PDF Report

Muhammad Amirul Arsyad Arrayyan

1103204207

Chapter 00

1. PyTorch Overview:

PyTorch adalah framework open source untuk machine learning dan deep learning.

Digunakan untuk memanipulasi dan memproses data serta menulis algoritma machine learning menggunakan kode Python.

Pengguna PyTorch:

Digunakan oleh perusahaan-perusahaan teknologi besar seperti Meta (Facebook), Tesla, dan Microsoft, serta perusahaan riset kecerdasan buatan seperti OpenAI.

Aplikasi PyTorch:

Digunakan di berbagai industri, termasuk pertanian untuk computer vision pada traktor.

Alasan Menggunakan PyTorch:

Disukai oleh peneliti machine learning.

PyTorch adalah framework deep learning yang paling banyak digunakan menurut Papers With Code hingga Februari 2022.

Menangani percepatan GPU dan memudahkan pengembangan dengan fokus pada manipulasi data dan penulisan algoritma.

2. Tensors:

Tensors adalah blok dasar machine learning dan deep learning, digunakan untuk merepresentasikan data dalam bentuk numerik.

Membuat tensors untuk mewakili berbagai jenis data seperti gambar, kata, dan tabel.

Manipulasi tensors dalam algoritma machine learning, termasuk penjumlahan, perkalian, dan operasi lainnya.

Penanganan bentuk (shape) tensors dan pengindeksan pada tensors.

Kesalahan umum seperti bentuk yang tidak sesuai dapat terjadi dalam machine learning.

Importing PyTorch:

Penggunaan PyTorch dimulai dengan mengimpor library dan memeriksa versi yang digunakan.

In [1]:
```
import torch
torch.__version__
```

Out[1]: '2.1.0+cu121'

Versi PyTorch 2.1.0+, lebih update dari contoh yang diberikan

Creating Tensors:

Scalar, vector, matrix, dan tensor adalah bentuk-bentuk tensors dengan berbagai dimensi.

Tensors dapat dibuat dengan menggunakan fungsi **torch.tensor()**.

Scalar adalah bilangan tunggal dan dalam istilah tensor merupakan tensor berdimensi nol.

In [2]:
```
# Scalar
scalar = torch.tensor(7)
scalar
```

Out[2]: tensor(7)

Vektor adalah tensor berdimensi tunggal tetapi dapat memuat banyak bilangan.

In [5]:
```
# Vector
vector = torch.tensor([7, 7])
vector
```

Out[5]: tensor([7, 7])

Matriks sama fleksibelnya dengan vektor, hanya saja matriks mempunyai dimensi tambahan.

In [8]:
```
# Matrix
MATRIX = torch.tensor([[7, 8],
                       [9, 10]])
MATRIX
```

Out[8]:
```
tensor([[ 7,  8],
        [ 9, 10]])
```

Tensor dapat mewakili hampir semua hal. Pada contoh dibawah adalah angka penjualan toko steak dan mentega almond. Tensor menunjukkan hari, penjualan steak, dan penjualan mentega almond.

In [11]:
```
# Tensor
TENSOR = torch.tensor([[[1, 2, 3],
                        [3, 6, 9],
                        [2, 4, 5]]])
TENSOR
```

Out[11]:
```
tensor([[[1, 2, 3],
         [3, 6, 9],
         [2, 4, 5]]])
```

Random Tensors:

Model machine learning sering dimulai dengan tensors acak dan disesuaikan selama pelatihan.

Fungsi **torch.rand()** digunakan untuk membuat tensor acak.

```
In [14]:
# Create a random tensor of size (3, 4)
random_tensor = torch.rand(size=(3, 4))
random_tensor, random_tensor.dtype
```

```
Out[14]: (tensor([[0.3617, 0.2029, 0.3035, 0.1288],
                  [0.8978, 0.0664, 0.6195, 0.0327],
                  [0.0509, 0.5216, 0.9160, 0.2458]]),
          torch.float32)
```

Fleksibilitas **torch.rand()** membuat kita dapat mengatur **size** sesuai keinginan

```
In [15]:
# Create a random tensor of size (224, 224, 3)
random_image_size_tensor = torch.rand(size=(224, 224, 3))
random_image_size_tensor.shape, random_image_size_tensor.ndim
```

```
Out[15]: (torch.Size([224, 224, 3]), 3)
```

Zeros and Ones:

Tensors dapat diisi dengan nilai nol atau satu menggunakan fungsi **torch.zeros()** dan **torch.ones()**.

Dimulai dengan **torch.zeros()**, **size** tensor dibuat sebesar (3,4).

```
In [16]:
# Create a tensor of all zeros
zeros = torch.zeros(size=(3, 4))
zeros, zeros.dtype
```

```
Out[16]: (tensor([[0., 0., 0., 0.],
                  [0., 0., 0., 0.],
                  [0., 0., 0., 0.]]),
          torch.float32)
```

Lalu **torch.ones()** dengan **size** (3,4) juga.

```
In [17]:
# Create a tensor of all ones
ones = torch.ones(size=(3, 4))
ones, ones.dtype
```

```
Out[17]: (tensor([[1., 1., 1., 1.],
                  [1., 1., 1., 1.],
                  [1., 1., 1., 1.]]),
          torch.float32)
```

Creating a Range and Tensors Like:

Fungsi **torch.arange()** digunakan untuk membuat tensor yang berisi rentang nilai. Rentang nilai berisikan **(start,end,step)**, start range mulai, end range akhir dan step menyebutkan step diantara nilai/value.

In [18]:
```
# Use torch.arange(), torch.range() is deprecated
zero_to_ten_deprecated = torch.range(0, 10) # Note: this may return an error in the future

# Create a range of values 0 to 10
zero_to_ten = torch.arange(start=0, end=10, step=1)
zero_to_ten
```

```
<ipython-input-18-a09072c806d9>:2: UserWarning: torch.range is deprecated and will be removed in a future release because its behavior is inconsistent with Python's range builtin. Instead, use torch.arange, which produces values in [start, end).
  zero_to_ten_deprecated = torch.range(0, 10) # Note: this may return an error in the future
```

Out[18]:
```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Fungsi **torch.zeros_like()** dan **torch.ones_like()** digunakan untuk membuat tensor dengan bentuk yang sama seperti tensor lainnya.

In [19]:
```
# Can also create a tensor of zeros similar to another tensor
ten_zeros = torch.zeros_like(input=zero_to_ten) # will have same shape
ten_zeros
```

Out[19]:
```
tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Tensor Datatypes:

Terdapat berbagai jenis datatypes untuk tensors, termasuk float32, float16, dan integer dengan berbagai presisi.

Datatypes mempengaruhi kecepatan dan akurasi perhitungan dalam machine learning.

Device Issues:

Masalah umum dalam PyTorch melibatkan tipe datatypes dan device tensors.

Tensors harus memiliki datatype dan device yang sesuai untuk operasi yang efisien.

3. Getting information from tensors:

Terdapat 3 atribut yang biasa dipakai yaitu **shape dtype** dan **device**.
shape meminta shape dari tensor.
dtype menanyakan di tipe data apa elemen dalam tensor disimpan.
device menanyakan perangkat mana yang menyimpan tensor.

In [22]:
```
# Create a tensor
some_tensor = torch.rand(3, 4)

# Find out details about it
print(some_tensor)
print(f"Shape of tensor: {some_tensor.shape}")
print(f"Datatype of tensor: {some_tensor.dtype}")
print(f"Device tensor is stored on: {some_tensor.device}") # will default to CPU
```

```
tensor([[0.1843, 0.6883, 0.7791, 0.9722],
        [0.9420, 0.9171, 0.3786, 0.5284],
        [0.2892, 0.3967, 0.9849, 0.4442]])
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

4. Manipulating tensors (tensor operations):

Deep learning melibatkan representasi data sebagai tensor dan melakukan berbagai operasi untuk mempelajari pola. Operasi dasar melibatkan addition, subtraction, multiplication (element-wise), division, dan matrix multiplication.

Basic operations:

In [23]:
```
# Create a tensor of values and add a number to it
tensor = torch.tensor([1, 2, 3])
tensor + 10
```

Out[23]: tensor([11, 12, 13])

In [24]:
```
# Multiply it by 10
tensor * 10
```

Out[24]: tensor([10, 20, 30])

In [25]:
```
# Tensors don't change unless reassigned
tensor
```

Out[25]: tensor([1, 2, 3])

Let's subtract a number and this time we'll reassign the `tensor` variable.

In [26]:
```
# Subtract and reassign
tensor = tensor - 10
tensor
```

Out[26]: tensor([-9, -8, -7])

In [27]:
```
# Add and reassign
tensor = tensor + 10
tensor
```

Out[27]: tensor([1, 2, 3])

In [28]:
```
# Can also use torch functions
torch.multiply(tensor, 10)
```

Out[28]: tensor([10, 20, 30])

In [29]:
```
# Original tensor is still unchanged
tensor
```

Out[29]: tensor([1, 2, 3])

In [30]:
```
# Element-wise multiplication (each element multiplies its equivalent, index 0->0, 1->1, 2->2)
print(tensor, "*", tensor)
print("Equals:", tensor * tensor)
```

```
tensor([1, 2, 3]) * tensor([1, 2, 3])
Equals: tensor([1, 4, 9])
```

Addition

```
tensor + 10
```

Menambahkan nilai tertentu ke setiap elemen tensor.

Multiplication

