mat2 shapes cannot be multiplied

Fix: x.unsqueeze(dim=1) and check tensor shapes

.view()

- Reshapes a tensor freely while maintaining the same number of elements.
- Input parameter `*shape` is to control the desired size



.view()

```
import torch

x = torch.randn(32, 28, 28, 3)
x_view = x.view(32, -1)
print("View:", x_view.shape)
```

```
View: torch.Size([32, 2352])
```

*Important note: In view, -1 infers the dimension automatically so that the total number of elements stays the same.

Think -1 = whatever size makes the math work

.view()

torch.Tensor.view

`Tensor.view(*shape) → Tensor`

Returns a new tensor with the same data as the `self` tensor but of a different `shape`.

The returned tensor shares the same data and must have the same number of elements, but may have a different size. For a tensor to be viewed, the new view size must be compatible with its original size and stride, i.e., each new view dimension must either be a subspace of an original dimension, or only span across original dimensions $d, d+1, \ldots, d+k$ that satisfy the following contiguity-like condition that $\forall i = d, \ldots, d+k-1$,

$$\text{stride}[i] = \text{stride}[i+1] \times \text{size}[i+1]$$

Otherwise, it will not be possible to view `self` tensor as `shape` without copying it (e.g., via `contiguous()`). When it is unclear whether a `view()` can be performed, it is advisable to use `reshape()`, which returns a view if the shapes are compatible, and copies (equivalent to calling `contiguous()`) otherwise.

Parameters:

shape (*torch.Size* or *int*...) – the desired size

.view()

question: given the tensor x = torch.randn(4, 4), what is the output shape when we perform `x.view(16)`?

answer: 16

question: what is the output shape if we instead performed `x.view(-1, 8)`?

answer: (2, 8)

Changing data type

Depending on the loss function used, prediction and target tensors need to be cast to a specific dtype

Examples:

- nn.BCEWithLogitsLoss requires:
  - Predictions: float
  - Targets: float
- If targets are int, we will get:

RuntimeError: result type Float can't be cast to Long

Changing data type

Depending on the loss function used, prediction and target tensors need to be cast to a specific dtype

question: What are the appropriate data types for **logits** and **targets** when computing the loss using nn.CrossEntropyLoss?

*hint: Go to nn.CrossEntropyLoss documentation, and look at the examples section

answer: logits: float, targets: long

Changing data type

- Use `.to(dtype)` to convert a tensor's data type.

```
import torch

# Create a float32 tensor
x = torch.tensor([1.2, 2.3, 3.4], dtype=torch.float32)
print(x.dtype)  # Output: torch.float32

# Convert to float16
x_half = x.to(torch.float16)
print(x_half.dtype)  # Output: torch.float16
```

```
torch.float32
torch.float16
```

Move Tensors to GPU (if available)

- GPUs are faster and more efficient in most cases when training or inferencing deep learning models
- Use `.to(device)` to move a tensor to GPU for faster computation.

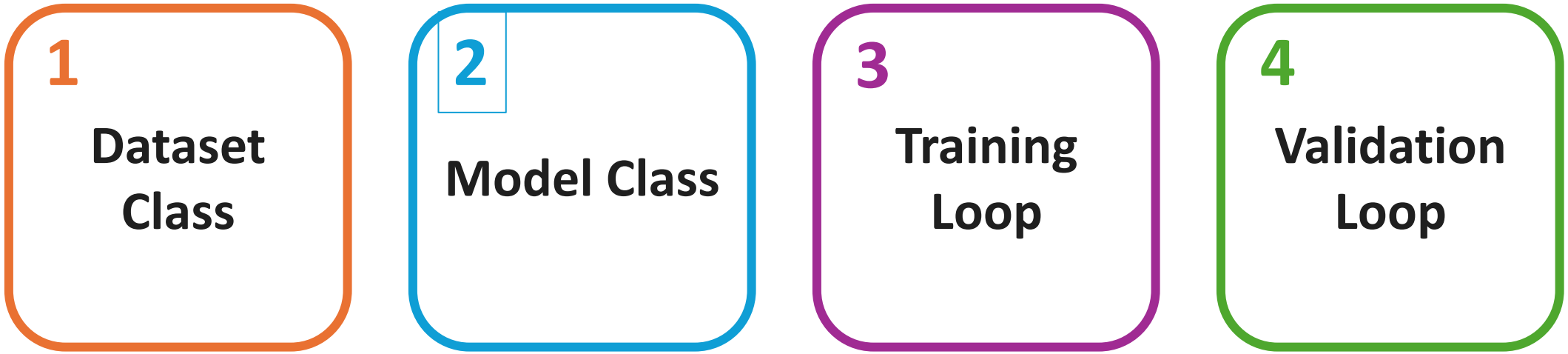
```
# Automatically select CPU or GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Create a tensor and move it to GPU
x_gpu = x.to(device)
print(x_gpu.device)  # Output: cuda:0 (if GPU is available) or cpu
```

```
cpu
```

