

TK

[← Home](#)

Mastering PyTorch: From Linear Regression to Computer Vision

TK 2025-11-26 • 27 min read

[#python](#) [#data_science](#) [#machine_learning](#)



Photo by Zach Lezniewicz

As part of my training to master machine learning, I keep honing my Python and PyTorch skills. Because ML is a very empirical field, I need hands-on practice and solve a lot of problems. This is why mastering an ML framework is an important step to master the field.

In this post, I will share a couple of building blocks or fundamental pieces of PyTorch and how you can use them to build and train ML models.

Here are the topics I have in mind:

- Tensors, the data building blocks
- Tensor operations
- Building and training a Linear Regression
- Data Management: datasets, data splitting, and dataloaders
- Building and training a simple CNN model on FashionMNIST
- Building and training a CNN model for a horse or human classification

Tensors, the data building blocks

A tensor is a fundamental piece of PyTorch and how it's used to build and train models. We'll learn how it is used to hold data, how you can use it to perform operations on top of data, and an important feature (gradients), especially for model training and model 'learning'.

To create a tensor, we need to import torch and wrap the value with the tensor. Let's create a tensor for a simple value:

```
value = torch.tensor(1) # tensor(1)
```

You can see that the data was already there, but now it's wrapped by the tensor object.

It's created with type int64 and we can show that using the dtype property.

```
torch.tensor(1).dtype # torch.int64
```

We can also have other data types for tensors. `float64` for extra precision, `int8` for memory efficiency, or `float32` as a default and reliable data type.

Tensors can hold other data structures, like vectors and matrices, which are very common types of data structures in machine learning.

```
vector = torch.tensor([1, 2, 3, 4, 5]) # tensor([1, 2, 3, 4, 5])
```

For matrices, it works in a similar way, but passing a different data structure:

```
matrix = torch.tensor([
    [1, 2, 3],
    [2, 3, 4],
    [3, 4, 5],
    [4, 5, 6],
    [5, 6, 7]
])
```

Other than `dtype`, we have other properties and functions that we can use to extract information about the tensor.

Let's say we want to check the tensor's dimension.

```
value.ndimension() # 0
vector.ndimension() # 1
matrix.ndimension() # 2
```

We can also check the tensor's size:

```
value.size() # torch.Size([])
vector.size() # torch.Size([5])
matrix.size() # torch.Size([5, 3])
```

value has zero dimension, vector has a list of 5 items, and matrix a list of 5 lists, each list with 3 elements.

This is similar to calling `shape` on each tensor:

```
value.shape # torch.Size([])
vector.shape # torch.Size([5])
matrix.shape # torch.Size([5, 3])
```

But `shape` is a property, not a function, of the tensor object.

Another important function of tensors is `item`. It returns the actual value the tensor holds. But this only works on tensors with one element.

```
value.item() # 1
vector.item() # RuntimeError: a Tensor with 5 elements cannot be converted to Scalar
matrix.item() # RuntimeError: a Tensor with 15 elements cannot be converted to Scalar
```

To extract an item from vector and matrix, we should get a tensor with only one element.

```
vector[0].item() # 1
vector[1].item() # 2
vector[2].item() # 3

matrix[0][0].item() # 1
matrix[0][1].item() # 2
matrix[0][2].item() # 3
```

These are the fundamental ideas of tensors and how they hold data.

Tensor Operations

In this section, we'll see how we can perform operations on tensors. To be able to transform, update, and handle data wrapped by tensors, we

need to understand how reshaping, indexing, slicing, and math and boolean operations work. Besides that, we'll see how derivatives are done in tensors, as this is an important feature to let models learn in backpropagation and gradient descent.

Reshaping Operations

In reshaping operations, it's common to use functions like `unsqueeze`, `squeeze`, and `view`. In simple terms, `unsqueeze` adds a new dimension to a tensor, `squeeze` removes a dimension, and `view` reshapes a tensor object based on the 'row' and 'column' parameters passed.

Let's see them in practice.

`unsqueeze` adds a new dimension to the tensor object. We need to pass which index where we need to insert a new dimension.

```
tensor = torch.tensor(1)
batch = tensor.unsqueeze(0)
tensor.shape # torch.Size([])
batch.shape  # torch.Size([1])
```

Calling `shape`, we can see the new dimension added to the tensor. This is especially useful when we need to match tensors shape, and when there's a shape mismatch error between the model and the input data.

