

**TECHNICAL REPORT
MACHINE LEARNING**



Disusun oleh:

**Agung Aji Saputra
1103202114**

**PROGRAM STUDI TEKNIK KOMPUTER
FAKULTAS TEKNIK ELEKTRO
TELKOM UNIVERSITY**

2024

PyTorch adalah sebuah framework machine learning open-source yang digunakan untuk mengembangkan dan melatih model deep learning. Framework ini sangat populer di kalangan peneliti dan praktisi machine learning karena kemudahan penggunaannya, fleksibilitas, dan dukungannya terhadap dynamic computation graphs. Berikut adalah beberapa poin penting tentang PyTorch dalam konteks machine learning:

1. **Dynamic Computational Graphs:** Salah satu fitur kunci PyTorch adalah penggunaan dynamic computational graphs. Ini berarti kita dapat membangun, mengubah, dan mengoptimalkan graf secara dinamis saat program berjalan, yang memudahkan eksperimen dan pengembangan model.
2. **Tensor Operations:** PyTorch menyediakan struktur data utama yang disebut "Tensor" untuk merepresentasikan data numerik. Tensor mirip dengan array multidimensi dan mendukung operasi matematika efisien, seperti penjumlahan, perkalian, dan fungsi-fungsi lainnya, yang sangat penting dalam deep learning.
3. **Autograd:** Sistem autograd PyTorch memungkinkan perhitungan gradien otomatis. Ini berarti saat kita melatih model, PyTorch secara otomatis melacak setiap operasi yang melibatkan parameter dan dapat menghitung gradiennya untuk tujuan optimisasi.
4. **Module dan Optimizer:** PyTorch menyediakan kelas `nn.Module` untuk membangun arsitektur model. Ini memudahkan dalam mengorganisir dan merancang struktur model deep learning. Selain itu, terdapat berbagai optimizers seperti SGD, Adam, dan lainnya yang dapat digunakan untuk melatih model.
5. **Dukungan GPU:** PyTorch dapat diintegrasikan dengan perangkat keras GPU untuk mempercepat pelatihan model. Ini memberikan kecepatan pelatihan yang lebih tinggi dibandingkan dengan menggunakan CPU saja.
6. **Komunitas dan Dokumentasi:** PyTorch memiliki komunitas yang aktif dan dukungan dokumentasi yang baik. Hal ini membuatnya mudah untuk memahami dan memecahkan masalah yang mungkin muncul selama pengembangan model.
7. **Libraries dan Ecosystem:** PyTorch memiliki beragam pustaka dan ekosistem yang mendukung pengembangan model, seperti torchvision untuk visi komputer, torchtext untuk pemrosesan bahasa alami, dan lainnya. Ini mempermudah pengguna untuk memanfaatkan fungsi-fungsi khusus untuk berbagai tugas machine learning.

Dengan fitur-fitur ini, PyTorch menjadi pilihan yang populer di kalangan peneliti dan praktisi machine learning untuk membangun, melatih, dan menerapkan model deep learning dengan mudah dan efisien.

Chapter 00

```
import torch
torch.__version__

'2.1.0+cu121'

[ ] # Scalar
scalar = torch.tensor(7)
scalar

tensor(7)

[ ] scalar.ndim

0

[ ] # Get the Python number within a tensor (only works with one-element tensors)
scalar.item()

7

[ ] # Vector
vector = torch.tensor([7, 7])
vector

tensor([7, 7])

[ ] # Check the number of dimensions of vector
vector.ndim

1

[ ] # Check shape of vector
vector.shape

torch.Size([2])
```

Dengan melihat gambar di atas, dapat menemukan bahwa nilai tensor skalar adalah 7. Untuk menggunakan library PyTorch dan menampilkan versinya, kita dapat menggunakan perintah `torch.__version__`. Dengan perintah tersebut, kita dapat memeriksa versi PyTorch yang sedang digunakan. Atribut `ndim` digunakan untuk mengetahui jumlah dimensi dari tensor. Namun, untuk tensor skalar yang hanya memiliki satu elemen, nilai dimensinya adalah 0. Oleh karena itu, `scalar.ndim` akan menghasilkan nilai 0. Untuk mengambil nilai skalar dari tensor, kita dapat menggunakan metode `item()`; dalam contoh ini, metode tersebut akan mengembalikan nilai 7. Selanjutnya, kita membuat tensor vektor dengan dua elemen, yaitu `[7, 7]`. Sebagai informasi, vektor adalah tensor dengan satu dimensi.

```
TENSOR [2, 4, 5]]])

tensor([[[[1, 2, 3],
          [3, 6, 9],
          [2, 4, 5]]]])

[ ] # Check number of dimensions for TENSOR
TENSOR.ndim

3

And what about its shape?

[ ] # Check shape of TENSOR
TENSOR.shape

torch.Size([1, 3, 3])

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

(tensor([[0.9750, 0.3375, 0.8737, 0.5708],
         [0.2182, 0.0930, 0.1803, 0.2081],
         [0.7768, 0.3430, 0.9639, 0.4490]]),
 torch.float32)

[ ] # 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

(torch.Size([224, 224, 3]), 3)

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

Dalam kode di atas, melakukan berbagai operasi menggunakan library PyTorch. Pertama, kita membuat sebuah matriks dengan nilai tertentu dan menampilkan matriks tersebut. Selanjutnya, kita

menggunakan atribut `ndim` untuk mengetahui jumlah dimensi dari matriks, yang ternyata adalah 2. Selain itu, kita juga menggunakan atribut `shape` untuk mendapatkan informasi tentang ukuran matriks. Kemudian, membuat sebuah tensor tiga dimensi dan mengecek dimensi serta ukurannya. Selanjutnya, kita membuat tensor acak dengan ukuran (3, 4) dan menampilkan tensor beserta tipe datanya. Terdapat juga pembuatan tensor acak dengan ukuran (224, 224, 3), yang dapat diartikan sebagai representasi gambar dengan tinggi 224, lebar 224, dan 3 saluran warna (RGB). Terakhir, kita membuat tensor berisi semua nilai nol dengan ukuran (3, 4) dan menampilkan tensor tersebut beserta tipe datanya. Semua operasi ini memberikan pemahaman yang baik tentang penggunaan PyTorch dalam membuat dan bekerja dengan matriks serta tensor pada pemrograman machine learning.

```
(tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]])
torch.float32)

# 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-19-a09072c806d9>:2: UserWarning: torch.range is deprecated and will be removed in a future release because
zero_to_ten_deprecated = torch.range(0, 10) # Note: this may return an error in the future
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# 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

tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

# Default datatype for tensors is float32
float_32_tensor = torch.tensor([3.0, 6.0, 9.0],
                                dtype=None, # defaults to None, which is torch.float32 or whatever datatype is passed
                                device=None, # defaults to None, which uses the default tensor type
                                requires_grad=False) # If True, operations performed on the tensor are recorded

float_32_tensor.shape, float_32_tensor.dtype, float_32_tensor.device
(torch.Size([3]), torch.float32, device(type='cpu'))

# float_16 tensor = torch.tensor([3.0, 6.0, 9.0],
#                                dtype=torch.float16) # torch.half would also work
float_16_tensor.dtype
```

Dalam kode di atas, melakukan berbagai operasi pembuatan dan manipulasi tensor menggunakan PyTorch. Pertama, kita membuat tensor yang diisi dengan nilai satu berukuran (3, 4). Selanjutnya, terdapat penggunaan fungsi `torch.range()` yang sudah tidak direkomendasikan (`deprecated`), yang dapat menyebabkan kesalahan di masa depan. Kemudian, kita membuat tensor rentang nilai dari 0 hingga 10 menggunakan `torch.arange()`. Selain itu, ada pembuatan tensor nol dengan bentuk yang sama seperti tensor sebelumnya. Selanjutnya, kita menciptakan tensor `float32` dengan nilai tertentu, dengan informasi tentang bentuk, tipe data, dan perangkat penyimpanan tensor ditampilkan. Terakhir, kita membuat tensor `float16` dan menampilkan tipe datanya. Semua operasi ini memberikan gambaran tentang cara membuat, mengonfigurasi tipe data, dan melakukan operasi dasar dengan tensor menggunakan PyTorch.