PyTorch Workflow Organization

consists of 4 main components:



Dataset class

- A Dataset Class is responsible for transforming raw data into samples that are ready to be used by a model.
- Each sample returned consists of:
  - An input (features or image)
  - Its corresponding label

Dataset class

Typical workflow:

1. Load data
2. Train/Test split
3. Standardize
4. Transform to tensors
5. Wrap in `TensorDataset`
6. Use `DataLoader`

Dataset class

1.
```
# load data
data = load_breast_cancer()
X = data.data
y = data.target
```

2.
```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

3.
```
# Standardize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

4.
```
# transform to tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test  = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
y_test  = torch.tensor(y_test, dtype=torch.float32)
```

Dataset class

5.

```
# TensorDataset pairs input features (X) with their corresponding labels (y)
# Each item in the dataset is returned as (X[i], y[i])
train_dataset = TensorDataset(X_train, y_train)
test_dataset = TensorDataset(X_test, y_test)
```

6.

```
# DataLoader for training data
train_loader = DataLoader(
    train_dataset,
    batch_size=32,
    shuffle=True,
    num_workers=2
)
# DataLoader for test/validation data
test_loader = DataLoader(
    test_dataset,
    batch_size=32,
    shuffle=False,
    num_workers=2
)
```

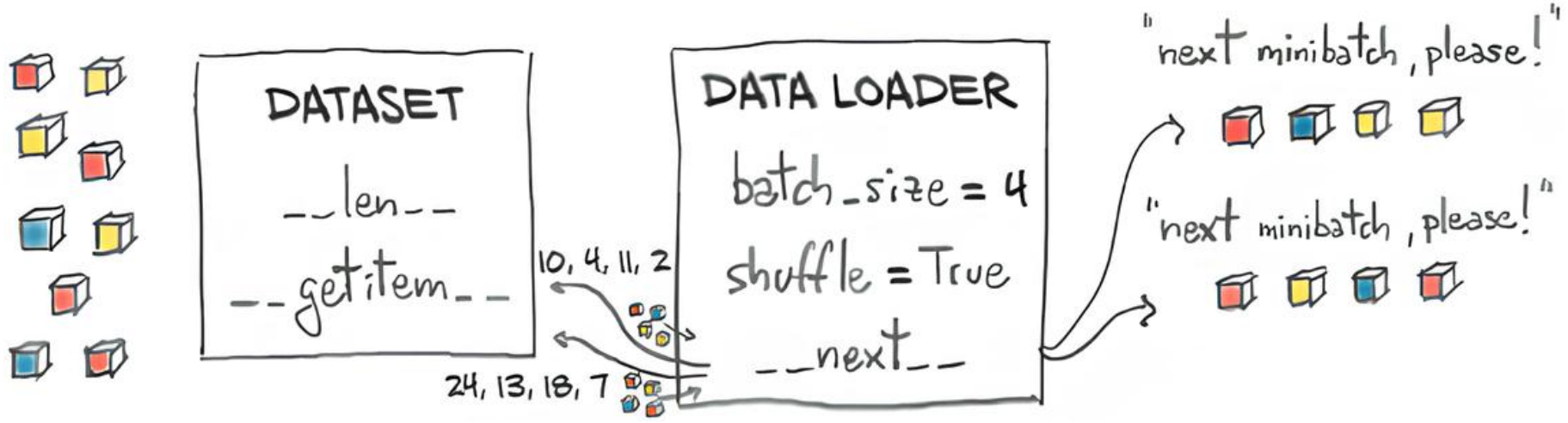
TensorDataset

- A wrapper that lets you treat one or more tensors as a dataset, so they can be fed cleanly into a `DataLoader` for training.
- we can use `TensorDataset` to pair input features with their labels.