`squeeze` works in reverse. It removes the dimension.

```
tensor = torch.tensor([1.0])
squeezed_tensor = tensor.squeeze(0)
tensor.shape      # torch.Size([1])
squeezed_tensor   # torch.Size([])
```

`view` is another function that serves as a reshaping operation. It's a very common function used in ML model training, as it's flexible and data needs to be reshaped all the time to handle mismatch situations.

Imagine we have a tensor list with 5 elements. Its shape has size `[5]`.

But we want to reshape from a 1-dimensional to a 2-dimensional tensor. Let's see it in practice:

```
tensor = torch.tensor([0, 1, 2, 3, 4]) # tensor([0, 1, 2, 3, 4])
tensor.shape                          # torch.Size([5])
reshaped_tensor = tensor.view(5, 1)
reshaped_tensor                       # tensor([[0],[1],[2],[3],[4]])
reshaped_tensor.shape                 # torch.Size([5, 1])
```

The first parameter of the `view` function is the row and the second is the column. This means that we went from a tensor with 1 dimension, a list of 5 elements, to a new 2-dimensional tensor with 5 rows and 1 column.

The size went from `[5]` to `[5, 1]`.

If we have a tensor with a dynamic size, we can use the `-1` placeholder. It's pretty common to see this type of usage:

```
tensor = torch.tensor([0, 1, 2, 3, 4]) # tensor([0, 1, 2, 3, 4])
tensor.shape                          # torch.Size([5])
reshaped_tensor = tensor.view(-1, 1)
reshaped_tensor                       # tensor([[0],[1],[2],[3],[4]])
reshaped_tensor.shape                 # torch.Size([5, 1])
```

It infers the total number of elements in the original tensor.

Indexing & Slicing Operations

Now that we understand more about reshaping operations, let's see how indexing and slicing work.

An indexing operation is similar to an index in Python. We pass the specified index to the tensor, and expect it to return the respective tensor. Take this tensor as an example:

```
tensor = torch.tensor([
    [1, 2, 3, 4],
```

```
[5, 6, 7, 8],  
[9, 10, 11, 12]  
)
```

Let's perform some indexing operations on top of it.

```
second_row_third_column = tensor[1, 2]  
last_row = tensor[-1]  
second_row_third_column # tensor(7)  
last_row                 # tensor([9, 10, 11, 12])
```

Getting the second row and third column looks similar to how Python handles indexing. But it's written in the same bracket. We can also do:

```
second_row_third_column = tensor[1][2]  
second_row_third_column # tensor(7)
```

Which is exactly how we do it in Python.

Using the concept of slicing, we can also perform other interesting operations like “Get the third column and further of the second row”:

```
tensor[1][2:] # tensor([7, 8])
```

We can also do it for both the columns and rows together:

```
tensor[:2, 2:]  
# tensor([[3, 4],  
#         [7, 8]])
```

This is getting the second column and further for the first and second rows.

Math & Boolean Operations

These are also pretty important operations and are highly used when handling data and training models.

The `min` function returns the minimum value of all elements in the tensor object.

```
torch.tensor([1, 2, 3, 4, 5]).min() # tensor(1)
```

For a more complex tensor, it also returns the minimum value:

```
torch.tensor([
    [16, 32, 73, 94, 75],
    [31, 42, 23, 84, 35],
    [41, 52, 13, 74, 55]
]).min()
# tensor(13)
```

The `max` is similar but returns the maximum value.

```
torch.tensor([1, 2, 3, 4, 5]).max() # tensor(5)
```

With a more complex tensor:

```
torch.tensor([
    [16, 32, 73, 94, 75],
    [31, 42, 23, 84, 35],
    [41, 52, 13, 74, 55]
]).max()
# tensor(94)
```

We can also get statistics from tensor values using the `mean` (average) and `std` (standard deviation).

```
data = torch.tensor([10.0, 20.0, 30.0, 40.0, 50.0])
data.mean() # tensor(30.)
```

```
data.std() # tensor(15.8114)
```

And finally, some math operations. We are going to see addition, multiplication, dot product, and matrix multiplication. They are highly used in ML.