```
tensor * 10
```

Mengalikan setiap elemen tensor dengan nilai tertentu.

Reassigning Tensors

```
tensor = tensor - 10
```

Tensor tetap tidak berubah kecuali diassign ulang. Operasi seperti pengurangan dapat dilakukan dan hasilnya di-assign kembali ke variabel tensor.

Built-in Functions

```
torch.multiply(tensor, 10)
```

PyTorch menyediakan fungsi seperti **torch.add()** dan **torch.mul()** untuk penambahan dan perkalian, secara berturut-turut.

Element-wise Operations

```
tensor * tensor
```

Simbol operator seperti ***** umumnya digunakan untuk operasi berdasarkan elemen, di mana setiap elemen dari satu tensor dikombinasikan dengan elemennya yang setara di tensor lain.

Matrix multiplication:

Perkalian matriks adalah operasi fundamental dalam algoritma pembelajaran mesin dan pembelajaran mendalam, seperti jaringan saraf. PyTorch mengimplementasikan fungsi perkalian matriks menggunakan metode **torch.matmul()**.

Aturan utama perkalian matriks:

- Dimensi dalam matriks yang bersinggungan harus sesuai untuk perkalian.
- Matriks hasil memiliki bentuk dari dimensi luar.

In [31]:
```
import torch
tensor = torch.tensor([1, 2, 3])
tensor.shape
```

Out[31]:
```
torch.Size([3])
```

```
In [32]:
# Element-wise matrix multiplication
tensor * tensor

Out[32]: tensor([1, 4, 9])

In [33]:
# Matrix multiplication
torch.matmul(tensor, tensor)

Out[33]: tensor(14)

In [34]:
# Can also use the "@" symbol for matrix multiplication, though not recommended
tensor @ tensor

Out[34]: tensor(14)

In [35]:
%%time
# Matrix multiplication by hand
# (avoid doing operations with for loops at all cost, they are computationally expensive)
value = 0
for i in range(len(tensor)):
  value += tensor[i] * tensor[i]
value

CPU times: user 1.24 ms, sys: 45 µs, total: 1.28 ms
Wall time: 1.28 ms
Out[35]: tensor(14)

In [36]:
%%time
torch.matmul(tensor, tensor)

CPU times: user 438 µs, sys: 79 µs, total: 517 µs
Wall time: 438 µs
Out[36]: tensor(14)
```

```
torch.matmul(tensor, tensor)
```

Gunakan metode **torch.matmul()** atau simbol "**@**" untuk perkalian matriks.

```
tensor * tensor
```

Perkalian elemen melibatkan perkalian elemen yang sesuai dalam dua tensor.

```
torch.matmul(tensor, tensor)
```

Perkalian matriks melakukan perkalian titik dari elemen yang sesuai dan menjumlahkan hasilnya.

```
torch.matmul(tensor, tensor)
```

Metode torch.matmul() bawaan direkomendasikan untuk perkalian matriks karena efisiensinya secara komputasional.

Chapter 01 PyTorch Workflow Fundamentals

Import

In [2]:
```
import torch
from torch import nn # nn contains all of PyTorch's building blocks for neural networks
import matplotlib.pyplot as plt

# Check PyTorch version
torch.__version__
```

Out[2]: '2.1.0+cu121'

Meng import nn untuk building blocks dari neural network dan matplotlib.

1. Data (preparing and loading)

Dalam machine learning, data dapat berupa berbagai bentuk, termasuk tabel angka (seperti spreadsheet Excel), gambar, video (YouTube memiliki banyak data!), file audio seperti lagu atau podcast, struktur protein, teks, dan lainnya. Proses machine learning melibatkan dua tahap:

1. mengubah data menjadi set angka yang representatif dan

2. memilih atau membangun model untuk memahami representasi tersebut sebaik mungkin.

Terkadang langkah satu dan dua dapat dilakukan secara bersamaan.

Untuk ilustrasi, contoh sederhana dari regresi linear digunakan untuk membuat data sintetis. Parameter-parameter yang diketahui (bobot dan bias) ditentukan, dan PyTorch digunakan untuk menghasilkan titik data berdasarkan parameter-parameter ini.

In [3]:
```
# Create *known* parameters
weight = 0.7
bias = 0.3

# Create data
start = 0
end = 1
step = 0.02
X = torch.arange(start, end, step).unsqueeze(dim=1)
y = weight * X + bias

X[:10], y[:10]
```

Out[3]:
```
(tensor([[0.0000],
         [0.0200],
         [0.0400],
         [0.0600],
         [0.0800],
         [0.1000],
         [0.1200],
         [0.1400],
         [0.1600],
         [0.1800]]),
 tensor([[0.3000],
         [0.3140],
         [0.3280],
         [0.3420],
         [0.3560],
         [0.3700],
         [0.3840],
         [0.3980],
         [0.4120],
         [0.4260]]))
```

Tensor X dan y yang dihasilkan mewakili fitur dan label.

Split data into training and test sets

Pembagian dataset menjadi set pelatihan dan pengujian adalah salah satu langkah paling penting dalam proyek machine learning. Setiap pembagian memiliki tujuan khusus, yaitu:

- **Set Pelatihan:** Digunakan untuk model belajar (sekitar 60-80% dari total data).
- **Set Pengujian:** Digunakan untuk mengevaluasi model (sekitar 10-20% dari total data).

Membuat Pembagian Pelatihan dan Pengujian:

Dalam contoh ini, hanya dilakukan pembagian menjadi set pelatihan dan pengujian. Proses ini dilakukan dengan membagi tensor X dan y menjadi dua bagian, masing-masing untuk pelatihan dan pengujian. Panjang setiap set (pelatihan dan pengujian) ditampilkan untuk memverifikasi pembagian yang dilakukan.

In [4]:
```
# Create train/test split
train_split = int(0.8 * len(X)) # 80% of data used for training set, 20% for testing
X_train, y_train = X[:train_split], y[:train_split]
X_test, y_test = X[train_split:], y[train_split:]

len(X_train), len(y_train), len(X_test), len(y_test)
```

Out[4]: (40, 40, 10, 10)

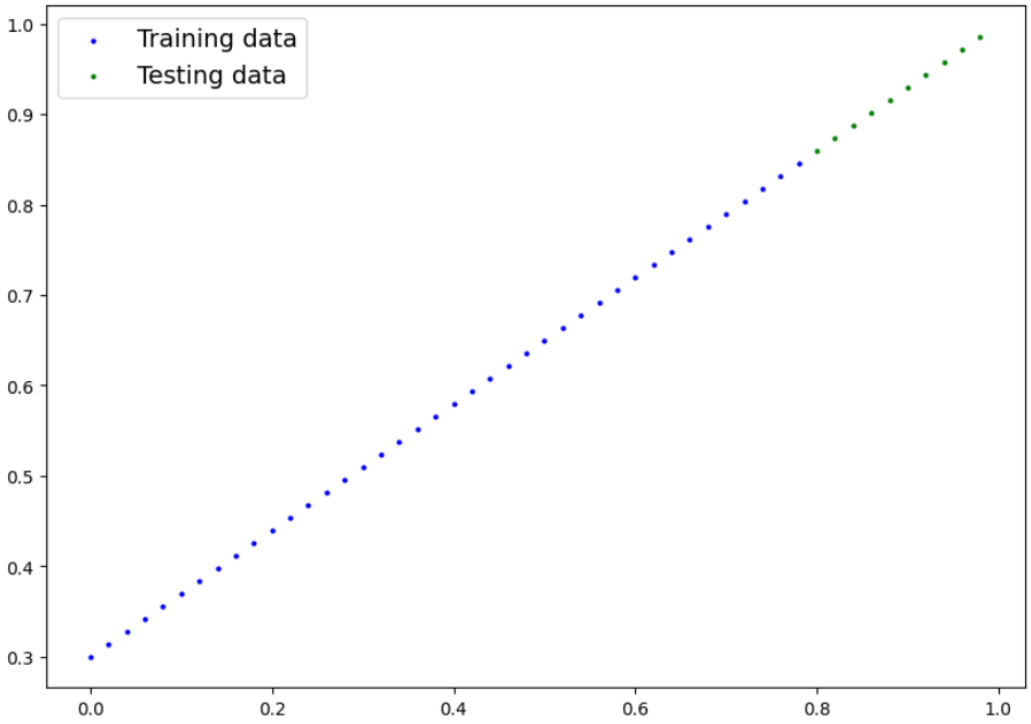
Artinya, terdapat 40 sampel untuk pelatihan dan 10 sampel untuk pengujian.

Visualisasi Data:

Dibuat fungsi **plot_predictions** untuk memvisualisasikan set pelatihan, set pengujian, dan perbandingan prediksi model. Set pelatihan ditampilkan dalam warna biru, set pengujian dalam warna hijau, dan prediksi dalam warna merah (jika ada).

In [5]:
```
def plot_predictions(train_data=X_train,
                     train_labels=y_train,
                     test_data=X_test,
                     test_labels=y_test,
                     predictions=None):
  """
  Plots training data, test data and compares predictions.
  """
  plt.figure(figsize=(10, 7))

  # Plot training data in blue
  plt.scatter(train_data, train_labels, c="b", s=4, label="Training data")

  # Plot test data in green
  plt.scatter(test_data, test_labels, c="g", s=4, label="Testing data")

  if predictions is not None:
    # Plot the predictions in red (predictions were made on the test data)
    plt.scatter(test_data, predictions, c="r", s=4, label="Predictions")