```
# 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([[0.4603, 0.5583, 0.1101, 0.2314],
        [0.8011, 0.6625, 0.9742, 0.7229],
        [0.3781, 0.9612, 0.2161, 0.1147]])
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Dalam kode di atas, membuat sebuah tensor menggunakan PyTorch dengan ukuran 3 baris dan 4

kolom, yang diisi dengan nilai acak antara 0 dan 1. Output dari tensor tersebut kemudian ditampilkan, memberikan kita gambaran tentang nilainya. Selanjutnya, kita mencetak informasi terkait tensor tersebut, termasuk bentuk (shape) dengan menggunakan `some_tensor.shape`, tipe data dengan `some_tensor.dtype`, dan perangkat penyimpanan tensor dengan `some_tensor.device`. Dalam contoh ini, tensor disimpan di CPU. Keseluruhan kode memberikan pemahaman yang baik tentang cara membuat tensor, mengekstrak informasi tentang tensor, dan mengetahui perangkat penyimpanan yang digunakan.

```
tensor([ 1.0000,  2.0000,  3.0000])
[ ] tensor + 10
    tensor([11, 12, 13])

[ ] # Multiply it by 10
    tensor * 10
    tensor([10, 20, 30])

[ ] # Tensors don't change unless reassigned
    tensor
    tensor([1, 2, 3])

[ ] # Subtract and reassign
    tensor = tensor - 10
    tensor
    tensor([-9, -8, -7])

[ ] # Add and reassign
    tensor = tensor + 10
    tensor
    tensor([1, 2, 3])

[ ] # Can also use torch functions
    torch.multiply(tensor, 10)
    tensor([10, 20, 30])

[ ] # Original tensor is still unchanged
    tensor
    tensor([1, 2, 3])

[ ] # Element-wise multiplication (each element multiplies its equivalent, index 0->0, 1->1, 2->2)
```

Dalam kode di atas, membuat tensor dengan nilai [1, 2, 3] menggunakan PyTorch. Kemudian, melakukan operasi penambahan dan perkalian terhadap tensor tersebut, dengan menambahkan 10 dan mengalikan dengan 10. Penting untuk dicatat bahwa operasi ini tidak mengubah tensor asli kecuali jika di-assign kembali. Setelah itu, kita menggunakan operasi pengurangan dan penggantian nilai tensor, yang kemudian diubah lagi dengan operasi penambahan. Selanjutnya, kita mencoba menggunakan fungsi `torch.multiply()` untuk mengalikan tensor dengan 10. Meskipun operasi ini berhasil, tensor asli tetap tidak berubah, menunjukkan bahwa tensor hanya berubah jika di-assign kembali. Keseluruhan kode memberikan gambaran tentang bagaimana operasi dasar seperti penambahan, perkalian, pengurangan, dan penggantian nilai dapat dilakukan pada tensor menggunakan PyTorch.

```

1 # Element-wise multiplication (each element multiplies its equivalent, index 0-0, 1-1, 2-2)
2 print(tensor, "*", tensor)
3 print("equals:", tensor * tensor)
4
5 tensor([1, 2, 3]) * tensor([1, 2, 3])
6 equals: tensor([1, 4, 9])
7
8 import torch
9 tensor = torch.tensor([1, 2, 3])
10 tensor.shape
11
12 torch.Size([3])
13
14 # Element-wise matrix multiplication
15 tensor * tensor
16
17 tensor([1, 4, 9])
18
19 # Matrix multiplication
20 torch.matmul(tensor, tensor)
21
22 tensor(14)
23
24 # Can also use the "@" symbol for matrix multiplication, though not recommended
25 tensor @ tensor
26
27 tensor(14)
28
29 %%time
30 # Matrix multiplication by hand
31 # (avoid doing operations with for loops at all cost, they are computationally expensive)
32 value = 0
33 for i in range(len(tensor)):
34     value += tensor[i] * tensor[i]
35 value
36
37 CPU times: user 309 µs, sys: 60 µs, total: 369 µs
38 Wall time: 2.15 ms

```

Dalam kode di atas, mulai dengan mencetak tensor dan hasil perkaliannya dengan dirinya sendiri menggunakan operator *. Setelah itu, menggunakan library PyTorch untuk membuat tensor dengan nilai [1, 2, 3] dan mengecek bentuknya menggunakan tensor.shape, yang menghasilkan output berupa jumlah elemen dalam tensor. Selanjutnya, melakukan perkalian elemen-wise pada tensor dengan dirinya sendiri menggunakan operator *, dan melakukan perkalian matriks menggunakan torch.matmul(). Selain itu, kita mencoba menggunakan simbol @ untuk perkalian matriks, meskipun tidak direkomendasikan. Terakhir, kita melihat waktu yang diperlukan untuk melakukan perkalian matriks secara manual dengan menggunakan for loop. Penting untuk dihindari penggunaan for loop dalam operasi matriks karena dapat menjadi mahal secara komputasi. Keseluruhan kode memberikan gambaran tentang berbagai metode untuk melakukan operasi matriks, dari perkalian elemen-wise hingga perkalian matriks, menggunakan PyTorch.

```

%%time
torch.matmul(tensor, tensor)

CPU times: user 35 µs, sys: 7 µs, total: 42 µs
Wall time: 45.3 µs
tensor(14)

```

%%time diikuti oleh torch.matmul(tensor, tensor) digunakan untuk mengukur waktu eksekusi dari operasi perkalian matriks yang dilakukan oleh torch.matmul(). Hasil output dari pengukuran waktu menunjukkan bahwa waktu yang diperlukan untuk operasi ini adalah sekitar 45.3 mikrodetik (µs)

```

# Since the linear layer starts with a random weights matrix, let's make it reproducible (more on this later)
torch.manual_seed(42)
# This uses matrix multiplication
linear = torch.nn.Linear(in_features=2, # in_features = matches inner dimension of input
                        out_features=6) # out_features = describes outer value
x = tensor_A
output = linear(x)
print(f"Input shape: {x.shape}\n")
print(f"Output:\n{output}\n\nOutput shape: {output.shape}")

Input shape: torch.Size([3, 2])

Output:
tensor([[2.2368, 1.2292, 0.4714, 0.3864, 0.1309, 0.9838],
        [4.4919, 2.1970, 0.4469, 0.5285, 0.3481, 2.4777],
        [6.7469, 3.1648, 0.4224, 0.6705, 0.5493, 3.9716]])
grad_fn=<AddmmBackward0>

Output shape: torch.Size([3, 6])

```

Kode tersebut menunjukkan penggunaan layer linear pada PyTorch dengan memastikan hasil yang dapat direproduksi melalui penetapan seed generator acak. Dengan menciptakan objek linear

layer, tensor input dari tensor_A digunakan untuk menghasilkan output, yang memiliki bentuk (3, 6) setelah operasi matrix multiplication. Hasil output dan informasi tambahan tentang operasi backward dicetak, memberikan contoh sederhana dari transformasi linear pada PyTorch dengan memperhatikan reproduktibilitas pembuatan matriks bobot awal.

```
x = torch.arange(0, 100, 10)
x

tensor([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])

# let's perform some aggregation.

print(f"Minimum: {x.min()}")
print(f"Maximum: {x.max()}")
# print(f"Mean: {x.mean()}") # this will error
print(f"Mean: {x.type(torch.float32).mean()}") # won't work without float datatype
print(f"Sum: {x.sum()}")

Minimum: 0
Maximum: 90
Mean: 45.0
Sum: 450

torch.max(x), torch.min(x), torch.mean(x.type(torch.float32)), torch.sum(x)

(tensor(90), tensor(0), tensor(45.), tensor(450))

# Create a tensor
tensor = torch.arange(10, 100, 10)
print(f"Tensor: {tensor}")

# Returns index of max and min values
print(f"Index where max value occurs: {tensor.argmax()}")
print(f"Index where min value occurs: {tensor.argmin()}")

Tensor: tensor([10, 20, 30, 40, 50, 60, 70, 80, 90])
Index where max value occurs: 8
Index where min value occurs: 0

# Create a tensor and check its datatype
tensor = torch.arange(10., 100., 10.)
tensor.dtype
```

```
tensor_float16 = tensor.type(torch.float16)
tensor_float16

tensor([10., 20., 30., 40., 50., 60., 70., 80., 90.], dtype=torch.float16)

# we can do something similar to make a torch.int8 tensor.

# Create a int8 tensor
tensor_int8 = tensor.type(torch.int8)
tensor_int8

tensor([10, 20, 30, 40, 50, 60, 70, 80, 90], dtype=torch.int8)

# Create a tensor
import torch
x = torch.arange(1., 8.)
x, x.shape

tensor([1., 2., 3., 4., 5., 6., 7.]), torch.Size([7])

# let's add an extra dimension with torch.reshape().

# Add an extra dimension
x_resaped = x.reshape(1, 7)
x_resaped, x_resaped.shape

tensor([[1., 2., 3., 4., 5., 6., 7.]]) torch.Size([1, 7])

# we can also change the view with torch.view().

# Change view (keeps same data as original but changes view)
# See more: https://stackoverflow.com/a/54507446/7900723
z = x.view(1, 7)
z, z.shape
```

```
[ ] # Stack tensors on top of each other
x_stacked = torch.stack([x, x, x, x], dim=0) # try changing dim to dim=1 and see what happens
x_stacked

tensor([[[5., 2., 3., 4., 5., 6., 7.],
         [5., 2., 3., 4., 5., 6., 7.],
         [5., 2., 3., 4., 5., 6., 7.],
         [5., 2., 3., 4., 5., 6., 7.]])

print(f"Previous tensor: {x_resized}")
print(f"Previous shape: {x_resized.shape}")

# Remove extra dimension from x_resized
x_squeezed = x_resized.squeeze()
print(f"\nNew tensor: {x_squeezed}")
print(f"New shape: {x_squeezed.shape}")

Previous tensor: tensor([[[5., 2., 3., 4., 5., 6., 7.]])
Previous shape: torch.Size([1, 7])

New tensor: tensor([5., 2., 3., 4., 5., 6., 7.])
New shape: torch.Size([7])

print(f"Previous tensor: {x_squeezed}")
print(f"Previous shape: {x_squeezed.shape}")

## Add an extra dimension with unsqueeze
x_unsqueezed = x_squeezed.unsqueeze(dim=0)
print(f"\nNew tensor: {x_unsqueezed}")
print(f"New shape: {x_unsqueezed.shape}")

Previous tensor: tensor([5., 2., 3., 4., 5., 6., 7.])
Previous shape: torch.Size([7])
```