In matrix addition, we sum the respective values in each row-column:

```
t1 = torch.tensor([1, 0])
t2 = torch.tensor([0, 1])

t1 + t2 # tensor([1, 1])
```

In multiplication, it performs a similar operation to addition, but it multiplies each value:

```
t1 = torch.tensor([1, 2])
t2 = torch.tensor([3, 4])

t1 * t2 # tensor([4, 6])
```

In the dot product, it multiplies each value of one tensor by its respective value in the other tensor, and then it sums everything.

```
t1 = torch.tensor([1, 2, 3])
t2 = torch.tensor([4, 5, 6])

torch.dot(t1, t2) # tensor(32)
# 1 * 4 + 2 * 5 + 3 * 6
# 4 + 10 + 18
# 32
```

In matrix multiplication, we need to follow the rules of linear algebra, where matrices should match: the left matrix's column should match the right matrix's row.

One example is a `matrix1` with 2 rows and 3 columns forming a shape (2, 3) and a `matrix2` with 3 rows and 2 columns forming a shape (3, 2).

The number of columns of `matrix1` (3) matches the number of rows of `matrix2` (3). Let's see it in practice:

```
matrix1 = torch.tensor([[0, 1, 1], [1, 0, 1]])
matrix2 = torch.tensor([[1, 1], [1, 0], [0, 1]])

torch.mm(matrix1, matrix2)
# tensor([[1, 1],
#         [1, 2]])
```

Derivatives in tensor

Another important feature of tensors that I think is worth mentioning is the idea of derivatives and gradients.

Derivatives in tensors are the building blocks of how neural networks learn through gradient descent. Here is an example:

```
x = torch.tensor(2.0, requires_grad=True) # tensor(2., requires_grad=True)
y = x ** 2                                # tensor(4., grad_fn=<PowBackward0>)
y.backward()
x.grad                                    # tensor(4.)
```

For a tensor of value 2, we use it to build a linear relationship for `y`. Calling `backward` on `y`, it computes the gradients of `y` with respect to all tensors that `y` depends on, in this case, `x`.

The gradient is stored in the `grad` attribute of `x` in the form of `2 * x` (which is the derivative of `x ** 2`). When calling `x.grad`, it passes `x = 2.0` and returns `2 * 2.0 = 4.0` in the form of a tensor `tensor(4.)`.

The default value of `requires_grad` is `True` so we don't need to specify it, but I wanted to show it explicitly for us to visualize it.

To finalize this tensor piece, I think it's useful to learn about how to go from NumPy data and Pandas dataframes to PyTorch tensors and vice versa.

To transform NumPy data into tensors, we use the `from_numpy` function:

```
vector = np.array([1.0, 2.0, 3.0])  
torch.from_numpy(vector) # tensor([1., 2., 3.], dtype=torch.float64)
```

And going from PyTorch to NumPy, we use the `numpy` function.

```
tensor = torch.tensor([1., 2., 3.])  
tensor.numpy() # array([1., 2., 3.], dtype=float32)
```

From Pandas dataframes, it follows the same process, because we transform the dataframes into NumPy values using the `values` method and then use the same `from_numpy` function:

```
df = pd.DataFrame([1.0, 2.0, 3.0])  
torch.from_numpy(df.values) # tensor([1., 2., 3.], dtype=torch.float64)
```

To transform into dataframes, we don't need to transform into NumPy anymore. We just pass the tensor to the DataFrame object.

```
tensor = torch.tensor([1., 2., 3.])  
pd.DataFrame(tensor)
```

Enough of tensors. Let's build models and learn how to train them.

Building and training a Linear Regression

Before moving to a more complex model like CNN for computer vision

problems, we take a step back and build the simplest model possible to understand some important building blocks: how to build ML models and train them in PyTorch.

Let's build a linear regression with two parameters, a weight w and a bias b .

(TODO: replace it with katex)

$$y = xw + b$$

For this, we can use the `Linear` class to build this model.

```
model = nn.Linear(in_features=1, out_features=1, bias=True)
model(torch.tensor([[2.], [4.])))
# if weight = -0.4443 and bias = -0.5045,
# the result will be tensor([[ -1.3930], [ -2.2815]])
```

Another important piece of PyTorch is custom models using `nn.Module`. Let's see it in practice:

```
class LinearRegression(nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        return self.linear(x)
```

We built a new `LinearRegression` custom model, and define the `Linear` representation as we did before.

In the `__init__` method, we define this linear representation with custom input and output features.

