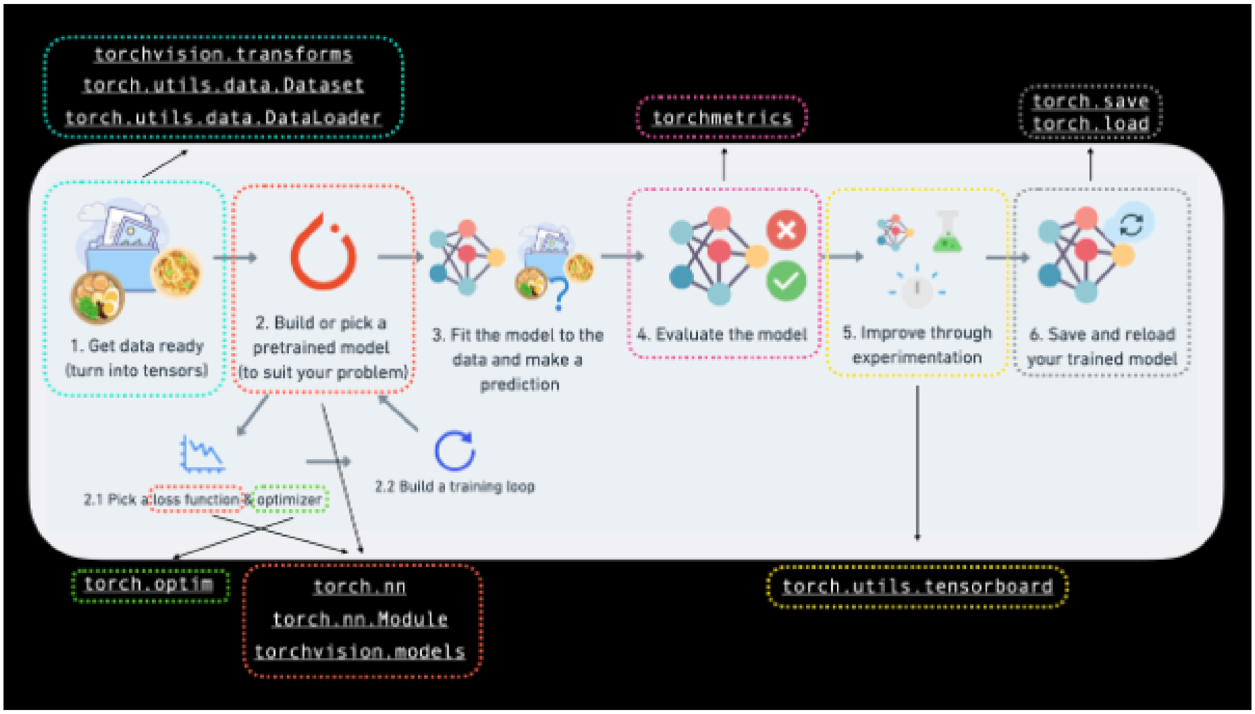
03. PyTorch Computer Vision



Topic	Contents
0. Computer vision libraries in PyTorch	PyTorch has a bunch of built-in helpful computer vision libraries, let's check them out.
1. Load data	To practice computer vision, we'll start with some images of different pieces of clothing from FashionMNIST.
2. Prepare data	We've got some images, let's load them in with a PyTorch `DataLoader` so we can use them with our training loop.
3. Model 0: Building a baseline model	Here we'll create a multi-class classification model to learn patterns in the data, we'll also choose a **loss function**, **optimizer** and build a **training loop**.
4. Making predictions and evaluting model 0	Let's make some predictions with our baseline model and evaluate them.
5. Setup device agnostic code for future models	It's best practice to write device-agnostic code, so let's set it up.

6. Model 1: Adding non-linearity	Experimenting is a large part of machine learning, let's try and improve upon our baseline model by adding non-linear layers.
7. Model 2: Convolutional Neural Network (CNN)	Time to get computer vision specific and introduce the powerful convolutional neural network architecture.
8. Comparing our models	We've built three different models, let's compare them.
9. Evaluating our best model	Let's make some predictons on random images and evaluate our best model.
10. Making a confusion matrix	A confusion matrix is a great way to evaluate a classification model, let's see how we can make one.
11. Saving and loading the best performing model	Since we might want to use our model for later, let's save it and make sure it loads back in correctly.