  # Show the legend
  plt.legend(prop={"size": 14});
```

In [6]:
```
plot_predictions();
```



2. Build model

   Making Linear Regression model class:

Membuat kelas model Linear Regression menggunakan PyTorch. Kelas tersebut menjadi subclass dari **nn.Module**, yang merupakan dasar bagi semua modul jaringan saraf PyTorch. Dua parameter utama model adalah **weights** (berat) dan **bias** (bias), keduanya diinisialisasi dengan nilai acak untuk kemudian disesuaikan selama pembelajaran. Menggunakan **nn.Parameter** untuk menyimpan tensor yang dapat digunakan dengan **nn.Module**. Jika **requires_grad=True**, gradient (digunakan untuk pembaruan parameter model melalui gradien descent) dihitung secara otomatis.

In [7]:
```
# Create a Linear Regression model class
class LinearRegressionModel(nn.Module): # <- almost everything in PyTorch is a nn.Module (think of this as neural network le
    def __init__(self):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(1, # <- start with random weights (this will get adjusted as the model learn
                                                dtype=torch.float), # <- PyTorch loves float32 by default
                                    requires_grad=True) # <- can we update this value with gradient descent?)

        self.bias = nn.Parameter(torch.randn(1, # <- start with random bias (this will get adjusted as the model learns)
                                             dtype=torch.float), # <- PyTorch loves float32 by default
                                 requires_grad=True) # <- can we update this value with gradient descent?))

    # Forward defines the computation in the model
    def forward(self, x: torch.Tensor) -> torch.Tensor: # <- "x" is the input data (e.g. training/testing features)
        return self.weights * x + self.bias # <- this is the linear regression formula (y = m*x + b)
```

   Forward Method:

Menggunakan metode **forward** untuk mendefinisikan komputasi dalam model.

Input **x** adalah data masukan (misalnya, fitur pelatihan/pengujian).

Komputasi dalam metode **forward** mengikuti formula Linear Regression: **y = mx + b**.

Dalam pembuatan model dengan PyTorch, konvensi penggunaan kelas Python digunakan, di mana hampir semua hal dalam PyTorch adalah **nn.Module**. **torch.nn.Module** berisi blok-blok dasar untuk grafik komputasi, yang merupakan serangkaian perhitungan yang dieksekusi dengan cara tertentu. **torch.nn.Parameter** digunakan untuk menyimpan parameter-parameter seperti berat dan bias, yang kemudian digunakan dalam **nn.Module** untuk membentuk model.

Checking the contents of a PyTorch model:

Atur seed manual untuk reproduksibilitas menggunakan **torch.manual_seed(42)**. Buat instance model (**LinearRegressionModel**), yang merupakan subclass dari **nn.Module** yang berisi **nn.Parameter(s)**.

Gunakan **model_0.parameters()** untuk memeriksa **nn.Parameter(s)** dalam subclass **nn.Module**. Hasilnya adalah daftar parameter, masing-masing direpresentasikan oleh tensor dengan nilai dan informasi pelacakan gradien.

```
In [8]:
# Set manual seed since nn.Parameter are randomly initialzied
torch.manual_seed(42)

# Create an instance of the model (this is a subclass of nn.Module that contains nn.Parameter(s))
model_0 = LinearRegressionModel()

# Check the nn.Parameter(s) within the nn.Module subclass we created
list(model_0.parameters())
```

```
Out[8]: [Parameter containing:
         tensor([0.3367], requires_grad=True),
         Parameter containing:
         tensor([0.1288], requires_grad=True)]
```

Gunakan **model_0.state_dict()** untuk mendapatkan state model, yang mencakup nilai parameter. Hasilnya adalah **OrderedDict** di mana setiap parameter dinamai, dan nilai tensor yang sesuai ditampilkan.

```
In [9]:
# List named parameters
model_0.state_dict()
```

```
Out[9]: OrderedDict([('weights', tensor([0.3367])), ('bias', tensor([0.1288]))])
```

Nilai untuk **weights** dan **bias** awalnya adalah tensor float acak. Nilai acak ini diperoleh dengan menginisialisasi mereka menggunakan **torch.randn()** selama pembuatan instance model.

Mengubah nilai **torch.manual_seed()** mempengaruhi inisialisasi parameter acak. Ini dapat menjadi langkah penting untuk reproduktibilitas dalam eksperimen machine learning.

Karena model dimulai dengan nilai acak, kekuatan prediktifnya saat ini rendah. Tujuannya adalah memperbarui parameter ini melalui pelatihan untuk mendapatkan hasil prediksi yang lebih baik.

Making predictions using torch.inference_mode()

Untuk menguji seberapa dekat model memprediksi **y_test**, kita menggunakan data uji **X_test**. Melalui **torch.inference_mode()**, model (**model_0**) digunakan untuk membuat prediksi pada data uji (**X_test**). **torch.inference_mode()** digunakan sebagai manajer konteks, mematikan beberapa fitur (seperti pelacakan gradien) untuk meningkatkan kecepatan prediksi.

In [10]:
```
# Make predictions with model
with torch.inference_mode():
    y_preds = model_0(X_test)

# Note: in older PyTorch code you might also see torch.no_grad()
# with torch.no_grad():
#   y_preds = model_0(X_test)
```

torch.inference_mode() digunakan saat melakukan inferensi (membuat prediksi). Mematikan beberapa fitur yang tidak diperlukan selama inferensi, seperti pelacakan gradien, untuk meningkatkan kecepatan.

Dalam kode PyTorch yang lebih lama, mungkin penggunaan **torch.no_grad()** untuk inferensi. **torch.inference_mode()** lebih baru, potensial lebih cepat, dan disarankan.

Hasil prediksi (**y_preds**) terdiri dari nilai untuk setiap sampel data uji (**X_test**). Setiap nilai merupakan prediksi yang dihasilkan oleh model.

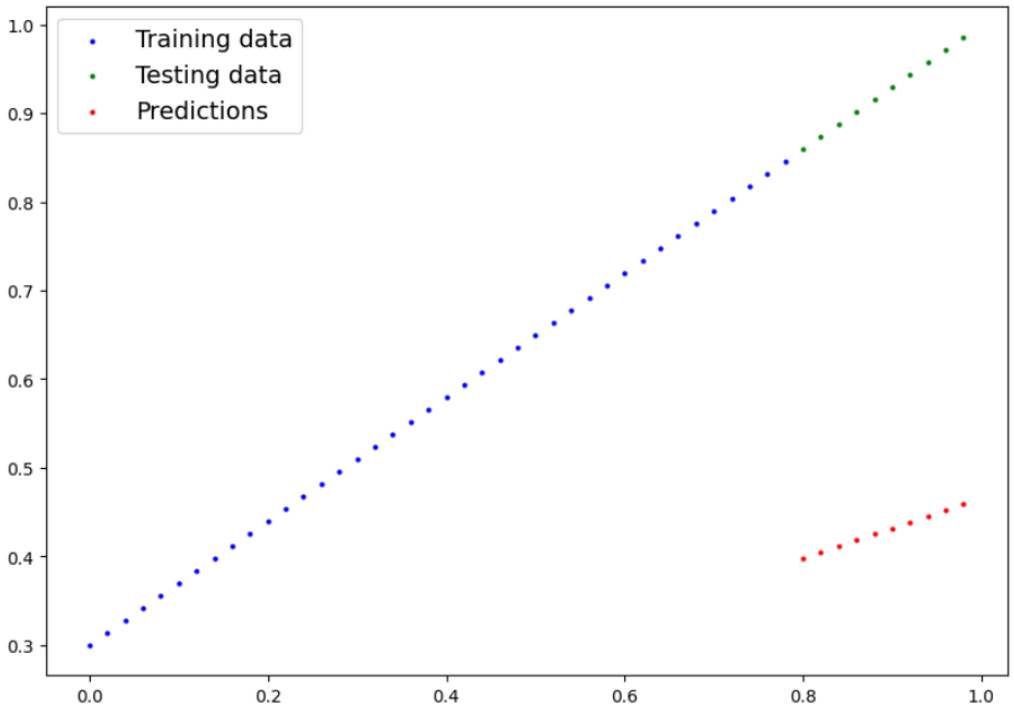
In [11]:
```
# Check the predictions
print(f"Number of testing samples: {len(X_test)}")
print(f"Number of predictions made: {len(y_preds)}")
print(f"Predicted values:\n{y_preds}")
```

```
Number of testing samples: 10
Number of predictions made: 10
Predicted values:
tensor([[0.3982],
        [0.4049],
        [0.4116],
        [0.4184],
        [0.4251],
        [0.4318],
        [0.4386],
        [0.4453],
        [0.4520],
        [0.4588]])
```

Terdapat satu nilai prediksi untuk setiap sampel pengujian, sesuai dengan jenis data yang digunakan (satu nilai X berpasangan dengan satu nilai y pada garis lurus). Model pembelajaran mesin bersifat fleksibel dan dapat menangani skenario di mana beberapa nilai X berkorelasi dengan satu atau lebih nilai y.

Visualisasikan hasil prediksi menggunakan fungsi **plot_predictions()** yang telah dibuat sebelumnya.

```
In [12]: plot_predictions(predictions=y_preds)
```



Perbandingan antara nilai aktual **y_test** dan prediksi **y_preds**. Terlihat bahwa prediksi masih buruk karena model hanya menggunakan nilai parameter acak.