And in the `forward` method, we call the linear model. The `forward` method is called when we pass the input data to the model. x is the

input data the model receives.

With the model created, let's train it. This is another important piece of PyTorch that keeps repeating in ML training.

The training looks like this: predict → calculate loss → adjust parameters → predict → calculate loss → adjust parameters. The loop goes on until it finishes all epochs.

- To predict, we need the model
- To calculate the loss, we need a loss function
- To adjust the parameters, we need an optimizer

Here is a more detailed version of this loop:

- Zero the gradients in the optimizer
- Do model prediction
- Compute the loss
- Compute the gradients
- Update the parameters

First, we define the input data. We are going to measure the relationship between distance and time.

```
distances = torch.tensor([[1.0], [2.0], [3.0], [4.0]], dtype=torch.float32)
times = torch.tensor([[6.96], [12.11], [16.77], [22.21]], dtype=torch.float32)
```

`distances` is the input and `times` are the targets.

Let's see the model training in practice:

```
loss_function = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

for epoch in range(500):
    optimizer.zero_grad()
    outputs = model(distances)
    loss = loss_function(outputs, times)
    loss.backward()
```

```
optimizer.step()

if epoch % 50 == 0:
    print(f"Epoch {epoch}: Loss = {loss.item()}")
```

And now let's unpack what's going on here.

As I mentioned before, we need the model, a loss function, and an optimizer to build the entire training process. The model was built before. Now we should define the `loss_function`, which is an MSE loss function, and an `optimizer`, which is an SGD.

It runs on 500 epochs. In each epoch, we should zero the gradients in the optimizer. This is important so the optimizer doesn't carry the gradient through epochs, which would lead to wrong parameter updates.

Then we call the model to make the prediction. We pass the prediction output and the target to the `loss_function` to compute the loss. The `backward` method performs backpropagation and computes the gradients. And finally, the `step` method updates the model parameters.

Data Management: datasets, data splitting, and dataloaders

In this section, we'll talk about data management, that is basically how to download data, build custom datasets, apply some transformations (preprocessing & augmentation), split data, and build dataloaders.

We start with the download task. There is a flower images dataset that we need to download and store locally. It comes as a `tgz` file, so we need to unzip it. From another url, we will download the labels of each flower that are the actual targets used for model training.

Let's start by creating a folder locally so we can store the downloaded data:

```
os.makedirs("flower_data", exist_ok=True)
```

To request and download the data, we'll use `requests` :

```
import requests

image_url = "https://www.robots.ox.ac.uk/~vgg/data/flowers/102/102flowers.tgz"
response = requests.get(image_url, stream=True)
```

Noticed that we added a `stream` parameter to the request so it can stream the data.

Now, let's define the paths:

```
# used in kaggle notebooks (but can be replaced with other directories)
root_dir = "/kaggle/working/flower_data"

tgz_path = os.path.join(root_dir, '102flowers.tgz')
```

We also want to define the total size of the response content because we'll iterate over the data in chunks to store it locally.

```
from tqdm import tqdm

total_size = int(response.headers.get("content-length", 0))

with open(tgz_path, "wb") as file:
    for data in tqdm(
        response.iter_content(chunk_size=1024),
        total=total_size // 1024
    ):
        file.write(data)
```

We use `tqdm` to time the iterations based on each chunk being stored locally. That way, we can visually see this process making progress.

The last part of this data is to extract the `tgz` file:

```
import tarfile
```

```
with tarfile.open(tgz_path, "r:gz") as tar:  
    tar.extractall(root_dir)
```

That done, we should do a similar operation for the labels. Request and download the data and store it locally.

```
labels_file_path = os.path.join(root_dir, 'imagelabels.mat')  
  
labels_url = "https://www.robots.ox.ac.uk/~vgg/data/flowers/102/imagelabels.mat"  
response = requests.get(labels_url, stream=True)  
total_size = int(response.headers.get("content-length", 0))  
  
with open(labels_file_path, "wb") as file:  
    for data in tqdm(  
        response.iter_content(chunk_size=1024),  
        total=total_size // 1024,  
    ):  
        file.write(data)
```

This data doesn't need to be extracted, as it is a mat file.

With the downloaded data, we can start building the dataset.

Specifically for image data, this idea of custom datasets is quite interesting because we can 'lazily' open each input data (pixels) as images individually on demand, so we don't overflow the memory usage.