Dalam kode tersebut, sebuah tensor x dibuat dengan rentang nilai dari 0 hingga 90 dengan langkah 10, dan informasi statistik seperti nilai minimum, maksimum, rata-rata, dan jumlah dari tensor tersebut dicetak. Untuk menghindari kesalahan, perlu dilakukan konversi tipe data saat menghitung rata-rata. Selanjutnya, sebuah tensor lain tensor dibuat, dan fungsi `argmax()` dan `argmin()` digunakan untuk mendapatkan indeks nilai maksimum dan minimum. Kode juga menunjukkan cara mengubah tipe data tensor, seperti ke `float16` atau `int8`. Selain itu, contoh manipulasi dimensi tensor diberikan, termasuk menambah, menghapus, dan kembali menambah dimensi. Potongan kode tersebut memberikan gambaran tentang berbagai operasi pada tensor PyTorch, termasuk perhitungan statistik, manipulasi bentuk, dan konversi tipe data.

```
import torch

# Create two random tensors
random_tensor_A = torch.rand(3, 4)
random_tensor_B = torch.rand(3, 4)

print(f"Tensor A:\n{random_tensor_A}\n")
print(f"Tensor B:\n{random_tensor_B}\n")
print(f"Does Tensor A equal Tensor B? (anywhere)")
random_tensor_A == random_tensor_B

Tensor A:
tensor([[0.8016, 0.3649, 0.6286, 0.9663],
        [0.7687, 0.4566, 0.5745, 0.9200],
        [0.3230, 0.8613, 0.0919, 0.3102]])

Tensor B:
tensor([[0.9536, 0.6002, 0.0351, 0.6826],
        [0.3743, 0.5220, 0.1336, 0.9666],
        [0.9754, 0.8474, 0.8988, 0.1105]])

Does Tensor A equal Tensor B? (anywhere)
tensor([[False, False, False, False],
        [False, False, False, False],
        [False, False, False, False]])
```

Kode di atas membuat dua tensor acak, `random_tensor_A` dan `random_tensor_B`, dengan bentuk (shape) 3x4 menggunakan fungsi `torch.rand()`. Selanjutnya, kedua tensor tersebut dicetak untuk memeriksa nilainya. Terakhir, sebuah pernyataan dicetak untuk mengevaluasi apakah elemen-elemen pada `random_tensor_A` sama dengan `random_tensor_B` di mana saja. Hasil dari pernyataan tersebut adalah tensor boolean yang menunjukkan elemen-elemen yang setara dan tidak setara antara kedua tensor. Namun, perlu diperhatikan bahwa dalam kode tersebut,

pernyataan v tidak memiliki kejelasan atau kegunaan, sehingga tampaknya terdapat kesalahan atau ketidaksempurnaan dalam penulisan kode tersebut.

```
## Set the random seed
RANDOM_SEED=42 # try changing this to different values and see what happens to the numbers below
torch.manual_seed(seed=RANDOM_SEED)
random_tensor_C = torch.rand(3, 4)

# Have to reset the seed every time a new rand() is called
# Without this, tensor_D would be different to tensor_C
torch.random.manual_seed(seed=RANDOM_SEED) # try commenting this line out and seeing what happens
random_tensor_D = torch.rand(3, 4)

print(f"Tensor C:\n{random_tensor_C}\n")
print(f"Tensor D:\n{random_tensor_D}\n")
print(f"Does Tensor C equal Tensor D? (anywhere)")
random_tensor_C == random_tensor_D

Tensor C:
tensor([[[0.8823, 0.9150, 0.3829, 0.9593],
         [0.3904, 0.6009, 0.2566, 0.7936],
         [0.9408, 0.1332, 0.9346, 0.5936]]])

Tensor D:
tensor([[[0.8823, 0.9150, 0.3829, 0.9593],
         [0.3904, 0.6009, 0.2566, 0.7936],
         [0.9408, 0.1332, 0.9346, 0.5936]]])

Does Tensor C equal Tensor D? (anywhere)
tensor([[[True, True, True, True],
         [True, True, True, True],
         [True, True, True, True]]])
```

Dalam kode tersebut, menetapkan seed acak menggunakan `torch.manual_seed()` dan `torch.random.manual_seed()` untuk memastikan keberulangan hasil dari fungsi `torch.rand()`. Dua tensor acak, `random_tensor_C` dan `random_tensor_D`, kemudian dibuat dengan bentuk (shape) 3x4 menggunakan fungsi `torch.rand()`. Perlu diperhatikan bahwa penetapan seed menggunakan `torch.manual_seed()` diperlukan sebelum menciptakan `random_tensor_C` untuk memastikan reproduktibilitas, tetapi juga diperlukan untuk `torch.random.manual_seed()` sebelum menciptakan `random_tensor_D`. Tanpa penetapan seed pada kedua tahap tersebut, nilai tensor `random_tensor_D` dapat berbeda dari `random_tensor_C`. Hasil cetakan dari kedua tensor tersebut dan hasil perbandingan elemen-elemen mereka menunjukkan bahwa semua elemen pada `random_tensor_C` sama dengan `random_tensor_D`. Pada akhirnya, kode tersebut menunjukkan penggunaan seed acak untuk memastikan keberulangan hasil saat menciptakan tensor acak.

Chapter 01

```
# 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] if
(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]]))
```

Kode tersebut berfungsi untuk membuat data linear sederhana menggunakan framework PyTorch di Python. Dua parameter yang sudah diketahui, yaitu weight dan bias, didefinisikan terlebih dahulu. Selanjutnya, data input X dibuat dengan menggunakan fungsi torch.arange, dan data output y dihasilkan sesuai dengan persamaan linear sederhana

```
train_split = int(0.8 * len(X))
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)

(40, 40, 10, 10)

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))
    plt.scatter(train_data, train_labels, c="b", s=4, label="Training data")
    plt.scatter(test_data, test_labels, c="g", s=4, label="Testing data")

    if predictions is not None:
        plt.scatter(test_data, predictions, c="r", s=4, label="Predictions")
    plt.legend(prop={"size": 14});

plot_predictions();
```

Kode tersebut pertama-tama melakukan pembagian dataset menjadi data pelatihan (X_train dan y_train) dan

data pengujian (X_{test} dan y_{test}) menggunakan proporsi 80-20. Kemudian, panjang masing-masing set data dihitung untuk memverifikasi pembagian yang benar. Fungsi `plot_predictions` kemudian didefinisikan untuk membuat plot yang memvisualisasikan data pelatihan dan pengujian dengan menggunakan warna berbeda

```
torch.manual_seed(42)

model_0 = LinearRegressionModel({})

list(model_0.parameters())

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

model_0.state_dict()

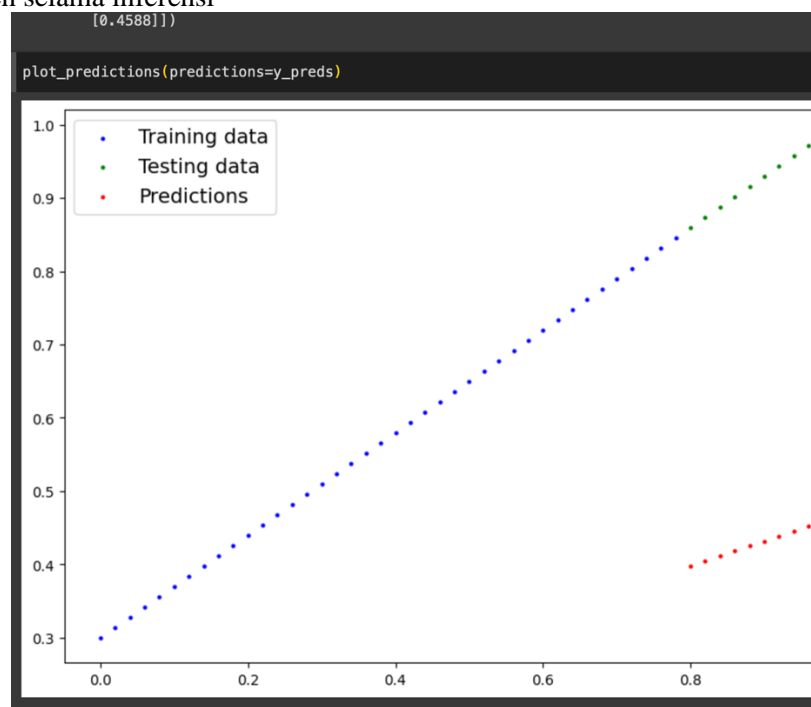
OrderedDict([('weights', tensor([0.3367])), ('bias', tensor([0.1288]))])

with torch.inference_mode():
    y_preds = model_0(X_test)

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]])
```

seed untuk generator angka acak PyTorch diatur ke 42 dengan `torch.manual_seed(42)`, sehingga hasil dapat direproduksi. Kemudian, objek model regresi linear (`model_0`) dibuat menggunakan kelas yang tidak didefinisikan dalam contoh kode ini. Fungsi `list(model_0.parameters())` digunakan untuk menampilkan parameter-parameter model. `model_0.state_dict()` memberikan representasi kamus dari status model, yang mencakup parameter dan tensor lainnya. Selanjutnya, prediksi model pada data pengujian (X_{test}) dilakukan dengan menggunakan `torch.inference_mode()`. Hasil prediksi disimpan dalam variabel `y_preds`, dan informasi tentang jumlah sampel pengujian dan panjang hasil prediksi ditampilkan. Namun, ada sebuah kesalahan di kode: `torch.inference_mode()` seharusnya diganti dengan `torch.no_grad()` untuk menonaktifkan perhitungan gradien selama inferensi