```
In [13]: y_test - y_preds
```

```
Out[13]: tensor([[0.4618],
                 [0.4691],
                 [0.4764],
                 [0.4836],
                 [0.4909],
                 [0.4982],
                 [0.5054],
                 [0.5127],
                 [0.5200],
                 [0.5272]])
```

3. Train model
   PyTorch testing loop

Melibatkan pembuatan model, pelatihan selama 100 epoch (iterasi), dan evaluasi setiap 10 epoch. Menyimpan nilai kerugian pelatihan dan pengujian untuk plot.

In [15]:

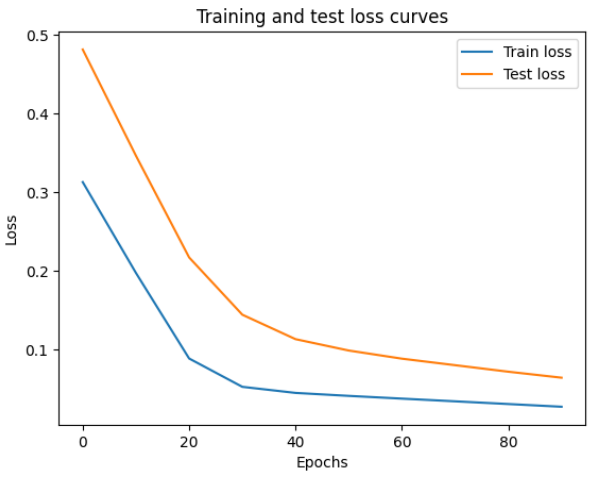
```
torch.manual_seed(42)

# Set the number of epochs (how many times the model will pass over the training data)
epochs = 100

# Create empty loss lists to track values
train_loss_values = []
test_loss_values = []
epoch_count = []

for epoch in range(epochs):
    ### Training

    # Put model in training mode (this is the default state of a model)
    model_0.train()

    # 1. Forward pass on train data using the forward() method inside
    y_pred = model_0(X_train)
    # print(y_pred)

    # 2. Calculate the loss (how different are our models predictions to the ground truth)
    loss = loss_fn(y_pred, y_train)

    # 3. Zero grad of the optimizer
    optimizer.zero_grad()

    # 4. Loss backwards
    loss.backward()

    # 5. Progress the optimizer
    optimizer.step()

    ### Testing

    # Put the model in evaluation mode
    model_0.eval()

    with torch.inference_mode():
      # 1. Forward pass on test data
      test_pred = model_0(X_test)

      # 2. Caculate loss on test data
      test_loss = loss_fn(test_pred, y_test.type(torch.float)) # predictions come in torch.float datatype, so comparisons n

      # Print out what's happening
      if epoch % 10 == 0:
            epoch_count.append(epoch)
            train_loss_values.append(loss.detach().numpy())
            test_loss_values.append(test_loss.detach().numpy())
            print(f"Epoch: {epoch} | MAE Train Loss: {loss} | MAE Test Loss: {test_loss} ")
```

```
Epoch: 0 | MAE Train Loss: 0.31288138031959534 | MAE Test Loss: 0.48106518387794495
Epoch: 10 | MAE Train Loss: 0.1976713240146637 | MAE Test Loss: 0.3463551998138428
Epoch: 20 | MAE Train Loss: 0.08908725529909134 | MAE Test Loss: 0.21729660034179688
Epoch: 30 | MAE Train Loss: 0.053148526698350906 | MAE Test Loss: 0.14464017748832703
Epoch: 40 | MAE Train Loss: 0.04543796554207802 | MAE Test Loss: 0.11360953003168106
Epoch: 50 | MAE Train Loss: 0.04167863354086876 | MAE Test Loss: 0.09919948130846024
Epoch: 60 | MAE Train Loss: 0.03818932920694351 | MAE Test Loss: 0.08886633068323135
Epoch: 70 | MAE Train Loss: 0.03476089984178543 | MAE Test Loss: 0.0805937647819519
Epoch: 80 | MAE Train Loss: 0.03132382780313492 | MAE Test Loss: 0.07232122868299484
Epoch: 90 | MAE Train Loss: 0.02788739837706089 | MAE Test Loss: 0.06473556160926819
```

Kurva kerugian menunjukkan penurunan kerugian seiring waktu. Kerugian mengukur seberapa salah model, jadi semakin rendah, semakin baik.

In [16]:

```
# Plot the loss curves
plt.plot(epoch_count, train_loss_values, label="Train loss")
plt.plot(epoch_count, test_loss_values, label="Test loss")
plt.title("Training and test loss curves")
plt.ylabel("Loss")
plt.xlabel("Epochs")
plt.legend();
```



Model mendekati nilai awal untuk bobot dan bias yang diatur. **Catatan:** Eksekusi percobaan untuk mengamati perubahan hasil dengan mengubah nilai **epochs** menjadi 200.

In [17]:
```
# Find our model's learned parameters
print("The model learned the following values for weights and bias:")
print(model_0.state_dict())
print("\nAnd the original values for weights and bias are:")
print(f"weights: {weight}, bias: {bias}")
```

```
The model learned the following values for weights and bias:
OrderedDict([('weights', tensor([0.5784])), ('bias', tensor([0.3513]))])

And the original values for weights and bias are:
weights: 0.7, bias: 0.3
```

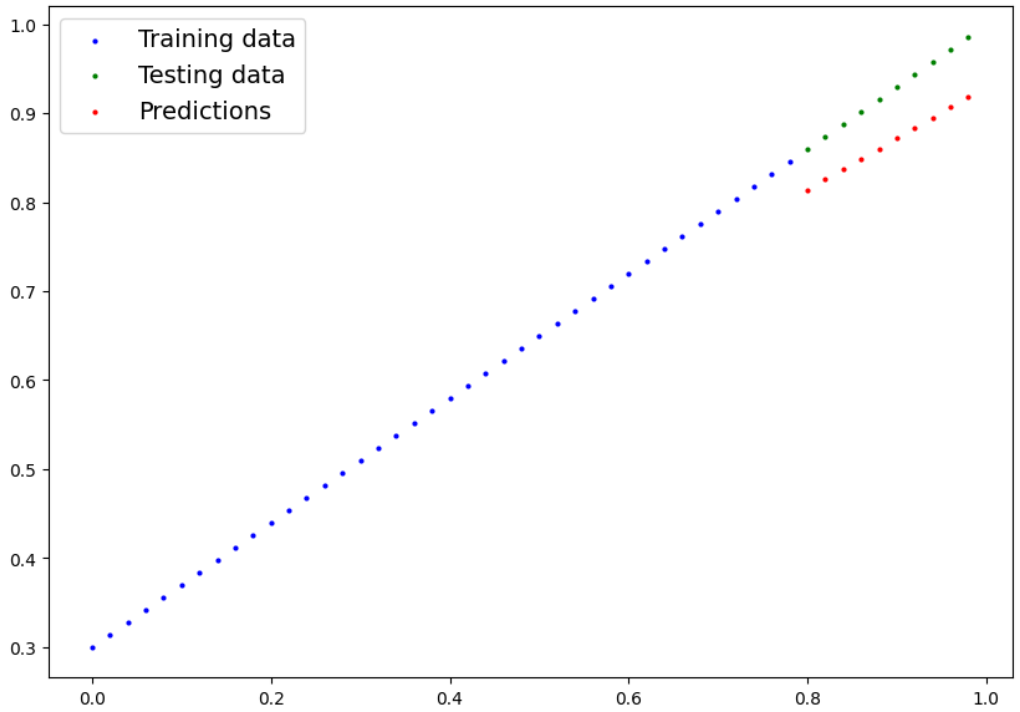
4. Making predictions with a trained PyTorch model (inference) Langkah-langkah Melakukan Prediksi:

1. Setel model dalam mode evaluasi (model.eval()).
2. Lakukan prediksi menggunakan manajer konteks inference mode (with torch.inference_mode(): ...).
3. Pastikan semua prediksi dilakukan dengan objek pada perangkat yang sama (mis., data dan model hanya pada GPU atau hanya pada CPU).

In [18]:
```
# 1. Set the model in evaluation mode
model_0.eval()

# 2. Setup the inference mode context manager
with torch.inference_mode():
  # 3. Make sure the calculations are done with the model and data on the same device
  # in our case, we haven't setup device-agnostic code yet so our data and model are
  # on the CPU by default.
  # model_0.to(device)
  # X_test = X_test.to(device)
  y_preds = model_0(X_test)
y_preds
```

Out[18]:
```
tensor([[0.8141],
        [0.8256],
        [0.8372],
        [0.8488],
        [0.8603],
        [0.8719],
        [0.8835],
        [0.8950],
        [0.9066],
        [0.9182]])
```

```
In [19]: plot_predictions(predictions=y_preds)
```



Prediksi tampak jauh lebih dekat daripada sebelumnya, menunjukkan model yang telah dilatih dapat memberikan hasil yang lebih baik.

5. Saving and loading a PyTorch model

torch.save: Menyimpan objek yang telah diserialisasi ke disk menggunakan utilitas pickle Python. Model, tensor, dan berbagai objek Python lainnya seperti kamus dapat disimpan menggunakan torch.save.

torch.load: Menggunakan fitur unpickling dari pickle untuk deserialisasi dan memuat file objek Python yang telah di-pickle (seperti model, tensor, atau kamus) ke dalam memori. Dapat juga menetapkan perangkat mana untuk memuat objek tersebut (CPU, GPU, dll).

torch.nn.Module.load_state_dict: Memuat kamus parameter model (model.state_dict()) menggunakan objek state_dict() yang telah disimpan.

Proses Menyimpan Model PyTorch:

1. Buat direktori untuk menyimpan model, misalnya, "models".
2. Tentukan jalur file untuk menyimpan model.
3. Gunakan torch.save(obj, f) di mana obj adalah state_dict() model target, dan f adalah nama file untuk menyimpan model.

In [20]:
```
from pathlib import Path