This custom dataset has three features:

- `__init__`: setup the data and the data path
- `__len__`: sample size
- `__getitem__`: get item from sample through index (idx) – it should return the image and the label

Let's build it:

```
import scipy  
  
from torch.utils.data import Dataset
```

```
from PIL import Image

class OxfordFlowersDataset(Dataset):
    def __init__(self, root_dir):
        self.root_dir = root_dir
        self.img_dir = os.path.join(root_dir, "jpg")

        labels_mat = scipy.io.loadmat(os.path.join(root_dir, "image_labels.mat"))
        self.labels = labels_mat["labels"][0] - 1

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        image_name = f"image_{idx+1:05d}.jpg"
        image_path = os.path.join(self.img_dir, image_name)
        image = Image.open(image_path)
        label = self.labels[idx]

        return image, label
```

Now let's use it:

```
dataset = OxfordFlowersDataset(root_dir)
img, label = dataset[0]
img      # <PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=591x500>
label    # 76
```

Because now we have the actual image and not the pixels, we can plot it:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))
plt.title(label)
plt.imshow(np.array(img))
plt.axis("off")
plt.show
```

Here is the image with label 76:

76



With the dataset built, now it's time to learn about transformations. There are two important pieces: preprocessing and data augmentation.

For image data, we preprocess data using resize, crop, transform into tensors, and normalize it so that it can be transformed into a standard format used across the model training.

Let's start with the preprocessing piece. The idea is to standardize each item in the dataset so all of them look the same and the model can learn properly.

```
from torchvision import transforms
```

```
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

With the transforms, we can use Resize, we can crop the image, transform it into tensors, and normalize the data.

The idea is to pass this transform to the dataset and it can be used to transform each image when we access each of them.

```
class OxfordFlowersDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.img_dir = os.path.join(root_dir, "jpg")
        self.transform = transform

        labels_mat = scipy.io.loadmat(os.path.join(root_dir, "imagelabels.mat"))
        self.labels = labels_mat["labels"][0] - 1

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        image_name = f"image_{idx+1:05d}.jpg"
        image_path = os.path.join(self.img_dir, image_name)
        image = Image.open(image_path)
        label = self.labels[idx]

        if self.transform:
            image = self.transform(image)

        return image, label
```

The dataset receives the transform and set it in the initializer. In the `__getitem__` method, we should transform the image as the final part of this process if there's a transform passed to the dataset.

An interesting way to debug each part of this transformation process is to do it one by one and see what we get as output:

```
raw_dataset = OxfordFlowersDataset(root_dir)
image, label = raw_dataset[0]

print(f"Raw image: {image.size}")
image = transforms.Resize(256)(image)
print(f"Resized image: {image.size}")
image = transforms.CenterCrop(224)(image)
print(f"Cropped image: {image.size}")
image = transforms.ToTensor()(image)
print(f"Tensor image: {image.shape}, range[{image.min():.1f}, {image.max():.1f}]")
image = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
print(f"Normalized image: {image.shape}, range[{image.min():.1f}, {image.max():.1f}]")

# Raw image: (591, 500)
# Resized image: (302, 256)
# Cropped image: (224, 224)
# Tensor image: torch.Size([3, 224, 224]), range[0.0, 1.0]
# Normalized image: torch.Size([3, 224, 224]), range[-2.1, 2.6]
```

Data augmentation is another part of the data transformation, but the idea is to create copies of existing data to increase the size and diversity of the training dataset.

For this, we have APIs like `RandomHorizontalFlip`, `RandomRotation`, `RandomAffine`, etc. When building the CNN model for horse or human classification, we'll take a closer look at how to use these APIs to increase data diversity for the model to be trained.

To split data into training, validation, and test, we just use basic math together with the `random_split` API.

This is how we want to split the data:

- Training: ~5700 (70%)
- Validation: ~1200 (15%)
- Test: ~1200 (15%)

```
from torch.utils.data import random_split

train_size = int(len(dataset) * 0.7)
validation_size = int(len(dataset) * 0.15)
test_size = len(dataset) - train_size - validation_size

train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, va
```

`random_split` will get sizes and randomly split the dataset into training, validation, and test datasets with the specific sizes.

With the split done, we can pass the outputs to the dataloader. The dataloader is very useful because it comes with nice features, for example, shuffling the data, organizing the data into batches with specified batch sizes, and we can loop through it.