`plot_predictions(predictions=y_preds)` berfungsi untuk memvisualisasikan data pelatihan (biru), data pengujian (hijau), dan prediksi model (merah) pada grafik scatter plot. Fungsi ini memanfaatkan `matplotlib` untuk membuat plot dan menerima argumen `predictions`

```
test_loss_values = []
epoch_count = []

for epoch in range(epochs):
    model_0.train()
    y_pred = model_0(X_train)
    loss = loss_fn(y_pred, y_train)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    model_0.eval()

    with torch.inference_mode():
        test_pred = model_0(X_test)

        test_loss = loss_fn(test_pred, y_test.type(torch.float))

        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
```

Kode tersebut melibatkan pelatihan model regresi linear dengan PyTorch selama 100 epoch. Proses pelatihan mencakup perhitungan kerugian pada set pelatihan dan pengujian, serta pembaruan model menggunakan algoritma optimasi. Hasil keluaran mencakup informasi mengenai nilai Mean Absolute Error pada set pelatihan dan pengujian setiap 10 epoch, memberikan gambaran kinerja model selama pelatihan.

```
class LinearRegressionModelV2(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear_layer = nn.Linear(in_features=1,
                                       out_features=1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.linear_layer(x)

torch.manual_seed(42)
model_1 = LinearRegressionModelV2()
model_1, model_1.state_dict()

(LinearRegressionModelV2(
  (linear_layer): Linear(in_features=1, out_features=1, bias=True)
),
OrderedDict([('linear_layer.weight', tensor([0.7645])),
             ('linear_layer.bias', tensor([0.8300]))])
```

Dalam kodingan tersebut, sebuah model regresi linear baru (`LinearRegressionModelV2`) didefinisikan menggunakan modul PyTorch (`nn.Module`). Model ini memiliki satu layer linear dengan input features sejumlah 1 dan output features sejumlah 1. Selanjutnya, seed untuk generator angka acak PyTorch diatur ke 42 dengan `torch.manual_seed(42)`. Objek model (`model_1`) diinisialisasi dan dicetak bersama dengan state dictionary-nya, yang memuat informasi tentang bobot (`weight`) dan bias dari layer linear. Hasil output menunjukkan bahwa bobot memiliki nilai sekitar 0.7645 dan bias sekitar 0.8300

```

torch.manual_seed(42)

epochs = 1000

X_train = X_train.to(device)
X_test = X_test.to(device)
y_train = y_train.to(device)
y_test = y_test.to(device)

for epoch in range(epochs):
    model_1.train()
    y_pred = model_1(X_train)

    loss = loss_fn(y_pred, y_train)

    optimizer.zero_grad()

    loss.backward()

    optimizer.step()
    model_1.eval()
    with torch.inference_mode():
        test_pred = model_1(X_test)
        test_loss = loss_fn(test_pred, y_test)

    if epoch % 100 == 0:
        print(f'Epoch: {epoch} | Train loss: {loss} | Test loss: {test_loss}')

Epoch: 0 | Train loss: 0.5551779866218567 | Test loss: 0.5739762187004089
Epoch: 100 | Train loss: 0.006215679459273815 | Test loss: 0.014086711220443249
Epoch: 200 | Train loss: 0.0012645035749301314 | Test loss: 0.013801807537674904
Epoch: 300 | Train loss: 0.0012645035749301314 | Test loss: 0.013801807537674904
Epoch: 400 | Train loss: 0.0012645035749301314 | Test loss: 0.013801807537674904
Epoch: 500 | Train loss: 0.0012645035749301314 | Test loss: 0.013801807537674904
Epoch: 600 | Train loss: 0.0012645035749301314 | Test loss: 0.013801807537674904
Epoch: 700 | Train loss: 0.0012645035749301314 | Test loss: 0.013801807537674904
Epoch: 800 | Train loss: 0.0012645035749301314 | Test loss: 0.013801807537674904
Epoch: 900 | Train loss: 0.0012645035749301314 | Test loss: 0.013801807537674904

```

Dalam kodingan tersebut, model regresi linear (model_1) dilatih menggunakan data pelatihan (X_train dan y_train) selama 1000 epoch. Proses pelatihan melibatkan komputasi prediksi model (y_pred), perhitungan kerugian menggunakan fungsi kerugian (loss_fn), dan optimisasi parameter model dengan algoritma optimasi (optimizer). Setelah setiap 100 epoch, nilai kerugian pada set pelatihan dan pengujian (X_test dan y_test) dicetak ke layar. Hasil output menunjukkan bahwa seiring berjalannya epoch, nilai kerugian pada set pelatihan terus berkurang, mencapai nilai yang sangat rendah (sekitar 0.001) pada akhirnya.

```

from pprint import pprint
print("The model learned the following values for weights and bias:")
pprint(model_1.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([('linear_layer.weight', tensor([[0.6968]])),
            ('linear_layer.bias', tensor([0.3025]))])

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

model_1.eval()

with torch.inference_mode():
    y_preds = model_1(X_test)
y_preds

tensor([[0.8600],
        [0.8739],
        [0.8878],
        [0.9018],
        [0.9157],
        [0.9296],
        [0.9436],
        [0.9575],
        [0.9714],
        [0.9854]])

```

Kode di atas bertujuan untuk menampilkan nilai bobot (weight) dan bias dari model regresi linear (model_1) setelah proses pelatihan. Hasil output menunjukkan bahwa model telah mempelajari nilai-nilai baru untuk bobot dan bias, yaitu bobot sekitar 0.6968 dan bias sekitar 0.3025. Ini menunjukkan bahwa setelah pelatihan, model telah mengoptimalkan parameter-parameter tersebut untuk mencocokkan data pelatihan dengan lebih baik. Perbandingan dengan nilai awal yang ditentukan sebelumnya (weight=0.7 dan bias=0.3) model regresi linear (model_1) diubah ke mode evaluasi dengan menggunakan metode .eval(), dan kemudian dilakukan inferensi pada data pengujian (X_test) untuk mendapatkan prediksi (y_preds). Hasil output menunjukkan tensor yang berisi nilai-nilai prediksi yang dihasilkan oleh model untuk setiap sampel dalam data pengujian.

```

from pathlib import Path
MODEL_PATH = Path("models")
MODEL_PATH.mkdir(parents=True, exist_ok=True)
MODEL_NAME = "01_pytorch_workflow_model_1.pth"
MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME
print(f"Saving model to: {MODEL_SAVE_PATH}")
torch.save(model_1.state_dict(),
           f=MODEL_SAVE_PATH)

Saving model to: models/01_pytorch_workflow_model_1.pth

loaded_model_1 = LinearRegressionModelV2()

loaded_model_1.load_state_dict(torch.load(MODEL_SAVE_PATH))

loaded_model_1.to(device)

print(f"Loaded model:\n{loaded_model_1}")
print(f"Model on device:\n{next(loaded_model_1.parameters()).device}")

Loaded model:
LinearRegressionModelV2(
  (linear_layer): Linear(in_features=1, out_features=1, bias=True)
)
Model on device:
cpu

```

Kodingan tersebut bertujuan untuk menyimpan parameter-parameter dari model regresi linear (model_1) setelah proses pelatihan. Path untuk menyimpan model ditentukan sebagai "models/01_pytorch_workflow_model_1.pth" dengan menggunakan library Path dari Python. Jika direktori "models" belum ada, maka akan dibuat. Selanjutnya, model disimpan pada path tersebut menggunakan fungsi torch.save(). Hasil output menunjukkan bahwa model telah berhasil disimpan ke dalam file dengan nama "01_pytorch_workflow_model_1.pth" dalam direktori "models" model regresi linear (loaded_model_1) baru dibuat menggunakan kelas LinearRegressionModelV2. Selanjutnya, parameter-parameter dari model tersebut diinisialisasi ulang menggunakan parameter yang telah disimpan sebelumnya. Ini dilakukan dengan memuat state dictionary dari model yang telah disimpan ke dalam loaded_model_1 menggunakan fungsi load_state_dict(torch.load(MODEL_SAVE_PATH)). Setelah itu, model dipindahkan ke perangkat yang ditentukan (device) dengan menggunakan metode .to(device).

```

loaded_model_1.eval()
with torch.inference_mode():
    loaded_model_1_preds = loaded_model_1(X_test)
y_preds == loaded_model_1_preds

tensor([[True],
        [True],
        [True],
        [True],
        [True],
        [True],
        [True],
        [True],
        [True]])

```

semua elemen dalam tensor output memiliki nilai True, yang menunjukkan bahwa nilai prediksi dari model yang dimuat sama dengan nilai prediksi