0. Computer vision libraries in PyTorch

PyTorch module	What does it do?
`torchvision`	Contains datasets, model architectures and image transformations often used for computer vision problems.
`torchvision.datasets`	Here you'll find many example computer vision datasets for a range of problems from image classification, object detection, image captioning, video classification and more. It also contains a series of base classes for making custom datasets.
`torchvision.models`	This module contains well-performing and commonly used computer vision model architectures implemented in PyTorch, you can use these with your own problems.
`torchvision.transforms`	Often images need to be transformed (turned into numbers/processed/augmented) before being used with a model, common image transformations are found here.
`torch.utils.data.Dataset`	Base dataset class for PyTorch.
`torch.utils.data.DataLoader`	Creates a Python iterable over a dataset (created with `torch.utils.data.Dataset`).

1. Getting a dataset

PyTorch has a bunch of common computer vision datasets stored in `torchvision.datasets`.

Including FashionMNIST in `torchvision.datasets.FashionMNIST()`.

To download it, we provide the following parameters:

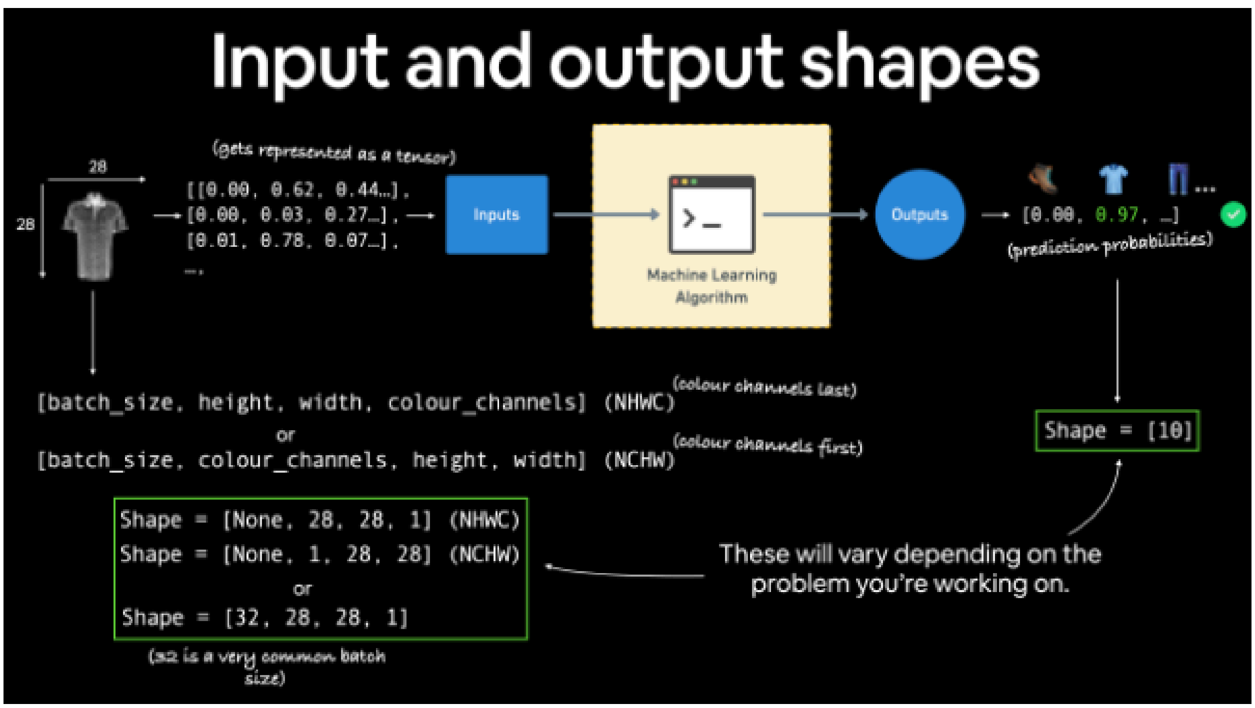
- `root: str` - which folder do you want to download the data to?
- `train: Bool` - do you want the training or test split?
- `download: Bool` - should the data be downloaded?
- `transform: torchvision.transforms` - what transformations would you like to do on the data?
- `target_transform` - you can transform the targets (labels) if you like too.

```
# Setup training data
train_data = datasets.FashionMNIST(
    root="data", # where to download data to?
```

```
    train=True, # get training data
    download=True, # download data if it doesn't exist on disk
    transform=ToTensor(), # images come as PIL format, we want to turn into Torch tensors
    target_transform=None # you can transform labels as well
)

# Setup testing data
test_data = datasets.FashionMNIST(
    root="data",
    train=False, # get test data
    download=True,
    transform=ToTensor()
)
```

1.1 Input and output shapes of a computer vision model



Various problems will have various input and output shapes. But the premise remains: encode data into numbers, build a model to find patterns in those numbers, convert those patterns into something meaningful.