Let's see it in practice:

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)
validation_dataloader = DataLoader(validation_dataset, batch_size=32, shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Here is an important consideration:

- Shuffling the training set is important so the model doesn't learn through the position of each image.
- Shuffling is not important for validation and test datasets because the model is not learning, it's only evaluating.

We can loop through the `train_dataloader` to see each item's shape:

```
for images, labels in train_dataloader:
    print(f"Batch: {images.shape}, {labels.shape}")
    break

# Batch: torch.Size([32, 3, 224, 224]), torch.Size([32])
```

Building and training a Neural Network model on FashionMNIST

Before moving to a more complex model, we are going to build a neural network model to train on the FashionMNIST data.

The first step is to download, transform, split, and put it into a dataloader:

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.2860,), (0.3530,))
])

train_dataset = datasets.FashionMNIST(root="./data", train=True, download=True,
test_dataset = datasets.FashionMNIST(root="./data", train=False, download=True,

train_dataloader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

For this problem, we split the data into training and test, with a batch size of 64. In terms of transformation, we preprocess the data, transforming it into tensors and normalizing it.

```
for image, label in train_dataloader:
    print(image.shape, label.shape)
    break

# torch.Size([64, 1, 28, 28]) torch.Size([64])
```

This is a greyscale 28x28 image dataset in a batch size of 64 images. The label follows the same batch size.

The idea here is to have a model that receives this type of data and classifies it against 10 classes. So the model is basically a classifier:

```
class FashionMNISTModel(nn.Module):
    def __init__(self):
        super(FashionMNISTModel, self).__init__()
        self.flatten = nn.Flatten()
        self.linear = nn.Sequential(
            nn.Linear(28 * 28, 128),
            nn.ReLU(),
            nn.Linear(128, 10),
            nn.LogSoftmax(dim=1)
        )

    def forward(self, x):
        logits = self.linear(self.flatten(x))
        return logits
```

Let's unpack what we have here. This is a neural network with two layers.

The first step is to flatten the data, because the model doesn't accept 2 or 3-dimensional tensors. We should flatten the data, so it goes from `torch.Size([64, 1, 28, 28])` to `torch.Size([64, 784])`.

$28 * 28 = 784$: the size of the flattened image. 128 is the number of neurons in this first layer.

Then, we apply the ReLU function, another linear layer going from 128 to 10 neurons. We choose 10 because we have 10 classes. And finally, we have `LogSoftmax`, that calculates the log of the probabilities of all 10 classes.

In the forward method, we just put everything together: flatten + linear.

Before training the model, we should set up important components: model, loss function, and optimizer:

```
model = FashionMNISTModel()
loss_function = nn.NLLLoss()
optimizer = optim.Adam(model.parameters())
```

There is another important function we need to build to get the metric of the model and evaluate it through the training process: accuracy.

```
def get_accuracy(pred, labels):
    _, predictions = torch.max(pred, 1)
    correct = predictions.eq(labels).float().sum()
    accuracy = correct / labels.shape[0]
    return accuracy
```

To train the model, we need the process we talked about before: zero the gradients in the optimizer, call the model, compute the loss, do backpropagation, and update the parameters.

```
def train(dataloader, model, loss_function, optimizer):
    size = len(dataloader.dataset)
    total_loss, total_accuracy = 0, 0
    model.train()

    for batch, (x, y) in enumerate(dataloader):
        optimizer.zero_grad()
        logits = model(x)
        loss = loss_function(logits, y)
        loss.backward()
        optimizer.step()
        accuracy = get_accuracy(logits, y)
        total_accuracy += accuracy.item()
        total_loss += loss.item()

    if batch % 100 == 0:
        avg_loss = total_loss / (batch + 1)
        avg_accuracy = total_accuracy / (batch + 1) * 100.0
```

```
print(f"Batch {batch}, Loss: {avg_loss:>7f}, Accuracy: {avg_accuracy
```