# 1. Create models directory
MODEL_PATH = Path("models")
MODEL_PATH.mkdir(parents=True, exist_ok=True)

# 2. Create model save path
MODEL_NAME = "01_pytorch_workflow_model_0.pth"
MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME

# 3. Save the model state dict
print(f"Saving model to: {MODEL_SAVE_PATH}")
torch.save(obj=model_0.state_dict(), # only saving the state_dict() only saves the models learned parameters
           f=MODEL_SAVE_PATH)
```

```
Saving model to: models/01_pytorch_workflow_model_0.pth
```

In [21]:
```
# Check the saved file path
!ls -l models/01_pytorch_workflow_model_0.pth
```

```
-rw-r--r-- 1 root root 1680 Jan  4 14:55 models/01_pytorch_workflow_model_0.pth
```

Proses Memuat Model PyTorch:

1. Membuat model baru dengan instance LinearRegressionModel().
2. Memuat state_dict() model yang telah disimpan menggunakan torch.load() dan menyusunnya dengan model baru menggunakan load_state_dict().

In [22]:
```
# Instantiate a new instance of our model (this will be instantiated with random weights)
loaded_model_0 = LinearRegressionModel()

# Load the state_dict of our saved model (this will update the new instance of our model with trained weights)
loaded_model_0.load_state_dict(torch.load(f=MODEL_SAVE_PATH))
```

Out[22]:
```
<All keys matched successfully>
```

Melakukan Prediksi dengan Model yang Telah Dimuat:

1. Setel model yang dimuat dalam mode evaluasi.
2. Gunakan konteks manajer inference untuk melakukan prediksi.

In [23]:
```
# 1. Put the loaded model into evaluation mode
loaded_model_0.eval()

# 2. Use the inference mode context manager to make predictions
with torch.inference_mode():
    loaded_model_preds = loaded_model_0(X_test) # perform a forward pass on the test data with the loaded model
```

Melakukan prediksi dari model yang dimuat

In [24]:
```
# Compare previous model predictions with loaded model predictions (these should be the same)
y_preds == loaded_model_preds
```

Out[24]:
```
tensor([[True],
        [True],
        [True],
        [True],
        [True],
        [True],
        [True],
        [True],
        [True],
        [True]])
```

Prediksi terlihat sama dengan model sebelumnya yang dibuat

Chapter 02 PyTorch Neural Network Classification

1. Make classification data and get it ready

Menggunakan metode make_circles() dari Scikit-Learn untuk menghasilkan dua lingkaran dengan titik-titik berwarna berbeda. Membuat DataFrame dengan kolom X1, X2, dan label (y).

In [ ]:
```
from sklearn.datasets import make_circles


# Make 1000 samples
n_samples = 1000

# Create circles
X, y = make_circles(n_samples,
                    noise=0.03, # a little bit of noise to the dots
                    random_state=42) # keep random state so we get the same values
```

In [ ]:
```
# Make DataFrame of circle data
import pandas as pd
circles = pd.DataFrame({"X1": X[:, 0],
    "X2": X[:, 1],
    "label": y
})
circles.head(10)
```

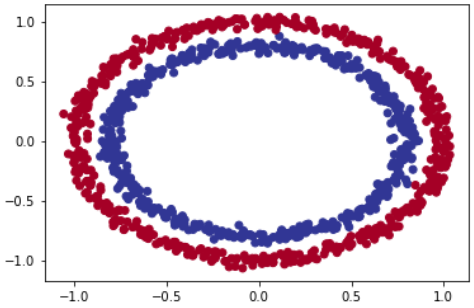
Out[ ]:

	X1	X2	label
0	0.754246	0.231481	1
1	-0.756159	0.153259	1
2	-0.815392	0.173282	1
3	-0.393731	0.692883	1
4	0.442208	-0.896723	0
5	-0.479646	0.676435	1
6	-0.013648	0.803349	1
7	0.771513	0.147760	1
8	-0.169322	-0.793456	1
9	-0.121486	1.021509	0

Visualize Data:

Melihat sebaran data menggunakan scatter plot.

In [ ]:
```
# Visualize with a plot
import matplotlib.pyplot as plt
plt.scatter(x=X[:, 0],
            y=X[:, 1],
            c=y,
            cmap=plt.cm.RdYlBu);
```



Check Shape Data:

Melihat bentuk (shape) dari fitur (X) dan label (y).

Melihat bentuk dari satu sampel fitur dan label.

```
In [ ]:
# Check the shapes of our features and labels
X.shape, y.shape
```

```
Out[ ]: ((1000, 2), (1000,))
```

Terlihat X dan Y memiliki nilai sama, namun angka 2 pada X belum diketahui

```
In [ ]:
# View the first example of features and labels
X_sample = X[0]
y_sample = y[0]
print(f"Values for one sample of X: {X_sample} and the same for y: {y_sample}")
print(f"Shapes for one sample of X: {X_sample.shape} and the same for y: {y_sample.shape}")
```

```
Values for one sample of X: [0.75424625 0.23148074] and the same for y: 1
Shapes for one sample of X: (2,) and the same for y: ()
```

Dijelaskan ternyata angka 2 pada X menandakan 2 features (vector), sementara y single features (scalar).

Turn data into tensors and create train and test splits

Mengonversi data menjadi tensor menggunakan torch.from_numpy(). Melakukan pembagian data menjadi set latih dan uji menggunakan train_test_split().

```
In [ ]:
# Turn data into tensors
# Otherwise this causes issues with computations later on
import torch
X = torch.from_numpy(X).type(torch.float)
y = torch.from_numpy(y).type(torch.float)

# View the first five samples
X[:5], y[:5]
```

```
Out[ ]: (tensor([[ 0.7542,  0.2315],
                 [-0.7562,  0.1533],
                 [-0.8154,  0.1733],
                 [-0.3937,  0.6929],
                 [ 0.4422, -0.8967]]),
         tensor([1., 1., 1., 1., 0.]))
```

Data telah diubah ke format tensor. Selanjutnya menggunakan test_size=0.2 (80% training dan 20% testing) dan random state=42

```
In [ ]:
# Split data into train and test sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2, # 20% test, 80% train
                                                    random_state=42) # make the random split reproducible

len(X_train), len(X_test), len(y_train), len(y_test)
```

```
Out[ ]: (800, 200, 800, 200)
```

Terdapat 800 training sample dan 200 testing sample.

2. Building a model

Device Agnostic Code

Menggunakan PyTorch untuk membuat model. Menentukan apakah akan menggunakan CPU atau GPU (jika tersedia).

In [ ]:
```
# Standard PyTorch imports
import torch
from torch import nn

# Make device agnostic code
device = "cuda" if torch.cuda.is_available() else "cpu"
device
```

Out[ ]:
```
'cuda'
```

Construct a model class

Membuat kelas model yang mewarisi dari **nn.Module**. Menentukan dua layer **nn.Linear** di konstruktor untuk menangani bentuk input dan output dari X dan y. Mendefinisikan metode **forward()** yang berisi komputasi forward pass dari model. Menginstansiasi kelas model dan mengirimkannya ke perangkat yang ditentukan.

In [ ]:
```
# 1. Construct a model class that subclasses nn.Module
class CircleModelV0(nn.Module):
    def __init__(self):
        super().__init__()
        # 2. Create 2 nn.Linear layers capable of handling X and y input and output shapes
        self.layer_1 = nn.Linear(in_features=2, out_features=5) # takes in 2 features (X), produces 5 features
        self.layer_2 = nn.Linear(in_features=5, out_features=1) # takes in 5 features, produces 1 feature (y)

    # 3. Define a forward method containing the forward pass computation
    def forward(self, x):
        # Return the output of layer_2, a single feature, the same shape as y
        return self.layer_2(self.layer_1(x)) # computation goes through layer_1 first then the output of layer_1 goes thr

# 4. Create an instance of the model and send it to target device
model_0 = CircleModelV0().to(device)
model_0
```

Out[ ]:
```
CircleModelV0(
  (layer_1): Linear(in_features=2, out_features=5, bias=True)
  (layer_2): Linear(in_features=5, out_features=1, bias=True)
)
```

Make predictions with the model

Menggunakan model untuk membuat prediksi pada data uji.

In [ ]:
```
# Make predictions with the model
untrained_preds = model_0(X_test.to(device))
print(f"Length of predictions: {len(untrained_preds)}, Shape: {untrained_preds.shape}")
print(f"Length of test samples: {len(y_test)}, Shape: {y_test.shape}")
print(f"\nFirst 10 predictions:\n{untrained_preds[:10]}")
print(f"\nFirst 10 test labels:\n{y_test[:10]}")
```

```
Length of predictions: 200, Shape: torch.Size([200, 1])
Length of test samples: 200, Shape: torch.Size([200])

First 10 predictions:
tensor([[-0.4279],
        [-0.3417],
        [-0.5975],
        [-0.3801],
        [-0.5078],
        [-0.4559],
        [-0.2842],
        [-0.3107],
        [-0.6010],
        [-0.3350]], device='cuda:0', grad_fn=<SliceBackward0>)