2. Prepare DataLoader

The next step is to prepare it with a `torch.utils.data.DataLoader` or `DataLoader` for short.

The `DataLoader` does what you think it might do.

It helps load data into a model.

For training and for inference.

It turns a large `Dataset` into a Python iterable of smaller chunks.

These smaller chunks are called **batches** or **mini-batches** and can be set by the `batch_size` parameter.

With **mini-batches** (small portions of the data), gradient descent is performed more often per epoch (once per mini-batch rather than once per epoch).


```
from torch.utils.data import DataLoader

# Setup the batch size hyperparameter

BATCH_SIZE = 32

# Turn datasets into iterables (batches)

train_dataloader = DataLoader(train_data, # dataset to turn into iterable

    batch_size=BATCH_SIZE, # how many samples per batch?

    shuffle=True # shuffle data every epoch?

)
```

```
test_dataloader = DataLoader(test_data,
    batch_size=BATCH_SIZE,
    shuffle=False # don't necessarily have to shuffle the testing data
)

# Let's check out what we've created
print(f"Dataloaders: {train_dataloader, test_dataloader}")
print(f"Length of train dataloader: {len(train_dataloader)} batches of {BATCH_SIZE}")
print(f"Length of test dataloader: {len(test_dataloader)} batches of {BATCH_SIZE}")

# Check out what's inside the training dataloader
train_features_batch, train_labels_batch = next(iter(train_dataloader))
train_features_batch.shape, train_labels_batch.shape
```

3. Model 0: Build a baseline model

Time to build a **baseline model** by subclassing `nn.Module`.

A **baseline model** is one of the simplest models you can imagine.

You use the baseline as a starting point and try to improve upon it with subsequent, more complicated models.

Our baseline will consist of two `nn.Linear()` layers.

We've done this in a previous section but there's going to one slight difference.

Because we're working with image data, we're going to use a different layer to start things off.

And that's the `nn.Flatten()` layer.

`nn.Flatten()` compresses the dimensions of a tensor into a single vector.

```
from torch import nn
class FashionMNISTModelV0(nn.Module):
    def __init__(self, input_shape: int, hidden_units: int, output_shape: int):
        super().__init__()
        self.layer_stack = nn.Sequential(
            nn.Flatten(), # neural networks like their inputs in vector form
```

```
            nn.Linear(in_features=input_shape, out_features=hidden_units), # in_features = number of features in a data sample (784 pixels)
            nn.Linear(in_features=hidden_units, out_features=output_shape)
        )

    def forward(self, x):
        return self.layer_stack(x)



torch.manual_seed(42)

# Need to setup model with input parameters
model_0 = FashionMNISTModelV0(input_shape=784, # one for every pixel (28x28)
    hidden_units=10, # how many units in the hiden layer
    output_shape=len(class_names) # one for every class
)
model_0.to("cpu") # keep model on CPU to begin with
```

3.1 Setup loss, optimizer and evaluation metrics

```
# Import accuracy metric
from helper_functions import accuracy_fn # Note: could also use torchmetrics.Accuracy(task = 'multiclass', num_classes=len(class_names)).to(device)

# Setup loss function and optimizer
loss_fn = nn.CrossEntropyLoss() # this is also called "criterion"/"cost function" in some places
optimizer = torch.optim.SGD(params=model_0.parameters(), lr=0.1)
```

3.2 Creating a function to time our experiments

```
from timeit import default_timer as timer
def print_train_time(start: float, end: float, device: torch.device = None):
    """Prints difference between start and end time.

    Args:
        start (float): Start time of computation (preferred in timeit format).
        end (float): End time of computation.
        device ([type], optional): Device that compute is running on. Defaults to None.
```

```
    Returns:
        float: time between start and end in seconds (higher is longer).
    """
    total_time = end - start
    print(f"Train time on {device}: {total_time:.3f} seconds")
    return total_time
```

3.3 Creating a training loop and training a model on batches of data

Since we're computing on batches of data, our loss and evaluation metrics will be calculated **per batch** rather than across the whole dataset.