And now we can call the `train` function over `N` epochs. In the following example, we have 5 epochs:

```
epochs = 5

for epoch in range(epochs):
    print(f"Epoch {epoch+1}\n-----")
    train(train_dataloader, model, loss_function, optimizer)
    print("=====")

print("Done!")

# Epoch 1
# -----
# Batch 0, Loss: 2.335469, Accuracy: 7.81% [ 0/60000]
# Batch 100, Loss: 0.728375, Accuracy: 74.15% [ 6400/60000]
# Batch 200, Loss: 0.615916, Accuracy: 77.78% [12800/60000]
# Batch 300, Loss: 0.568747, Accuracy: 79.64% [19200/60000]
# Batch 400, Loss: 0.543865, Accuracy: 80.47% [25600/60000]
# Batch 500, Loss: 0.521335, Accuracy: 81.20% [32000/60000]
# Batch 600, Loss: 0.505527, Accuracy: 81.74% [38400/60000]
# Batch 700, Loss: 0.489401, Accuracy: 82.29% [44800/60000]
# Batch 800, Loss: 0.479022, Accuracy: 82.68% [51200/60000]
# Batch 900, Loss: 0.470834, Accuracy: 82.94% [57600/60000]
# =====
# Epoch 2
# -----
# Batch 0, Loss: 0.239584, Accuracy: 93.75% [ 0/60000]
# Batch 100, Loss: 0.368888, Accuracy: 86.46% [ 6400/60000]
# Batch 200, Loss: 0.362973, Accuracy: 86.46% [12800/60000]
# Batch 300, Loss: 0.358299, Accuracy: 86.81% [19200/60000]
# Batch 400, Loss: 0.360760, Accuracy: 86.71% [25600/60000]
# Batch 500, Loss: 0.359864, Accuracy: 86.75% [32000/60000]
# Batch 600, Loss: 0.359880, Accuracy: 86.81% [38400/60000]
# Batch 700, Loss: 0.358785, Accuracy: 86.84% [44800/60000]
# Batch 800, Loss: 0.355679, Accuracy: 86.99% [51200/60000]
# Batch 900, Loss: 0.352759, Accuracy: 87.08% [57600/60000]
# =====
# ...
```

```
# ...  
# ...  
# Done!
```

To evaluate the model on the test data, it follows the same idea of training, but it doesn't need to learn, so we focus on the evaluation and the accuracy metric:

```
def test(dataloader, model, loss_function):  
    size = len(dataloader.dataset)  
    num_batches = len(dataloader)  
    test_loss, correct = 0, 0  
    model.eval()  
  
    with torch.no_grad():  
        for batch, (x, y) in enumerate(dataloader):  
            logits = model(x)  
            loss = loss_function(logits, y)  
            test_loss += loss.item()  
            correct += (logits.argmax(1) == y).sum().item()  
  
    test_loss /= num_batches  
    correct /= size  
    print(f"Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f}")  
  
test(test_dataloader, model, loss_function)  
# Accuracy: 87.4%, Avg loss: 0.350322
```

Building and training a CNN model for a horse or human classification

We're going to work on a dataset of images of horses and humans and the task is to classify if it's one or the other. For this, we're going to use the Convolutional Neural Network (CNN) for the first time in PyTorch.

First, let's build the dataset, starting with the image download of the training data:

```
import urllib.request
import zipfile

url = "https://storage.googleapis.com/learning-datasets/horse-or-human.zip"
file_name = "horse-or-human.zip"
training_dir = "horse-or-human/training/"
urllib.request.urlretrieve(url, file_name)

zip_ref = zipfile.ZipFile(file_name, "r")
zip_ref.extractall(training_dir)
zip_ref.close()
```

We should do the same for the validation data:

```
url = "https://storage.googleapis.com/learning-datasets/validation-horse-or-human.zip"
file_name = "validation-horse-or-human.zip"
validation_dir = 'horse-or-human/validation/'
urllib.request.urlretrieve(url, file_name)

zip_ref = zipfile.ZipFile(file_name, 'r')
zip_ref.extractall(validation_dir)
zip_ref.close()
```

To build the dataset, we are going to use the `ImageFolder` dataset object from PyTorch. But, in the process of building it, we need to set up the transforms for the training and the validation data.

Let's start with the training first.

```
train_transform = transforms.Compose([
    transforms.Resize([150, 150]),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(20),
    transforms.RandomResizedCrop(150),
    transforms.RandomAffine(
        degrees=0,
        translate=(0.2, 0.2),
        scale=(0.8, 1.2),
        shear=20
```

```
),  
transforms.ToTensor(),  
transforms.Normalize(mean=[.5, .5, .5], std=[.5, .5, .5])  
)
```

Here we have the preprocessing step that resizes, crops, transforms into tensors, and normalizes the data. And then, we have the augmentation process that flips and randomly rotates, translates, and scales the data.