DataLoader

Problem: We need batches, not single samples

Solution: We use `DataLoader` to handle batching automatically.



DataLoader

- A `DataLoader` is a PyTorch utility that takes a Dataset and does:
  - **Batching**: Groups multiple samples together for faster processing.
  - **Shuffling**: Randomizes data order to improve training.
  - **Multi-threading**: Loads data efficiently in parallel.

Argument	Description
`dataset`	The dataset object (e.g., `train_dataset`)
`batch_size`	Number of samples per batch (e.g., `32`)
`shuffle`	Whether to **randomly shuffle** data each epoch (`True` = better for training)
`num_workers`	Number of parallel **CPU workers** to load data faster
`collate_fn`	A function to **customize how data is stacked** (useful when data has variable sizes)

Model Class

In PyTorch, `nn.Linear(in_features, out_features)` creates a fully connected (dense) layer that applies a linear transformation:

$$y = xW^T + b$$

`in_features`: number of input features

`out_features`: number of neurons (outputs) in the layer

Model Class

```python
import torch
import torch.nn as nn

# Create a linear layer: 3 input features -> 1 output
linear = nn.Linear(in_features=3, out_features=1)

# Example input: batch of 4 samples, each with 3 features
x = torch.randn(4, 3)

# Forward through the layer
y = linear(x)

print("Input shape:", x.shape)   # torch.Size([4, 3])
print("Output shape:", y.shape)  # torch.Size([4, 1])
```

```
Input shape: torch.Size([4, 3])
Output shape: torch.Size([4, 1])
```

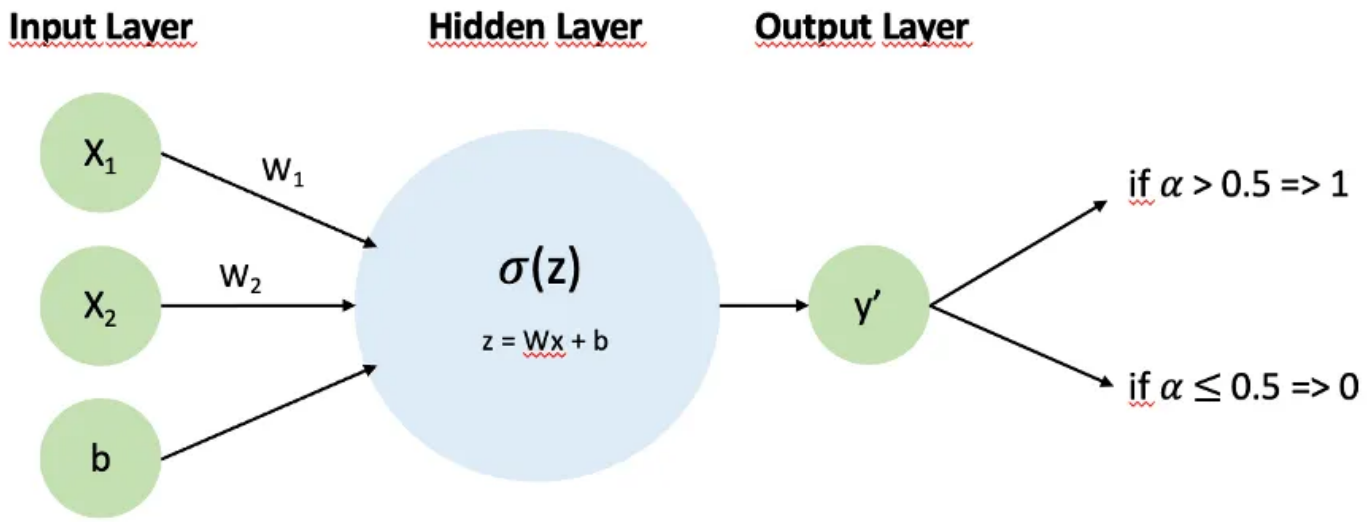
Model Class

Inside the `__init__` method, we define what the neural network looks like. This includes:

- Number of layers (`nn.Linear`)
- Hidden layer sizes
- Activation functions (ex: ReLU)
- Output activation function (ex: Sigmoid, Softmax)

Model Class

The `forward()` method defines how the input data flows through the model to produce the final output.



Model Class

1-Layer NN

```
class NN1Layer(nn.Module):

  def __init__(self, input_dim):

    super(NN1Layer, self).__init__()
    # input_dim = num of features, Output for binary classification is 1
    self.layer_1 = nn.Linear(input_dim, 1)

    # output activation function
    self.sigmoid = nn.Sigmoid()

  # forward pass
  def forward(self, x):
    z = self.layer_1(x)
    a = self.sigmoid(z)
    return a
```

2-Layer NN

```
class NN2Layer(nn.Module):

    def __init__(self, input_dim, hidden_dim):

        super(NN2Layer, self).__init__()
        # input_dim = num of features, hidden_dim = num of neurons
        self.layer1 = nn.Linear(input_dim, hidden_dim)
        # hidden_dim = num of neurons, Output for binary classification is 1
        self.layer2 = nn.Linear(hidden_dim, 1)

        # non-linearity activation function
        self.relu = nn.ReLU()
        # output activation function
        self.sigmoid = nn.Sigmoid()

    # forward pass
    def forward(self, x):
        z1 = self.layer1(x)
        a1 = self.relu(z1)
        z2 = self.layer2(a1)
        a2 = self.sigmoid(z2)
        return a2
```

Training Loop

The training loop is responsible for updating the model's weights so that it learns to minimize the loss function.

Parameters

- model – The neural network to be trained.
- optimizer – Updates model parameters (e.g., SGD, Adam).
- criterion – Loss function, depends on the task.
- train_loader – PyTorch DataLoader that provides batches of training data.
- device – Device used for computation (cpu or cuda).

Training Loop

```python
def train_one_epoch(model, optimizer, criterion, train_loader, device):
    # Set the model to training mode
    model.train()

    running_loss = 0.0

    for X_batch, y_batch in train_loader:
        # Move batch to the selected device
        X_batch = X_batch.to(device)
        y_batch = y_batch.view(-1, 1).to(device)

        # Forward pass
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)

        # Backward pass & optimization
        optimizer.zero_grad()   # Clear previous gradients
        loss.backward()         # Compute gradients
        optimizer.step()        # Update model parameters

        running_loss += loss.item()

    # Average loss over all batches
    avg_loss = running_loss / len(train_loader)

    return avg_loss
```

Validation Loop

The validation loop evaluates the model's performance on unseen data without updating the weights.

It is used to measure how well the model generalizes.

Parameters

- model – The trained neural network to be evaluated.
- criterion – Loss function used for evaluation
- test_loader – PyTorch DataLoader that provides batches of validation/test data.
- device – Device used for computation (cpu or cuda).

Validation Loop

```python
def validate(model, criterion, test_loader, device):
    # Set the model to evaluation mode
    model.eval()

    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():  # Disable gradient computation
        for X_batch, y_batch in test_loader:
            # Move batch to the selected device
            X_batch = X_batch.to(device)
            y_batch = y_batch.view(-1, 1).to(device)

            # Forward pass
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            running_loss += loss.item()

            # Binary predictions
            predicted = (outputs > 0.5).float()

            # Accuracy calculation
            correct += (predicted == y_batch).sum().item()
            total += y_batch.size(0)

    avg_loss = running_loss / len(test_loader)
    accuracy = correct / total

    return avg_loss, accuracy
```

Note: Gradients are disabled during validation using torch.no_grad() to improve efficiency and prevent weight updates.