Chapter 02

```
print(f"First 5 X features:\n{X[:5]}")
print(f"\nFirst 5 y labels:\n{y[:5]}")

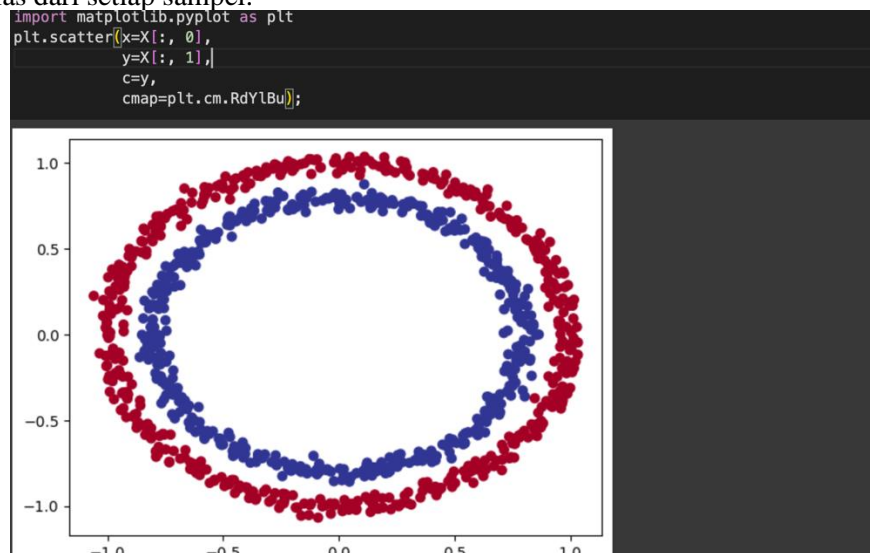
First 5 X features:
[[ 0.75424625  0.23148074]
 [-0.75615888  0.15325888]
 [-0.81539193  0.17328203]
 [-0.39373073  0.69288277]
 [ 0.44220765 -0.89672343]]

First 5 y labels:
[1 1 1 1 0]

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

	X1	X2	label
0	0.754246	0.231481	1
1	-0.756159	0.153259	1
2	-0.815392	0.173282	1

Kode tersebut mencetak lima sampel pertama dari fitur-input (X) dan label-output (y). Hasil output menunjukkan bahwa setiap sampel dalam fitur-input (X) memiliki dua nilai, dan lima sampel tersebut membentuk matriks 2D. Sedangkan, label-output (y) menunjukkan lima nilai biner (0 atau 1). Dengan demikian, setiap sampel fitur-input memiliki dua nilai, dan label-outputnya adalah biner yang menunjukkan kategori atau kelas dari setiap sampel.



Kodingan tersebut menggunakan matplotlib untuk membuat scatter plot dari data yang terdiri dari dua fitur ($X[:, 0]$ dan $X[:, 1]$). Setiap titik dalam plot diwarnai berdasarkan nilai label-output (y) dengan menggunakan colormap `plt.cm.RdYlBu`. Warna merah (RdYlBu) menunjukkan nilai label-output yang lebih tinggi, sedangkan warna biru menunjukkan nilai yang lebih rendah.


```

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: ()

Turn data into tensors and create train and test splits

import torch
X = torch.from_numpy(X).type(torch.float)
y = torch.from_numpy(y).type(torch.float)

X[:5], y[:5]

(tensor([[ 0.7542,  0.2315],
        [-0.7562,  0.1533],
        [-0.6194,  0.1733],
        [-0.3937,  0.6929],
        [ 0.4422, -0.8967]]),
 tensor([1.,  1.,  1.,  1.,  0.]))

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,
                                                    random_state=42)

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

(800, 200, 800, 200)

```

Kodingan tersebut mencetak nilai dan bentuk (shape) dari satu sampel dari fitur-input (X) dan label-output (y). Hasil output menunjukkan bahwa satu sampel fitur-input (X_sample) terdiri dari dua nilai (0.75424625 dan 0.23148074) dan bentuknya adalah satu dimensi (2,). Sedangkan, satu sampel label-output (y_sample) memiliki nilai 1 dan bentuknya adalah nol dimensi (). Kemudian menggunakan PyTorch untuk mengonversi data dari NumPy arrays menjadi tensors. Dengan menggunakan torch.from_numpy(), fitur-input (X) dan label-output (y) diubah menjadi tensors PyTorch dan diberi tipe data float. Hasil output menunjukkan lima sampel pertama dari fitur-input (X[:5]) dan label-output (y[:5]) dalam bentuk tensors. Lalu menggunakan train_test_split dari scikit-learn untuk membagi dataset menjadi data pelatihan dan pengujian. Argumen X dan y mewakili fitur-input dan label-output. Parameter test_size=0.2 menunjukkan bahwa 20% dari data akan dialokasikan sebagai data pengujian. random_state=42 digunakan untuk memastikan reproducibility atau hasil yang dapat direproduksi

```

class CircleModelV0(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(in_features=2, out_features=5)
        self.layer_2 = nn.Linear(in_features=5, out_features=1)
    def forward(self, x):
        return self.layer_2(self.layer_1(x))
model_0 = CircleModelV0().to(device)
model_0

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

model_0 = nn.Sequential(
  nn.Linear(in_features=2, out_features=5),
  nn.Linear(in_features=5, out_features=1)
).to(device)
model_0

Sequential(
  (0): Linear(in_features=2, out_features=5, bias=True)
  (1): Linear(in_features=5, out_features=1, bias=True)
)

```

output menunjukkan arsitektur dari model tersebut, yang terdiri dari dua layer linear dengan ukuran input dan output yang sesuai. to(device) digunakan untuk memindahkan model ke perangkat yang ditentukan (device), yang dapat menjadi CPU atau GPU, sesuai dengan konfigurasi sistem. kemudian output menunjukkan arsitektur dari model tersebut dalam bentuk Sequential. Dua layer linear yang digunakan dijelaskan dalam output, dengan ukuran input dan output yang sesuai. Penggunaan Sequential mempermudah pendefinisian model dengan cara yang lebih ringkas dan mudah dibaca,


```

torch.manual_seed(42)
epochs = 100

X_train, y_train = X_train.to(device), y_train.to(device)
X_test, y_test = X_test.to(device), y_test.to(device)

for epoch in range(epochs):
    model_0.train()
    y_logits = model_0(X_train).squeeze()
    y_pred = torch.round(torch.sigmoid(y_logits))
    loss = loss_fn(y_logits, y_train)
    acc = accuracy_fn(y_true=y_train, y_pred=y_pred)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    model_0.eval()
    with torch.inference_mode():
        test_logits = model_0(X_test).squeeze()
        test_pred = torch.round(torch.sigmoid(test_logits))
        test_loss = loss_fn(test_logits, y_test)
        test_acc = accuracy_fn(y_true=y_test, y_pred=test_pred)

    if epoch % 10 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc:.2f}% | Test loss: {test_loss:.5f}, Test acc: {test_acc:.2f}%")

Epoch: 0 | Loss: 0.69488, Accuracy: 50.50% | Test loss: 0.69504, Test acc: 54.00%
Epoch: 10 | Loss: 0.69422, Accuracy: 50.00% | Test loss: 0.69464, Test acc: 51.00%
Epoch: 20 | Loss: 0.69391, Accuracy: 49.62% | Test loss: 0.69450, Test acc: 50.50%
Epoch: 30 | Loss: 0.69372, Accuracy: 49.38% | Test loss: 0.69446, Test acc: 51.00%
Epoch: 40 | Loss: 0.69360, Accuracy: 49.12% | Test loss: 0.69445, Test acc: 50.00%
Epoch: 50 | Loss: 0.69350, Accuracy: 49.38% | Test loss: 0.69445, Test acc: 49.50%
Epoch: 60 | Loss: 0.69342, Accuracy: 49.50% | Test loss: 0.69446, Test acc: 49.00%
Epoch: 70 | Loss: 0.69336, Accuracy: 49.75% | Test loss: 0.69447, Test acc: 49.50%

```

output menunjukkan perkembangan pelatihan setiap 10 epoch, dengan mencantumkan nilai kerugian dan akurasi pada set pelatihan serta nilai kerugian dan akurasi pada set pengujian. Terlihat bahwa selama pelatihan, nilai kerugian pada set pelatihan dan pengujian tidak mengalami perubahan yang signifikan, dan akurasi pun tidak meningkat secara konsisten

```

class CircleModelV1(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(in_features=2, out_features=10)
        self.layer_2 = nn.Linear(in_features=10, out_features=10) # extra layer
        self.layer_3 = nn.Linear(in_features=10, out_features=1)

    def forward(self, x):
        return self.layer_3(self.layer_2(self.layer_1(x)))

model_1 = CircleModelV1().to(device)
model_1

CircleModelV1(
  (layer_1): Linear(in_features=2, out_features=10, bias=True)
  (layer_2): Linear(in_features=10, out_features=10, bias=True)
  (layer_3): Linear(in_features=10, out_features=1, bias=True)
)

```

output menunjukkan arsitektur dari model tersebut, yaitu tiga layer linear yang terdiri dari layer_1, layer_2, dan layer_3. Model tersebut telah dipindahkan ke perangkat yang ditentukan (device),

```

torch.manual_seed(42)

epochs = 1000
X_train, y_train = X_train.to(device), y_train.to(device)
X_test, y_test = X_test.to(device), y_test.to(device)

for epoch in range(epochs):
    y_logits = model_1(X_train).squeeze()
    y_pred = torch.round(torch.sigmoid(y_logits))
    loss = loss_fn(y_logits, y_train)
    acc = accuracy_fn(y_true=y_train, y_pred=y_pred)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    model_1.eval()
    with torch.inference_mode():
        test_logits = model_1(X_test).squeeze()
        test_pred = torch.round(torch.sigmoid(test_logits))
        test_loss = loss_fn(test_logits, y_test)
        test_acc = accuracy_fn(y_true=y_test, y_pred=test_pred)

    if epoch % 100 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc:.2f}% | Test loss: {test_loss:.5f}, Test acc: {test_acc:.2f}%")