With that, we can use it to build the dataset using `ImageFolder`:

```
train_dataset = datasets.ImageFolder(root=training_dir, transform=transform)
```

With that, we can build the dataloader for training:

```
train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

Batch size of 32 and shuffling the data.

We should do the same thing for the validation data, but with a subtle change. For the transformations, we should only consider using the preprocessing step, as the augmentation process is mainly used to train the model.

```
validation_transform = transforms.Compose([  
    transforms.Resize([150, 150]),  
    transforms.ToTensor(),  
    transforms.Normalize(mean=[.5, .5, .5], std=[.5, .5, .5])  
)
```

For the validation dataset, it's pretty much the same thing:

```
validation_dataset = datasets.ImageFolder(root=validation_dir, transform=transfo
```

And for the dataloader, it's quite similar, but shuffling is unnecessary:

```
validation_dataloader = DataLoader(validation_dataset, batch_size=32, shuffle=False)
```

In the end, the format of the dataset will come in this shape: [32, 3, 150, 150]

- batch_size = 32
- channels = 3 (RGB)
- height = 150
- width = 150

For the model, we are going to build convolution layers, more specifically, 3 convolution layers with ReLU and pooling layers, and finish with two fully connected layers with dropout, and a sigmoid function to classify the data.

```

class HorsesHumansCNN(nn.Module):
    def __init__(self):
        super(HorsesHumansCNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 18 * 18, 512)
        self.dropout = nn.Dropout(0.25)
        self.fc2 = nn.Linear(512, 1)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 18 * 18) # flatten
        x = self.fc1(x)
        x = self.dropout(x)
        x = self.fc2(x)
        x = torch.sigmoid(x)
        return x

```

If we create the model and visualize the summary, we have this:

```

model = HorsesHumansCNN()
summary(model, input_size=(3, 150, 150))

```

```

# -----
#          Layer (type)                Output Shape          Param #
# =====
#          Conv2d-1                    [-1, 16, 150, 150]       448
#          MaxPool2d-2                  [-1, 16, 75, 75]        0
#          Conv2d-3                     [-1, 32, 75, 75]       4,640
#          MaxPool2d-4                  [-1, 32, 37, 37]        0
#          Conv2d-5                     [-1, 64, 37, 37]      18,496
#          MaxPool2d-6                  [-1, 64, 18, 18]        0
#          Linear-7                     [-1, 512]             10,617,344
#          Dropout-8                   [-1, 512]              0
#          Linear-9                     [-1, 1]                513
# =====

```

```

# Total params: 10,641,441
# Trainable params: 10,641,441
# Non-trainable params: 0
# -----
# Input size (MB): 0.26
# Forward/backward pass size (MB): 5.98
# Params size (MB): 40.59
# Estimated Total Size (MB): 46.83
# -----

```

A 10.6 million-parameter CNN model.

As we did before, we should have the loss function and the optimizer to train the model.

```

loss_function = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

```

And then, we have the training implementation:

```

def train(epochs, train_dataloader, validation_dataloader, model, loss_function,
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for images, labels in train_dataloader:
            optimizer.zero_grad()
            outputs = model(images).view(-1)
            loss = loss_function(outputs, labels.float())
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            predictions = outputs > 0.5
            total += labels.size(0)
            correct += (predictions == labels).sum().item()

        print(f'Epoch {epoch + 1}, Loss: {running_loss / len(train_dataloader)}')
        print(f'Training Set Accuracy: {100 * correct / total}%')



```

```
model.eval()

with torch.no_grad():
    correct = 0
    total = 0

    for images, labels in validation_dataloader:
        outputs = model(images).view(-1)
        predictions = outputs > 0.5
        total += labels.size(0)
        correct += (predictions == labels).sum().item()

    print(f'Validation Set Accuracy: {100 * correct / total}%')
```

 Twitter ·  Github



TK • Newsletter

All about crafting software, ML/AI, and self-development.

By TK's Journey



Subscribe

By subscribing you agree to [Substack's Terms of Use](#), [our Privacy Policy](#) and [our Information collection notice](#)

 substack

[#python](#) [#data_science](#) [#machine_learning](#)

Copyright © 2025 TK

[Home](#) | [Writing](#) | [Support](#) | [Code](#) | [RSS](#)