First 10 test labels:
tensor([1., 0., 1., 0., 1., 1., 0., 0., 1., 0.])
```

Setup loss function and optimizer

Memilih fungsi kerugian yang sesuai dengan tipe masalah klasifikasi biner (binary cross entropy loss).

Memilih optimizer (SGD) untuk mengoptimalkan parameter model.

In [ ]:
```
# Create a loss function
# loss_fn = nn.BCELoss() # BCELoss = no sigmoid built-in
loss_fn = nn.BCEWithLogitsLoss() # BCEWithLogitsLoss = sigmoid built-in

# Create an optimizer
optimizer = torch.optim.SGD(params=model_0.parameters(),
                            lr=0.1)
```

Evaluation metric

Menyusun fungsi untuk menghitung akurasi sebagai metrik evaluasi.

In [ ]:
```
# Calculate accuracy (a classification metric)
def accuracy_fn(y_true, y_pred):
    correct = torch.eq(y_true, y_pred).sum().item() # torch.eq() calculates where two tensors are equal
    acc = (correct / len(y_pred)) * 100
    return acc
```

3. Train model

Going from raw model outputs to predicted labels (logits -> prediction probabilities -> prediction labels)

Sebelum memulai langkah pelatihan, periksa keluaran mentah model selama proses forward pass.

Menerapkan sigmoid activation function untuk mengubah logits menjadi probabilitas prediksi.

Menyaring probabilitas prediksi untuk mendapatkan label prediksi.

Membandingkan label prediksi dengan label sebenarnya untuk memastikan konsistensi.

In [ ]:
```
# View the frist 5 outputs of the forward pass on the test data
y_logits = model_0(X_test.to(device))[:5]
y_logits
```

Out[ ]:
```
tensor([[-0.4279],
        [-0.3417],
        [-0.5975],
        [-0.3801],
        [-0.5078]], device='cuda:0', grad_fn=<SliceBackward0>)
```

In [ ]:
```
# Use sigmoid on model logits
y_pred_probs = torch.sigmoid(y_logits)
y_pred_probs
```

Out[ ]:
```
tensor([[0.3946],
        [0.4154],
        [0.3549],
        [0.4061],
        [0.3757]], device='cuda:0', grad_fn=<SigmoidBackward0>)
```

In [ ]:
```
# Find the predicted labels (round the prediction probabilities)
y_preds = torch.round(y_pred_probs)

# In full
y_pred_labels = torch.round(torch.sigmoid(model_0(X_test.to(device))[:5]))

# Check for equality
print(torch.eq(y_preds.squeeze(), y_pred_labels.squeeze()))

# Get rid of extra dimension
y_preds.squeeze()
```

```
tensor([True, True, True, True, True], device='cuda:0')
```

Out[ ]:
```
tensor([0., 0., 0., 0., 0.], device='cuda:0', grad_fn=<SqueezeBackward0>)
```

In [ ]:
```
y_test[:5]
```

Out[ ]:
```
tensor([1., 0., 1., 0., 1.])
```

Building a training and testing loop

Model dilatih dan diuji selama 100 epoch.

Data dipindahkan ke perangkat target (device).

Loop pelatihan dan evaluasi dibangun dengan langkah-langkah berikut:

1. Forward pass (keluaran model adalah logits mentah).
2. Menghitung loss/akurasi pada data pelatihan.
3. Mengoptimalkan model dengan menghitung gradien loss dan melakukan langkah optimasi.
4. Forward pass pada data uji dan menghitung loss/akurasi pada data uji.
5. Menampilkan hasil setiap 10 epoch.

In [ ]:

```
torch.manual_seed(42)

# Set the number of epochs
epochs = 100

# Put data to target device
X_train, y_train = X_train.to(device), y_train.to(device)
X_test, y_test = X_test.to(device), y_test.to(device)

# Build training and evaluation loop
for epoch in range(epochs):
    ### Training
    model_0.train()

    # 1. Forward pass (model outputs raw logits)
    y_logits = model_0(X_train).squeeze() # squeeze to remove extra `1` dimensions, this won't work unless model and data a
    y_pred = torch.round(torch.sigmoid(y_logits)) # turn logits -> pred probs -> pred labls

    # 2. Calculate loss/accuracy
    # loss = loss_fn(torch.sigmoid(y_logits), # Using nn.BCELoss you need torch.sigmoid()
    #                y_train)
    loss = loss_fn(y_logits, # Using nn.BCEWithLogitsLoss works with raw logits
                   y_train)
    acc = accuracy_fn(y_true=y_train,
                      y_pred=y_pred)

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backwards
    loss.backward()

    # 5. Optimizer step
    optimizer.step()

    ### Testing
    model_0.eval()
    with torch.inference_mode():
        # 1. Forward pass
        test_logits = model_0(X_test).squeeze()
        test_pred = torch.round(torch.sigmoid(test_logits))
        # 2. Caculate loss/accuracy
        test_loss = loss_fn(test_logits,
                            y_test)
        test_acc = accuracy_fn(y_true=y_test,
                               y_pred=test_pred)

    # Print out what's happening every 10 epochs
    if epoch % 10 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc:.2f}% | Test loss: {test_loss:.5f}, Test acc: {test_acc:.
```

```
Epoch: 0 | Loss: 0.72090, Accuracy: 50.00% | Test loss: 0.72196, Test acc: 50.00%
Epoch: 10 | Loss: 0.70291, Accuracy: 50.00% | Test loss: 0.70542, Test acc: 50.00%
Epoch: 20 | Loss: 0.69659, Accuracy: 50.00% | Test loss: 0.69942, Test acc: 50.00%
Epoch: 30 | Loss: 0.69432, Accuracy: 43.25% | Test loss: 0.69714, Test acc: 41.00%
Epoch: 40 | Loss: 0.69349, Accuracy: 47.00% | Test loss: 0.69623, Test acc: 46.50%
Epoch: 50 | Loss: 0.69319, Accuracy: 49.00% | Test loss: 0.69583, Test acc: 46.00%
Epoch: 60 | Loss: 0.69308, Accuracy: 50.12% | Test loss: 0.69563, Test acc: 46.50%
Epoch: 70 | Loss: 0.69303, Accuracy: 50.38% | Test loss: 0.69551, Test acc: 46.00%
Epoch: 80 | Loss: 0.69302, Accuracy: 51.00% | Test loss: 0.69543, Test acc: 46.00%
Epoch: 90 | Loss: 0.69301, Accuracy: 51.00% | Test loss: 0.69537, Test acc: 46.00%
```

Chapter 03 PyTorch Computer Vision

Computer Vision adalah seni mengajari komputer untuk melihat dan memahami informasi visual. Contoh masalah Computer Vision melibatkan klasifikasi gambar (klasifikasi biner atau multikelas), deteksi objek, dan segmentasi panoptik.

Computer Vision digunakan dalam berbagai aplikasi sehari-hari, seperti di kamera ponsel, mobil modern, manufaktur, dan kamera keamanan. Hampir semua yang dapat dijelaskan secara visual dapat menjadi masalah Computer Vision.

Computer vision libraries in PyTorch

- **torchvision**: Berisi dataset, arsitektur model, dan transformasi gambar yang sering digunakan untuk masalah vision komputer.
- **torchvision.datasets**: Menyediakan berbagai dataset vision komputer untuk berbagai masalah, termasuk klasifikasi gambar, deteksi objek, image captioning, video classification, dan lainnya. Juga berisi serangkaian kelas dasar untuk membuat dataset kustom.
- **torchvision.models**: Berisi arsitektur model vision komputer yang sudah teruji dan umum digunakan, diimplementasikan dalam PyTorch, dapat digunakan untuk masalah kustom.
- **torchvision.transforms**: Transformasi gambar sering diperlukan sebelum digunakan dalam model, termasuk transformasi gambar umum.
- **torch.utils.data.Dataset**: Kelas dasar dataset untuk PyTorch, tidak hanya untuk vision komputer, tetapi juga dapat menangani berbagai jenis data.
- **torch.utils.data.DataLoader**: Membuat iterable Python dari dataset (dibuat dengan **torch.utils.data.Dataset**), mempermudah pelatihan model.

Import PyTorch and Check Version

Mengimpor PyTorch dan modul lainnya. Memeriksa versi PyTorch dan torchvision, memastikan versi PyTorch tidak lebih rendah dari 1.10.0 dan versi torchvision tidak lebih rendah dari 0.11.

In [ ]:

```python
# Import PyTorch
import torch
from torch import nn

# Import torchvision
import torchvision
from torchvision import datasets
from torchvision.transforms import ToTensor

# Import matplotlib for visualization
import matplotlib.pyplot as plt

# Check versions
# Note: your PyTorch version shouldn't be lower than 1.10.0 and torchvision version shouldn't be lower than 0.11
print(f"PyTorch version: {torch.__version__}\ntorchvision version: {torchvision.__version__}")
```