Epoch: 0 | Loss: 0.69396, Accuracy: 50.88% | Test loss: 0.69261, Test acc: 51.00%
Epoch: 100 | Loss: 0.69305, Accuracy: 50.38% | Test loss: 0.69379, Test acc: 48.00%
Epoch: 200 | Loss: 0.69299, Accuracy: 51.12% | Test loss: 0.69437, Test acc: 46.00%
Epoch: 300 | Loss: 0.69298, Accuracy: 51.62% | Test loss: 0.69458, Test acc: 45.00%
Epoch: 400 | Loss: 0.69298, Accuracy: 51.12% | Test loss: 0.69465, Test acc: 46.00%
Epoch: 500 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69467, Test acc: 46.00%
Epoch: 600 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%
Epoch: 700 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%
Epoch: 800 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%
Epoch: 900 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%

```

Kodingan tersebut merupakan implementasi pelatihan model neural network (model_1) pada data pelatihan (X_train dan y_train) dengan menggunakan PyTorch. Pelatihan dilakukan selama 1000 epoch. Dalam setiap epoch, model memperoleh prediksi (y_logits), menghasilkan prediksi biner (y_pred) dengan menggunakan fungsi aktivasi sigmoid dan pembulatan, menghitung nilai kerugian (loss) dan akurasi (acc) pada set

pelatihan, serta mengoptimalkan parameter model dengan menggunakan algoritma optimasi (optimizer)

```
weight = 0.7
bias = 0.3
start = 0
end = 1
step = 0.01
X_regression = torch.arange(start, end, step).unsqueeze(dim=1)
y_regression = weight * X_regression + bias

print(len(X_regression))
X_regression[:5], y_regression[:5]

100
(tensor([[0.0000],
         [0.0100],
         [0.0200],
         [0.0300],
         [0.0400]]),
 tensor([[0.3000],
         [0.3070],
         [0.3140],
         [0.3210],
         [0.3280]]))
```

X_regression memiliki panjang 100 (karena rentang dari 0 hingga 1 dengan langkah 0.01 menghasilkan 100 nilai). Kelima pasang nilai pertama dari X_regression dan y_regression juga ditampilkan, yang mencerminkan hubungan linier antara input dan output berdasarkan parameter model yang telah ditentukan.

```
def sigmoid(x):
    return 1 / (1 + torch.exp(-x))
sigmoid(A)

tensor([4.5398e-05, 1.2339e-04, 3.3535e-04, 9.1105e-04, 2.4726e-03, 6.6929e-03,
        1.7986e-02, 4.7426e-02, 1.1920e-01, 2.6894e-01, 5.0000e-01, 7.3106e-01,
        8.8080e-01, 9.5257e-01, 9.8201e-01, 9.9331e-01, 9.9753e-01, 9.9909e-01,
        9.9966e-01, 9.9988e-01])
```

Fungsi sigmoid yang didefinisikan mengimplementasikan fungsi aktivasi sigmoid, yang umum digunakan dalam jaringan saraf untuk mengonversi nilai kontinu menjadi rentang antara 0 dan 1.

```
import torch
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split

NUM_CLASSES = 4
NUM_FEATURES = 2
RANDOM_SEED = 42

X_blob, y_blob = make_blobs(n_samples=1000,
                             n_features=NUM_FEATURES,
                             centers=NUM_CLASSES,
                             cluster_std=1.5,
                             random_state=RANDOM_SEED)

X_blob = torch.from_numpy(X_blob).type(torch.float)
y_blob = torch.from_numpy(y_blob).type(torch.LongTensor)
print(X_blob[:5], y_blob[:5])
X_blob_train, X_blob_test, y_blob_train, y_blob_test = train_test_split(X_blob,
                                                                           y_blob,
                                                                           test_size=0.2,
                                                                           random_state=RANDOM_SEED)

plt.figure(figsize=(10, 7))
plt.scatter(X_blob[:, 0], X_blob[:, 1], c=y_blob, cmap=plt.cm.RdYlBu);

tensor([[ -8.4134,  6.9352],
        [-5.7665, -6.4312],
        [-6.0421, -6.7661],
        [ 3.9508,  0.6984],
        [ 4.2505, -0.2815]]) tensor([3, 2, 2, 1, 1])
```

Output menunjukkan beberapa contoh data dari dataset, di mana X_blob adalah tensor fitur dan y_blob adalah tensor label. Setiap baris X_blob mengandung dua nilai fitur, dan setiap elemen y_blob berisi label kelas yang sesuai

```

from torch import nn
class BlobModel(nn.Module):
    def __init__(self, input_features, output_features, hidden_units=8):
        """Initializes all required hyperparameters for a multi-class classification model.

        Args:
            input_features (int): Number of input features to the model.
            out_features (int): Number of output features of the model
            (how many classes there are).
            hidden_units (int): Number of hidden units between layers, default 8.
        """
        super().__init__()
        self.linear_layer_stack = nn.Sequential(
            nn.Linear(in_features=input_features, out_features=hidden_units),
            nn.Linear(in_features=hidden_units, out_features=hidden_units),
            nn.Linear(in_features=hidden_units, out_features=output_features),
        )

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

model_4 = BlobModel(input_features=NUM_FEATURES,
                    output_features=NUM_CLASSES,
                    hidden_units=8).to(device)

model_4

BlobModel(
  (linear_layer_stack): Sequential(
    (0): Linear(in_features=2, out_features=8, bias=True)
    (1): Linear(in_features=8, out_features=8, bias=True)
    (2): Linear(in_features=8, out_features=4, bias=True)
  )
)

```

Kodingan tersebut mendefinisikan sebuah model neural network untuk klasifikasi multi-kelas yang disebut BlobModel menggunakan modul PyTorch (nn.Module). Model ini memiliki tiga layer linear yang dihubungkan secara berurutan dengan menggunakan nn.Sequential.

```

model_4(X_blob_train.to(device))[:5]

tensor([[ -1.2711,  -0.6494,  -1.4740,  -0.7044],
        [  0.2210,  -1.5439,   0.0420,   1.1531],
        [  2.8698,   0.9143,   3.3169,   1.4027],
        [  1.9576,   0.3125,   2.2244,   1.1324],
        [  0.5458,  -1.2381,   0.4441,   1.1804]], grad_fn=<SliceBackward0>)

```

Output tersebut menunjukkan hasil prediksi dari model model_4 pada lima sampel pertama dari data latih X_blob_train. Hasil prediksi ini merupakan nilai keluaran dari model sebelum diterapkan fungsi softmax atau fungsi aktivasi lainnya yang dapat menghasilkan probabilitas kelas. Dalam hal ini, model_4 memberikan nilai logits untuk setiap kelas pada setiap sampel data.

```

epochs = 100

X_blob_train, y_blob_train = X_blob_train.to(device), y_blob_train.to(device)
X_blob_test, y_blob_test = X_blob_test.to(device), y_blob_test.to(device)

for epoch in range(epochs):
    model_4.train()

    y_logits = model_4(X_blob_train)
    y_pred = torch.softmax(y_logits, dim=1).argmax(dim=1)
    loss = loss_fn(y_logits, y_blob_train)
    acc = accuracy_fn(y_true=y_blob_train,
                     y_pred=y_pred)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    model_4.eval()

    with torch.inference_mode():
        test_logits = model_4(X_blob_test)
        test_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
        test_loss = loss_fn(test_logits, y_blob_test)
        test_acc = accuracy_fn(y_true=y_blob_test,
                             y_pred=test_pred)

    if epoch % 10 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Acc: {acc:.2f}% | Test Loss: {test_loss:.5f}, Test Acc: {test_acc:.2f}%")

Epoch: 0 | Loss: 1.04324, Acc: 65.50% | Test Loss: 0.57861, Test Acc: 95.50%
Epoch: 10 | Loss: 0.14398, Acc: 99.12% | Test Loss: 0.13037, Test Acc: 99.00%
Epoch: 20 | Loss: 0.08062, Acc: 99.12% | Test Loss: 0.07216, Test Acc: 99.50%
Epoch: 30 | Loss: 0.05924, Acc: 99.12% | Test Loss: 0.05133, Test Acc: 99.50%
Epoch: 40 | Loss: 0.04892, Acc: 99.00% | Test Loss: 0.04098, Test Acc: 99.50%
Epoch: 50 | Loss: 0.04295, Acc: 99.00% | Test Loss: 0.03486, Test Acc: 99.50%
Epoch: 60 | Loss: 0.03910, Acc: 99.00% | Test Loss: 0.03083, Test Acc: 99.50%
Epoch: 70 | Loss: 0.03643, Acc: 99.00% | Test Loss: 0.02799, Test Acc: 99.50%
Epoch: 80 | Loss: 0.03448, Acc: 99.00% | Test Loss: 0.02587, Test Acc: 99.50%
Epoch: 90 | Loss: 0.03300, Acc: 99.12% | Test Loss: 0.02423, Test Acc: 99.50%

```

Output tersebut mencakup hasil pelatihan model model_4 pada data latih (X_blob_train dan y_blob_train) selama beberapa epoch. Dalam setiap epoch, model dievaluasi pada data latih dan data uji untuk mengukur loss dan akurasi.