This means we'll have to divide our loss and accuracy values by the number of batches in each dataset's respective dataloader.

Let's step through it:

1. Loop through epochs.
2. Loop through training batches, perform training steps, calculate the train loss *per batch*.
3. Loop through testing batches, perform testing steps, calculate the test loss *per batch*.
4. Print out what's happening.
5. Time it all (for fun).

```
# Import tqdm for progress bar
from tqdm.auto import tqdm

# Set the seed and start the timer
torch.manual_seed(42)
train_time_start_on_cpu = timer()

# Set the number of epochs (we'll keep this small for faster training times)
epochs = 3

# Create training and testing loop
for epoch in tqdm(range(epochs)):
    print(f"Epoch: {epoch}\n-------")
    ### Training
    train_loss = 0
    # Add a loop to loop through training batches
    for batch, (X, y) in enumerate(train_dataloader):
        model_0.train()
        # 1. Forward pass
        y_pred = model_0(X)
```

```
        # 2. Calculate loss (per batch)
        loss = loss_fn(y_pred, y)
        train_loss += loss # accumulatively add up the loss per epoch

        # 3. Optimizer zero grad
        optimizer.zero_grad()

        # 4. Loss backward
        loss.backward()

        # 5. Optimizer step
        optimizer.step()

        # Print out how many samples have been seen
        if batch % 400 == 0:
            print(f"Looked at {batch * len(X)}/{len(train_dataloader.dataset)} samples")

    # Divide total train loss by length of train dataloader (average loss per batch per
epoch)
    train_loss /= len(train_dataloader)

    ### Testing
    # Setup variables for accumulatively adding up loss and accuracy
    test_loss, test_acc = 0, 0
    model_0.eval()
    with torch.inference_mode():
        for X, y in test_dataloader:
            # 1. Forward pass
            test_pred = model_0(X)

            # 2. Calculate loss (accumatively)
            test_loss += loss_fn(test_pred, y) # accumulatively add up the loss per epoch

            # 3. Calculate accuracy (preds need to be same as y_true)
            test_acc += accuracy_fn(y_true=y, y_pred=test_pred.argmax(dim=1))

        # Calculations on test metrics need to happen inside torch.inference_mode()
        # Divide total test loss by length of test dataloader (per batch)
        test_loss /= len(test_dataloader)

        # Divide total accuracy by length of test dataloader (per batch)
        test_acc /= len(test_dataloader)

    ## Print out what's happening
```

```
    print(f"\nTrain loss: {train_loss:.5f} | Test loss: {test_loss:.5f}, Test acc: {test_acc:.2f}%\n")

# Calculate training time
train_time_end_on_cpu = timer()
total_train_time_model_0 = print_train_time(start=train_time_start_on_cpu,
                                           end=train_time_end_on_cpu,
                                           device=str(next(model_0.parameters()).device))
```

4. Make predictions and get Model 0 results

5. Setup device agnostic-code (for using a GPU if there is one)

6. Model 1: Building a better model with non-linearity

```
# Create a model with non-linear and linear layers
class FashionMNISTModelV1(nn.Module):
    def __init__(self, input_shape: int, hidden_units: int, output_shape: int):
        super().__init__()
        self.layer_stack = nn.Sequential(
            nn.Flatten(), # flatten inputs into single vector
            nn.Linear(in_features=input_shape, out_features=hidden_units),
            nn.ReLU(),
            nn.Linear(in_features=hidden_units, out_features=output_shape),
            nn.ReLU()
        )

    def forward(self, x: torch.Tensor):
        return self.layer_stack(x)
```

6.1 Setup loss, optimizer and evaluation metrics

```
from helper_functions import accuracy_fn
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(params=model_1.parameters(),
                            lr=0.1)
```

6.2 Functionizing training and test loops

7. Model 2: Building a Convolutional Neural Network (CNN)

CNN's are known for their capabilities to find patterns in visual data.

And since we're dealing with visual data, let's see if using a CNN model can improve upon our baseline.

The CNN model we're going to be using is known as TinyVGG from the CNN Explainer website.