```
PyTorch version: 2.0.1+cu118
torchvision version: 0.15.2+cu118
```

1. Getting a dataset

FashionMNIST Dataset:

FashionMNIST adalah dataset vision komputer yang mirip dengan MNIST, tetapi berisi gambar grayscale dari 10 jenis pakaian yang berbeda. Dapat diakses melalui **torchvision.datasets.FashionMNIST()**.

Dataset Selection Parameters:

root: Lokasi folder untuk mengunduh data.

train: Menentukan apakah dataset yang diambil adalah data latih atau uji.

download: Menentukan apakah data akan diunduh jika belum ada di disk.

transform: Transformasi yang akan diterapkan pada gambar.

target_transform: Transformasi yang dapat diterapkan pada label.

In [ ]:
```
# 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()
)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to data/FashionMNIST/raw/tra
in-images-idx3-ubyte.gz
100%|██████████| 26421880/26421880 [00:01<00:00, 16189161.14it/s]
Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to data/FashionMNIST/raw/tra
in-labels-idx1-ubyte.gz
100%|██████████| 29515/29515 [00:00<00:00, 269809.67it/s]
Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to data/FashionMNIST/raw/t10k
-images-idx3-ubyte.gz
100%|██████████| 4422102/4422102 [00:00<00:00, 4950701.58it/s]
Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to data/FashionMNIST/raw/t10k
-labels-idx1-ubyte.gz
100%|██████████| 5148/5148 [00:00<00:00, 4744512.63it/s]
Extracting data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to data/FashionMNIST/raw
```

Data Preview:

Dalam contoh, data latih dan uji diunduh menggunakan FashionMNIST. Data latih diakses melalui **train_data** dan data uji melalui **test_data**. Contoh pertama dari data latih ditampilkan menggunakan **train_data[0]**, memberikan gambar dan labelnya.

In [ ]:

```python
# See first training sample
image, label = train_data[0]
image, label
```

Out[ ]:

```
(tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0039, 0.0000, 0.0000, 0.0510,
           0.2863, 0.0000, 0.0000, 0.0039, 0.0157, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0039, 0.0039, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0118, 0.0000, 0.1412, 0.5333,
           0.4980, 0.2431, 0.2118, 0.0000, 0.0000, 0.0000, 0.0039, 0.0118,
           0.0157, 0.0000, 0.0000, 0.0118],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0235, 0.0000, 0.4000, 0.8000,
           0.6902, 0.5255, 0.5647, 0.4824, 0.0902, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0471, 0.0392, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.6078, 0.9255,
           0.8118, 0.6980, 0.4196, 0.6118, 0.6314, 0.4275, 0.2510, 0.0902,
           0.3020, 0.5098, 0.2824, 0.0588],
```

Input and output shapes of a computer vision model

Image Shape:

Bentuk tensor gambar adalah **[1, 28, 28]**, yang dapat diuraikan sebagai **[color_channels=1, height=28, width=28]**.

Warna gambar dalam hal ini adalah grayscale, karena **color_channels=1**.

Tensor Format:

Urutan dimensi tensor saat ini sering disebut sebagai CHW (Color Channels, Height, Width).

PyTorch umumnya mengambil NCHW (channels first) sebagai format default, dengan N sebagai jumlah gambar dalam satu batch

Ada format lain seperti NHWC (channels last), di mana C (Channels) berada di posisi terakhir.

NHWC biasanya dianggap sebagai praktik terbaik dan memberikan performa lebih baik, terutama pada dataset dan model yang lebih besar.

Sample Value:

Jumlah sampel dalam data latih adalah 60,000, dan dalam data uji adalah 10,000.

Classes:

Kelas-kelas dapat diakses melalui atribut **.classes** dari dataset.

Kelas-kelas FashionMNIST terdiri dari 10 jenis pakaian.

In [ ]:
```
# What's the shape of the image?
image.shape
```

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

In [ ]:
```
# How many samples are there?
len(train_data.data), len(train_data.targets), len(test_data.data), len(test_data.targets)
```

Out[ ]:
```
(60000, 60000, 10000, 10000)
```

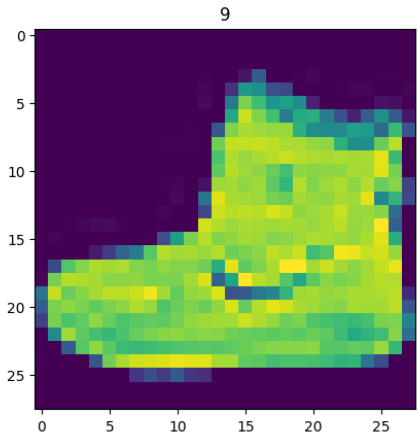
In [ ]:
```
# See classes
class_names = train_data.classes
class_names
```

Out[ ]:
```
['T-shirt/top',
 'Trouser',
 'Pullover',
 'Dress',
 'Coat',
 'Sandal',
 'Shirt',
 'Sneaker',
 'Bag',
 'Ankle boot']
```

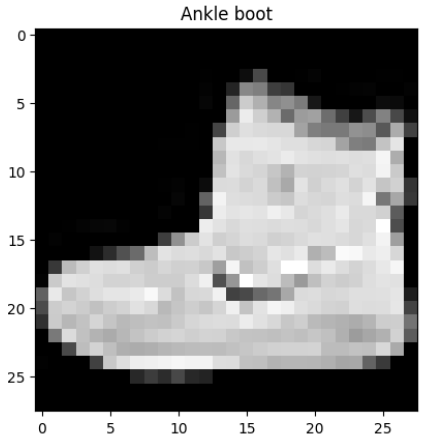
Visualizing our data

In [ ]:
```
import matplotlib.pyplot as plt
image, label = train_data[0]
print(f"Image shape: {image.shape}")
plt.imshow(image.squeeze()) # image shape is [1, 28, 28] (colour channels, height, width)
plt.title(label);
```

```
Image shape: torch.Size([1, 28, 28])
```



In [ ]:
```
plt.imshow(image.squeeze(), cmap="gray")
plt.title(class_names[label]);
```



In [ ]:

```
# Plot more images
torch.manual_seed(42)
fig = plt.figure(figsize=(9, 9))
rows, cols = 4, 4
for i in range(1, rows * cols + 1):
    random_idx = torch.randint(0, len(train_data), size=[1]).item()
    img, label = train_data[random_idx]
    fig.add_subplot(rows, cols, i)
    plt.imshow(img.squeeze(), cmap="gray")
    plt.title(class_names[label])
    plt.axis(False);
```

Gambar pertama dari dataset FashionMNIST ditampilkan dalam format grayscale.

Dengan mengatur parameter cmap="gray" pada plt.imshow(), kita dapat mengonversi gambar ke skala abu-abu.

Beberapa gambar acak dari dataset juga divisualisasikan.

Meskipun dataset ini mungkin tidak terlihat estetis, prinsip membangun model untuk menemukan pola dalam nilai piksel tetap relevan untuk berbagai masalah visi komputer.

Pentingnya pemahaman pola dalam data piksel menjadi dasar untuk membangun model yang dapat digunakan pada data piksel masa depan.

2. Prepare DataLoader

Dalam bagian ini menggunakan **torch.utils.data.DataLoader** untuk mempersiapkan dataset FashionMNIST agar dapat dimuat ke dalam model. DataLoader membantu membagi dataset besar menjadi potongan kecil yang disebut batch atau mini-batch.

In [ ]:
```
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}")
```

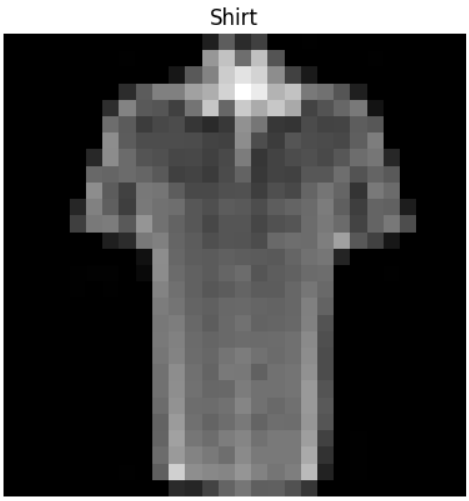
```
Dataloaders: (<torch.utils.data.dataloader.DataLoader object at 0x7fc991463cd0>, <torch.utils.data.dataloader.DataLoader objec
t at 0x7fc991475120>)
Length of train dataloader: 1875 batches of 32
Length of test dataloader: 313 batches of 32
```

In [ ]:
```
# 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
```

Out[ ]:
```
(torch.Size([32, 1, 28, 28]), torch.Size([32]))
```

In [ ]:
```
# Show a sample
torch.manual_seed(42)
random_idx = torch.randint(0, len(train_features_batch), size=[1]).item()
img, label = train_features_batch[random_idx], train_labels_batch[random_idx]
plt.imshow(img.squeeze(), cmap="gray")
plt.title(class_names[label])
plt.axis("Off");
print(f"Image size: {img.shape}")
print(f"Label: {label}, label size: {label.shape}")
```

```
Image size: torch.Size([1, 28, 28])
Label: 6, label size: torch.Size([])
```



DataLoader digunakan untuk mempermudah proses pemuatan data ke dalam model untuk pelatihan dan inferensi.

Batch size adalah parameter penting yang dapat diatur, dan seringkali nilainya berupa kelipatan dari 2 (misalnya 32, 64, 128).

DataLoader membagi dataset menjadi batch kecil, meningkatkan efisiensi komputasi dan memberikan model lebih banyak kesempatan untuk memperbaiki diri pada setiap epoch.

Pemuatan data dilakukan dengan menggunakan iterables, sehingga memungkinkan kita mengakses batch data selanjutnya secara mudah.

3. Model 0: Build a baseline model

Dalam bagian ini, kita membangun model dasar (baseline model) dengan menurunkan kelas dari **nn.Module**. Model dasar ini terdiri dari dua layer **nn.Linear()** dan menggunakan layer **nn.Flatten()** sebagai layer pertama.

nn.Flatten():

In [ ]:
```
# Create a flatten layer
flatten_model = nn.Flatten() # all nn modules function as a model (can do a forward pass)

# Get a single sample
x = train_features_batch[0]

# Flatten the sample
output = flatten_model(x) # perform forward pass

# Print out what happened
print(f"Shape before flattening: {x.shape} -> [color_channels, height, width]")
print(f"Shape after flattening: {output.shape} -> [color_channels, height*width]")

# Try uncommenting below and see what happens
#print(x)
#print(output)
```

```
Shape before flattening: torch.Size([1, 28, 28]) -> [color_channels, height, width]
Shape after flattening: torch.Size([1, 784]) -> [color_channels, height*width]
```

FashionMNISTModelV0 Class:

In [ ]:
```
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 samp
            nn.Linear(in_features=hidden_units, out_features=output_shape)
        )

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

Creating a Model Instantiation:

In [ ]:
```
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
```

Out[ ]:
```
FashionMNISTModelV0(
  (layer_stack): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=784, out_features=10, bias=True)
    (2): Linear(in_features=10, out_features=10, bias=True)
  )
)
```

Model dasar (baseline) ini menggunakan dua layer **nn.Linear()** dengan layer **nn.Flatten()** sebagai layer pertama.