```

y_pred_probs = torch.softmax(y_logits, dim=1)

y_preds = y_pred_probs.argmax(dim=1)

print(f"Predictions: {y_preds[:10]}\nLabels: {y_blob_test[:10]}")
print(f"Test accuracy: {accuracy_fn(y_true=y_blob_test, y_pred=y_preds)}%")

Predictions: tensor([1, 3, 2, 1, 0, 3, 2, 0, 2, 0])
Labels: tensor([1, 3, 2, 1, 0, 3, 2, 0, 2, 0])
Test accuracy: 99.5%

```

Predictions vs Labels: Tensor `y_preds` berisi kelas yang diprediksi oleh model untuk setiap sampel dalam data uji. Tensor ini dibuat dengan mengambil `argmax` dari nilai probabilitas yang dihasilkan oleh `softmax`. Hasil prediksi ini dibandingkan dengan label sebenarnya dalam tensor `y_blob_test`. Test Accuracy: Akurasi pengujian dihitung menggunakan fungsi `accuracy_fn` yang menerima label sebenarnya (`y_blob_test`) dan prediksi (`y_preds`). Akurasi dinyatakan dalam persentase. Keakuratan Tinggi: Dalam contoh ini, model mencapai akurasi pengujian sebesar 99.5%, yang menunjukkan bahwa sebagian besar prediksinya sesuai dengan label sebenarnya pada data uji. Tingginya akurasi dapat diartikan sebagai kesesuaian yang baik antara prediksi model dan label sebenarnya.

```

try:
    from torchmetrics import Accuracy
except:
    !pip install torchmetrics==0.9.3
    from torchmetrics import Accuracy
    torchmetrics_accuracy = Accuracy(task='multiclass', num_classes=4).to(device)
    torchmetrics_accuracy(y_preds, y_blob_test)

Collecting torchmetrics==0.9.3
  Downloading torchmetrics-0.9.3-py3-none-any.whl (419 kB)
    419.6/419.6 kB 4.9 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.17.2 in /usr/local/lib/python3.10/dist-packages (from torchmetrics==0.9.3) (1.23.5)
Requirement already satisfied: torch>=1.3.1 in /usr/local/lib/python3.10/dist-packages (from torchmetrics==0.9.3) (2.1.0+cu121)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from torchmetrics==0.9.3) (23.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch>=1.3.1->torchmetrics==0.9.3) (3.13.1)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from torch>=1.3.1->torchmetrics==0.9.3) (4.5.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch>=1.3.1->torchmetrics==0.9.3) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch>=1.3.1->torchmetrics==0.9.3) (3.2.1)
Requirement already satisfied: Jinja2 in /usr/local/lib/python3.10/dist-packages (from torch>=1.3.1->torchmetrics==0.9.3) (3.1.2)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch>=1.3.1->torchmetrics==0.9.3) (2023.6.0)
Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/dist-packages (from torch>=1.3.1->torchmetrics==0.9.3) (2.1.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from Jinja2->torch>=1.3.1->torchmetrics==0.9.3) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-packages (from sympy->torch>=1.3.1->torchmetrics==0.9.3) (1.3.0)
Installing collected packages: torchmetrics
Successfully installed torchmetrics-0.9.3
tensor(0.9950)

```

Setelah memasang `torchmetrics` dan menghitung akurasi dengan menggunakan objek `Accuracy`, kita dapat melihat bahwa nilai akurasi yang dihasilkan oleh `torchmetrics_accuracy` sangat mendekati nilai akurasi yang dihitung sebelumnya menggunakan `accuracy_fn`. Hal ini menunjukkan konsistensi antara implementasi akurasi dari `torchmetrics` dan implementasi manual yang digunakan sebelumnya. Outputnya adalah tensor dengan nilai akurasi sebesar 0.9950, atau 99.50%

Chapter 03

```

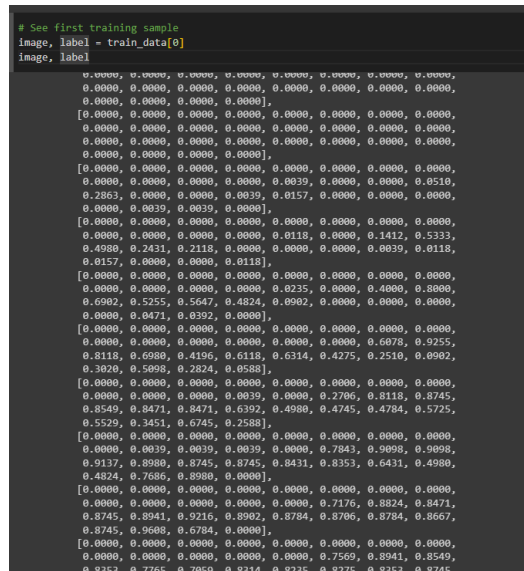
# Setup training data
train_data = datasets.FashionMNIST(
    root="data", # where to download data to?
    train=True, # get training data
    download=True, # download data if it doesn't exist on disk
    transform=ToTensor(), # images come as PIL format, we want to turn into torch tensors
    target_transform=None # you can transform labels as well
)

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

```

Kode di atas menginisialisasi data pelatihan dan pengujian untuk dataset FashionMNIST menggunakan pustaka PyTorch. Untuk data pelatihan, fungsi `datasets.FashionMNIST` digunakan dengan konfigurasi yang menentukan direktori penyimpanan data, pengunduhan data jika belum ada di disk, dan transformasi gambar menjadi tensor Torch menggunakan `ToTensor()`. Data pelatihan diakses dengan parameter `train=True`. Data pengujian diatur dengan konfigurasi serupa, namun dengan parameter `train=False`. Objek `train_data` dan `test_data` memuat dataset FashionMNIST yang siap digunakan untuk pelatihan dan pengujian model.

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



menampilkan contoh pertama dari dataset pelatihan FashionMNIST. Gambar pertama tersebut direpresentasikan dalam bentuk tensor 3 dimensi, yang mewakili intensitas piksel pada setiap saluran warna. Label kelas untuk gambar tersebut adalah 9

```
# What's the shape of the image?
image.shape

torch.Size([1, 28, 28])

# How many samples are there?
len(train_data.data), len(train_data.targets), len(test_data.data), len(test_data.targets)

(60000, 60000, 10000, 10000)

# See classes
class_names = train_data.classes
class_names

['T-shirt/top',
 'Trouser',
 'Pullover',
 'Dress',
 'Coat',
 'Sandal',
 'Shirt',
 'Sneaker',
 'Bag',
 'Tote bag']
```

image.shape digunakan untuk mengetahui bentuk tensor dari gambar pertama dalam dataset pelatihan FashionMNIST, len(train_data.data) memberikan jumlah sampel dalam dataset pelatihan, len(train_data.targets) memberikan jumlah label atau target dalam dataset pelatihan. Begitu pula untuk dataset pengujian, len(test_data.data) dan len(test_data.targets) memberikan informasi serupa untuk sampel dan label dalam dataset pengujian, class_names = train_data.classes digunakan untuk mendapatkan daftar nama kelas atau kategori dalam dataset FashionMNIST.



Kode di atas digunakan untuk membuat tampilan visual matriks 4x4 gambar acak beserta label kelasnya dari dataset pelatihan FashionMNIST. Dengan menggunakan kode tersebut, kita dapat melihat representasi matriks 4x4 yang memperlihatkan gambar-gambar acak dan label kelas terkait dari dataset FashionMNIST. Ini memberikan gambaran visual ringkas tentang variasi kelas dalam dataset pelatihan FashionMNIST

```

# 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)

```



```

# 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 (accumulatively)
        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

```

Kode di atas adalah implementasi pelatihan dan pengujian model menggunakan PyTorch pada dataset FashionMNIST. Dalam setiap epoch, dilakukan loop melalui batch pelatihan, dengan langkah-langkah mencakup forward pass, perhitungan loss, pengaturan gradien ke nol, backward pass, dan langkah optimizer. Rata-rata loss pelatihan dihitung per epoch. Selanjutnya, dilakukan loop pengujian untuk menghitung rata-rata loss dan akurasi pengujian. Hasil metrics seperti loss pelatihan, loss pengujian, dan akurasi pengujian dicetak setiap epoch. Waktu total pelatihan juga dihitung. Ada potensi perbaikan yang dapat dilakukan, seperti menggunakan `torch.no_grad()` selama pengujian dan penyesuaian lainnya sesuai kebutuhan model.

```

from torchmetrics import ConfusionMatrix
from mlxtend.plotting import plot_confusion_matrix

# 2. Setup confusion matrix instance and compare predictions to targets
confmat = ConfusionMatrix(num_classes=len(class_names), task='multiclass')
confmat_tensor = confmat(preds=y_pred_tensor,
                        target=test_data.targets)

# 3. Plot the confusion matrix
fig, ax = plot_confusion_matrix(
    conf_mat=confmat_tensor.numpy(), # matplotlib likes working with NumPy
    class_names=class_names, # turn the row and column labels into class names
    figsize=(10, 7)
);

```

Kode di atas berfungsi untuk menciptakan dan menampilkan matriks kebingungan (confusion matrix) menggunakan modul `torchmetrics` dan `mlxtend`. Melalui pendekatan ini, program menghasilkan serta memvisualisasikan matriks kebingungan yang memberikan pemahaman tentang kinerja model pada setiap kelas. Matriks kebingungan tersebut membandingkan prediksi model dengan label sebenarnya pada dataset pengujian, memberikan wawasan yang berguna tentang sejauh mana model dapat mengklasifikasikan data pada setiap kategori.