It follows the typical structure of a convolutional neural network:

`Input layer -> [Convolutional layer -> activation layer -> pooling layer] -> Output layer`

Where the contents of `[Convolutional layer -> activation layer -> pooling layer]` can be upscaled and repeated multiple times, depending on requirements.

Problem type	**Model to use (generally)**	**Code example**
Structured data (Excel spreadsheets, row and column data)	Gradient boosted models, Random Forests, XGBoost	`sklearn.ensemble`, XGBoost library
Unstructured data (images, audio, language)	Convolutional Neural Networks, Transformers	`torchvision.models`, HuggingFace Transformers

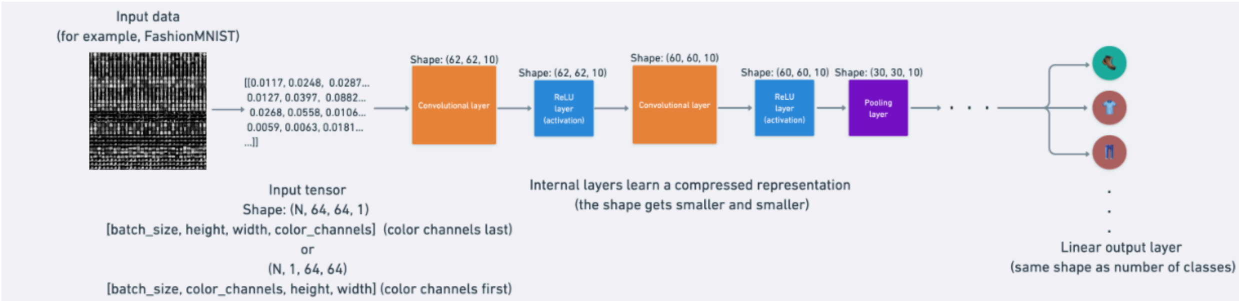
To do so, we'll leverage the `nn.Conv2d()` and `nn.MaxPool2d()` layers from `torch.nn`.

```
# Create a convolutional neural network
class FashionMNISTModelV2(nn.Module):
    """
    Model architecture copying TinyVGG from:
    https://poloclub.github.io/cnn-explainer/
    """
    def __init__(self, input_shape: int, hidden_units: int, output_shape: int):
        super().__init__()
        self.block_1 = nn.Sequential(
            nn.Conv2d(in_channels=input_shape,
                      out_channels=hidden_units,
                      kernel_size=3, # how big is the square that's going over the image?
```

```
                    stride=1, # default
                    padding=1),# options = "valid" (no padding) or "same" (output has same shape
as input) or int for specific number
            nn.ReLU(),
            nn.Conv2d(in_channels=hidden_units,
                    out_channels=hidden_units,
                    kernel_size=3,
                    stride=1,
                    padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,
                       stride=2) # default stride value is same as kernel_size
        )
        self.block_2 = nn.Sequential(
            nn.Conv2d(hidden_units, hidden_units, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(hidden_units, hidden_units, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            # Where did this in_features shape come from?
            # It's because each layer of our network compresses and changes the shape of our
inputs data.
            nn.Linear(in_features=hidden_units*7*7,
                    out_features=output_shape)
        )

    def forward(self, x: torch.Tensor):
        x = self.block_1(x)
        # print(x.shape)
        x = self.block_2(x)
        # print(x.shape)
        x = self.classifier(x)
        # print(x.shape)
        return x

torch.manual_seed(42)
model_2 = FashionMNISTModelV2(input_shape=1,
    hidden_units=10,
    output_shape=len(class_names)).to(device)
model_2
```


Input data
(for example, FashionMNIST)
[[0.0117, 0.0248, 0.0287...
0.0127, 0.0397, 0.0882...
0.0268, 0.0558, 0.0106...
0.0059, 0.0063, 0.0181...
...]]
Input tensor
Shape: (N, 64, 64, 1)
[batch_size, height, width, color_channels] (color channels last)
or
(N, 1, 64, 64)
[batch_size, color_channels, height, width] (color channels first)
Shape: (62, 62, 10)
Convolutional layer
Shape: (62, 62, 10)
ReLU layer (activation)
Shape: (60, 60, 10)
Convolutional layer
Shape: (60, 60, 10)
ReLU layer (activation)
Shape: (30, 30, 10)
Pooling layer
Internal layers learn a compressed representation
(the shape gets smaller and smaller)
Linear output layer
(same shape as number of classes)