Layer **nn.Flatten()** digunakan untuk meratakan dimensi tensor menjadi vektor satu dimensi.

Parameter model seperti **input_shape**, **hidden_units**, dan **output_shape** ditentukan untuk membangun model sesuai dengan karakteristik dataset FashionMNIST.

Model dibangun sebagai kelas yang menurunkan **nn.Module** dan memiliki metode **forward** untuk melakukan operasi forward pass.

Setup loss, optimizer and evaluation metrics

Dalam bagian ini menyiapkan fungsi loss, optimizer, dan metrik evaluasi untuk model. Kita juga mengimpor fungsi bantuan, terutama fungsi akurasi (**accuracy_fn**), yang telah didefinisikan sebelumnya.

Selanjutnya, kita mengimpor fungsi akurasi dari file **helper_functions** dan menyiapkan fungsi loss dan optimizer.

In [ ]:
```
import requests
from pathlib import Path

# Download helper functions from Learn PyTorch repo (if not already downloaded)
if Path("helper_functions.py").is_file():
  print("helper_functions.py already exists, skipping download")
else:
  print("Downloading helper_functions.py")
  # Note: you need the "raw" GitHub URL for this to work
  request = requests.get("https://raw.githubusercontent.com/mrdbourke/pytorch-deep-learning/main/helper_functions.py")
  with open("helper_functions.py", "wb") as f:
    f.write(request.content)
```

```
Downloading helper_functions.py
```

In [ ]:
```
# Import accuracy metric
from helper_functions import accuracy_fn # Note: could also use torchmetrics.Accuracy(task = 'multiclass', num_classes=len(

# 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)
```

Menggunakan fungsi akurasi (**accuracy_fn**) dari file **helper_functions** untuk mengukur performa model.

Fungsi loss yang digunakan adalah **nn.CrossEntropyLoss()**, yang cocok untuk masalah klasifikasi multi-kelas.

Optimizer yang digunakan adalah Stochastic Gradient Descent (SGD) dengan laju pembelajaran (learning rate) sebesar 0.1.

Kedua fungsi ini akan digunakan selama proses pelatihan untuk mengukur performa model dan melakukan penyesuaian parameter.

Creating a function to time our experiments

Dalam bagian ini membuat fungsi **print_train_time** untuk mencetak selisih waktu antara waktu mulai dan waktu selesai dari sebuah eksperimen pelatihan model. Fungsi ini akan menggunakan fungsi **default_timer** dari modul **timeit** di Python.

In [ ]:

```python
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
```

Fungsi **print_train_time** digunakan untuk mencetak selisih waktu antara waktu mulai dan waktu selesai dari eksperimen pelatihan model.

Fungsi ini dapat membantu kita membandingkan waktu pelatihan antara model yang dijalankan di CPU dan GPU dalam eksperimen berikutnya.

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

Dalam bagian ini, kita membuat loop pelatihan dan pengujian untuk melatih dan mengevaluasi model kita. Kita menggunakan DataLoader untuk memuat data dalam batch.

In [ ]:

```python
# 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))
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
Epoch: 0
-------
Looked at 0/60000 samples
Looked at 12800/60000 samples
Looked at 25600/60000 samples
Looked at 38400/60000 samples
Looked at 51200/60000 samples

Train loss: 0.59039 | Test loss: 0.50954, Test acc: 82.04%

Epoch: 1
-------
Looked at 0/60000 samples
Looked at 12800/60000 samples
Looked at 25600/60000 samples
Looked at 38400/60000 samples
Looked at 51200/60000 samples

Train loss: 0.47633 | Test loss: 0.47989, Test acc: 83.20%

Epoch: 2
-------
Looked at 0/60000 samples
Looked at 12800/60000 samples
Looked at 25600/60000 samples
Looked at 38400/60000 samples
Looked at 51200/60000 samples

Train loss: 0.45503 | Test loss: 0.47664, Test acc: 83.43%

Train time on cpu: 32.349 seconds
```

Dengan menggunakan DataLoader, dapat membuat loop pelatihan dan pengujian untuk melatih dan mengevaluasi model pada data dalam bentuk batch.

Loop pelatihan melibatkan langkah-langkah biasa seperti forward pass, perhitungan loss, backward pass, dan optimasi.

Loop pengujian melibatkan perhitungan loss dan akurasi pada setiap batch data pengujian.

Hasil pelatihan ditampilkan untuk setiap epoch, mencakup loss pelatihan, loss pengujian, dan akurasi pengujian.

Waktu total pelatihan juga dicetak untuk evaluasi kinerja.

Dengan menggunakan loop ini, dapat melihat bagaimana model baseline (model_0) tampil selama beberapa epoch pada data FashionMNIST.

4. Make predictions and get Model 0 results

Dalam bagian ini, kita membuat fungsi untuk mengevaluasi model. Fungsi ini menerima model yang telah dilatih, DataLoader, fungsi loss, dan fungsi akurasi sebagai argumen. Fungsi ini akan menggunakan model untuk membuat prediksi pada data dalam DataLoader dan mengevaluasi prediksi tersebut menggunakan fungsi loss dan fungsi akurasi.

In [ ]:
```python
torch.manual_seed(42)
def eval_model(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               accuracy_fn):
    """Returns a dictionary containing the results of model predicting on data_loader.

    Args:
        model (torch.nn.Module): A PyTorch model capable of making predictions on data_loader.
        data_loader (torch.utils.data.DataLoader): The target dataset to predict on.
        loss_fn (torch.nn.Module): The loss function of model.
        accuracy_fn: An accuracy function to compare the models predictions to the truth labels.

    Returns:
        (dict): Results of model making predictions on data_loader.
    """
    loss, acc = 0, 0
    model.eval()
    with torch.inference_mode():
        for X, y in data_loader:
            # Make predictions with the model
            y_pred = model(X)

            # Accumulate the loss and accuracy values per batch
            loss += loss_fn(y_pred, y)
            acc += accuracy_fn(y_true=y,
                               y_pred=y_pred.argmax(dim=1)) # For accuracy, need the prediction labels (logits -> pred_prob

        # Scale loss and acc to find the average loss/acc per batch
        loss /= len(data_loader)
        acc /= len(data_loader)

    return {"model_name": model.__class__.__name__, # only works when model was created with a class
            "model_loss": loss.item(),
            "model_acc": acc}

# Calculate model 0 results on test dataset
model_0_results = eval_model(model=model_0, data_loader=test_dataloader,
    loss_fn=loss_fn, accuracy_fn=accuracy_fn
)
model_0_results
```

Out[ ]:
```
{'model_name': 'FashionMNISTModelV0',
 'model_loss': 0.47663894295692444,
 'model_acc': 83.42651757188499}
```

Fungsi **eval_model** menerima model yang telah dilatih, DataLoader, fungsi loss, dan fungsi akurasi sebagai argumen.

Fungsi ini mengembalikan hasil prediksi model pada data DataLoader dalam bentuk dictionary.

Hasil yang dihasilkan melibatkan nama model, nilai loss model, dan akurasi model pada data DataLoader.

Hasil evaluasi model baseline (model_0) pada dataset uji (test) FashionMNIST ditampilkan dalam bentuk dictionary.

Dictionary ini dapat digunakan untuk membandingkan hasil model baseline dengan model lain yang akan dibangun selanjutnya.

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

Dalam bagian ini, kita menyiapkan kode yang agnostik perangkat, yang memungkinkan penggunaan GPU jika tersedia. Hal ini berguna untuk memaksimalkan kecepatan pelatihan model, terutama jika kita memiliki akses ke unit pemrosesan grafis (GPU).

In [ ]:
```python
# Setup device agnostic code
import torch
device = "cuda" if torch.cuda.is_available() else "cpu"
device
```

Out[ ]:
```
'cuda'
```

Kode ini menentukan perangkat yang akan digunakan untuk pelatihan model.

Jika GPU tersedia, perangkat akan diatur sebagai "cuda" (GPU), jika tidak, akan diatur sebagai "cpu" (CPU).

Keberadaan GPU dapat mempercepat pelatihan model, terutama untuk model dan dataset yang lebih besar.