```

# Evaluate loaded model
torch.manual_seed(42)

loaded_model_2_results = eval_model(
    model=loaded_model_2,
    data_loader=test_dataloader,
    loss_fn=loss_fn,
    accuracy_fn=accuracy_fn
)

loaded_model_2_results

```

Pemanggilan fungsi `eval_model` dilakukan untuk mengevaluasi model yang telah dimuat, yaitu `loaded_model_2`. Meskipun, detail implementasi fungsi `eval_model` tidak tersedia dalam pertanyaan ini. Hasil evaluasi model pada dataset pengujian memberikan informasi mengenai performa model "FashionMNISTModelV2" melalui berbagai metrik performa seperti loss dan akurasi. Performa yang baik dicirikan oleh nilai loss yang rendah dan akurasi yang tinggi pada dataset pengujian. Dengan demikian, hasil evaluasi ini memberikan gambaran sejauh mana model mampu mengklasifikasikan data pada FashionMNIST, diukur dari aspek ketepatan dan keakuratan prediksinya

```
# Check to see if results are close to each other (if they are very far away, there may be an error)
torch.isclose(torch.tensor(model_2_results["model_loss"]),
               torch.tensor(loaded_model_2_results["model_loss"]),
               atol=1e-08, # absolute tolerance
               rtol=0.0001) # relative tolerance
tensor(True)
```

`tensor(True)` menunjukkan bahwa nilai loss dari kedua model, yaitu model yang sedang dievaluasi (`model_2_results`) dan model yang telah dimuat (`loaded_model_2_results`), mendekati satu sama lain. Artinya, perbedaan antara kedua nilai loss tersebut berada dalam batas toleransi yang telah ditentukan (`atol=1e-08` dan `rtol=0.0001`). Sehingga, dari segi loss, kedua model dianggap memberikan hasil yang serupa atau mendekati satu sama lain.

Chapter 04

```
import requests
import zipfile
from pathlib import Path

# Setup path to data folder
data_path = Path("data/")
image_path = data_path / "pizza_steak_sushi"

# If the image folder doesn't exist, download it and prepare it...
if image_path.is_dir():
    print(f"{image_path} directory exists.")
else:
    print(f"Did not find {image_path} directory, creating one...")
    image_path.mkdir(parents=True, exist_ok=True)

# Download pizza, steak, sushi data
with open(data_path / "pizza_steak_sushi.zip", "wb") as f:
    request = requests.get("https://github.com/mrdbourke/pytorch-deep-learning/raw/main/data/pizza_steak_sushi.zip")
    print("Downloading pizza, steak, sushi data...")
    f.write(request.content)

# Unzip pizza, steak, sushi data
with zipfile.ZipFile(data_path / "pizza_steak_sushi.zip", "r") as zip_ref:
    print("Unzipping pizza, steak, sushi data...")
    zip_ref.extractall(image_path)
```

Kode di atas dirancang untuk mempersiapkan dataset yang berisi gambar-gambar pizza, steak, dan sushi. Dua path ditetapkan, yaitu satu untuk folder data secara keseluruhan (`data_path`) dan satu lagi untuk folder khusus yang akan berisi gambar-gambar tersebut (`image_path`). Pemeriksaan dilakukan untuk memastikan apakah direktori `image_path` sudah ada. Jika belum, program membuat direktori tersebut dan mengunduh dataset gambar pizza, steak, dan sushi dari GitHub menggunakan modul `requests`. Setelah mengunduh file zip, kode melakukan ekstraksi dan menyimpan gambar-gambar tersebut di dalam folder `image_path`.

```
walk_through_dir(image_path)

There are 2 directories and 1 images in 'data/pizza_steak_sushi'.
There are 3 directories and 0 images in 'data/pizza_steak_sushi/test'.
There are 0 directories and 19 images in 'data/pizza_steak_sushi/test/steak'.
There are 0 directories and 31 images in 'data/pizza_steak_sushi/test/sushi'.
There are 0 directories and 25 images in 'data/pizza_steak_sushi/test/pizza'.
There are 3 directories and 0 images in 'data/pizza_steak_sushi/train'.
There are 0 directories and 75 images in 'data/pizza_steak_sushi/train/steak'.
There are 0 directories and 72 images in 'data/pizza_steak_sushi/train/sushi'.
There are 0 directories and 78 images in 'data/pizza_steak_sushi/train/pizza'.
```

Output tersebut memberikan gambaran tentang struktur dataset yang telah dipersiapkan. Dataset ini terbagi menjadi dua bagian utama, yaitu bagian pelatihan (`train`) dan pengujian (`test`)


```

import random
from PIL import Image

# Set seed
random.seed(42) # <- try changing this and see what happens

# 1. Get all image paths (* means "any combination")
image_path_list = list(image_path.glob("**/*.jpg"))

# 2. Get random image path
random_image_path = random.choice(image_path_list)

# 3. Get image class from path name (the image class is the name of the directory where the image is stored)
image_class = random_image_path.parent.stem

# 4. Open image
img = Image.open(random_image_path)

# 5. Print metadata
print(f"Random image path: {random_image_path}")
print(f"Image class: {image_class}")
print(f"Image height: {img.height}")
print(f"Image width: {img.width}")

```

Kode di atas berfungsi untuk memilih secara acak satu gambar dari dataset yang telah disiapkan. Pertama, kode mengumpulkan semua path file gambar dalam bentuk list menggunakan modul pathlib. Selanjutnya, ia memilih satu path gambar secara acak dengan menggunakan fungsi random.choice. Dari path gambar yang dipilih, kode mendapatkan kelas gambar (image class) dari nama direktori tempat gambar tersebut disimpan. Setelah itu, gambar dibuka menggunakan modul PIL (Python Imaging Library) dan informasi metadata dari gambar tersebut, seperti tinggi (height) dan lebar (width), ditampilkan

```

# Set random seeds
torch.manual_seed(42)
torch.cuda.manual_seed(42)

# Set number of epochs
NUM_EPOCHS = 5

# Recreate an instance of TinyVGG
model_0 = TinyVGG(input_shape=3, # number of color channels (3 for RGB)
                  hidden_units=10,
                  output_shape=len(train_data.classes)).to(device)

# Setup loss function and optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(params=model_0.parameters(), lr=0.001)

# Start the timer
from timeit import default_timer as timer
start_time = timer()

# Train model_0
model_0_results = train(model=model_0,
                        train_dataloader=train_dataloader_simple,
                        test_dataloader=test_dataloader_simple,
                        optimizer=optimizer,
                        loss_fn=loss_fn,
                        epochs=NUM_EPOCHS)

# End the timer and print out how long it took
end_time = timer()
print(f"Total training time: {end_time-start_time:.3f} seconds")

```

Kode di atas digunakan untuk melatih model TinyVGG pada dataset yang telah disiapkan. Pertama, ditentukan random seed untuk pengulangan yang reproduksibel menggunakan fungsi torch.manual_seed dan torch.cuda.manual_seed. Kemudian, diinisialisasi model TinyVGG dengan parameter yang sesuai, seperti jumlah saluran warna input, unit tersembunyi, dan bentuk output. Selanjutnya, ditetapkan fungsi loss (nn.CrossEntropyLoss) dan optimizer (torch.optim.Adam). Proses pelatihan dimulai dengan memanggil fungsi train, yang mengambil model, dataloader pelatihan dan pengujian, optimizer, loss function, serta jumlah epoch sebagai parameter


```
# Plot custom image
plt.imshow(custom_image.permute(1, 2, 0)) # need to permute image dimensions from CHW -> HWC otherwise matplotlib will error
plt.title(f"Image shape: {custom_image.shape}")
plt.axis(False);
```

Kode di atas menggunakan Matplotlib untuk memplot gambar khusus yang telah dimuat sebelumnya. Sebelum melakukan plotting, perlu dilakukan permutasi pada dimensi gambar dari CHW (Channel, Height, Width) menjadi HWC (Height, Width, Channel). Hal ini diperlukan karena Matplotlib mengharapkan urutan dimensi yang berbeda. Selanjutnya, gambar tersebut diplot menggunakan `plt.imshow`, dan judul plot menunjukkan bentuk (shape) dari tensor gambar tersebut. Parameter `plt.axis(False)` digunakan untuk menyembunyikan sumbu pada plot.

```
# Print out prediction logits
print(f"Prediction logits: {custom_image_pred}")

# Convert logits -> prediction probabilities (using torch.softmax() for multi-class classification)
custom_image_pred_probs = torch.softmax(custom_image_pred, dim=1)
print(f"Prediction probabilities: {custom_image_pred_probs}")

# Convert prediction probabilities -> prediction labels
custom_image_pred_label = torch.argmax(custom_image_pred_probs, dim=1)
print(f"Prediction label: {custom_image_pred_label}")

Prediction logits: tensor([[ 0.1172,  0.0160, -0.1425]], device='cuda:0')
Prediction probabilities: tensor([[0.3738, 0.3378, 0.2883]], device='cuda:0')
Prediction label: tensor([0], device='cuda:0')
```

Prediction logits: Ini adalah nilai logit yang dihasilkan oleh model untuk setiap kelas. Dalam hal ini, terdapat tiga nilai logit karena model memiliki tiga kelas output. **Prediction probabilities:** Setelah menerapkan fungsi softmax, kita mendapatkan distribusi probabilitas untuk setiap kelas. Probabilitas ini menjelaskan seberapa yakin model terhadap prediksi kelas tertentu. Dalam contoh ini, probabilitas kelas pertama adalah 0.3738, kelas kedua adalah 0.3378, dan kelas ketiga adalah 0.2883. **Prediction label:** Ini adalah label prediksi yang diambil berdasarkan kelas dengan probabilitas tertinggi. Dalam contoh ini, kelas dengan probabilitas tertinggi adalah kelas pertama, dan label prediksi adalah 0.