

TECHNICAL REPORT UAS MACHINE LEARNING
Pytorch For Deep Learning



Oleh :

Nadhifi Qurrunul Bahratu Fauzan Hibatullah /1103204156

**PRODI S1 TEKNIK KOMPUTER
FAKULTAS TEKNIK ELEKTRO
UNIVERSITAS TELKOM
BANDUNG
2022**

Pytorch For Deep Learning

PyTorch for Deep Learning adalah sebuah modul Python yang menyediakan berbagai fungsi dan kelas untuk membantu pengembang membangun model pembelajaran mendalam (deep learning). Modul ini mencakup berbagai fungsi untuk memanipulasi data, membangun jaringan saraf, dan melatih model.

PyTorch for Deep Learning menawarkan beberapa keunggulan dibandingkan dengan framework pembelajaran mendalam lainnya, antara lain:

- Performa yang cepat: PyTorch dapat memanfaatkan kekuatan GPU untuk mempercepat pelatihan dan inferensi model pembelajaran mendalam.
- Fleksibilitas: PyTorch memungkinkan pengembang untuk mengontrol secara penuh proses pelatihan dan inferensi model.
- Komunitas yang aktif: PyTorch memiliki komunitas yang aktif yang menyediakan berbagai sumber daya dan dukungan untuk pengembang.

PyTorch for Deep Learning dapat digunakan untuk berbagai aplikasi pembelajaran mendalam, seperti:

- Pemrosesan bahasa alami (natural language processing)
- Pembelajaran mesin (machine learning)
- Komputer visi (computer vision)

Berikut adalah beberapa contoh penggunaan PyTorch for Deep Learning:

- Google Translate menggunakan PyTorch untuk menerjemahkan teks dari satu bahasa ke bahasa lain.
- Facebook menggunakan PyTorch untuk mengembangkan model pengenalan wajah.
- OpenAI menggunakan PyTorch untuk mengembangkan model bermain game.

Untuk memulai menggunakan PyTorch for Deep Learning, pengembang perlu memiliki pemahaman dasar tentang pembelajaran mendalam dan Python. Pengembang juga dapat mengikuti tutorial PyTorch untuk mempelajari cara menggunakan modul ini.

00 Pytorch Fundamental

1. Introduction To Tensors

```
[78] # Import library PyTorch
import torch

# Cetak versi PyTorch yang sedang digunakan
torch.__version__

'2.1.0+cu121'
```

```
[79] # Membuat tensor skalar dengan nilai 7
scalar = torch.tensor(7)
scalar

tensor(7)
```

```
[80] # Memeriksa jumlah dimensi dari tensor skalar (seharusnya 0 untuk skalar)
scalar.ndim

0
```

```
[81] # Mengekstrak angka Python dari tensor skalar (hanya berfungsi untuk tensor satu elemen)
scalar.item()

7
```

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

tensor([7, 7])
```

```

08 [83] # Membuat tensor vektor dengan dua elemen (keduanya bernilai 7)
    vector = torch.tensor([7, 7])
    vector

    tensor([7, 7])

08 [84] # Memeriksa bentuk dari tensor vektor (ukuran 2)
    vector.shape

    torch.Size([2])

08 [85] # Membuat tensor matriks (2x2) dengan nilai yang ditentukan
    MATRIX = torch.tensor([[7, 8],
                           [9, 10]])
    MATRIX

    tensor([[ 7,  8],
            [ 9, 10]])

08 [86] # Memeriksa jumlah dimensi dari tensor matriks (seharusnya 2 untuk matriks)
    MATRIX.ndim

    2

08 [87] # Memeriksa bentuk dari tensor matriks (ukuran 2x2)
    MATRIX.shape

    torch.Size([2, 2])

08 [88] # Membuat tensor 3D dengan nilai yang ditentukan
    TENSOR = torch.tensor([[[1, 2, 3],
                           [3, 6, 9],
                           [2, 4, 5]]])
    TENSOR

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

08 [89] # Memeriksa jumlah dimensi dari tensor 3D (seharusnya 3)
    TENSOR.ndim

    3

08 [90] # Memeriksa bentuk dari tensor 3D (ukuran 1x3x3)
    TENSOR.shape

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

```

Analisis:

- Kode tersebut berfokus pada penggunaan PyTorch untuk membuat tensor dengan berbagai dimensi (skalar, vektor, matriks, dan tensor 3D).
- Fungsi `torch.tensor()` digunakan untuk membuat tensor dengan nilai yang telah ditentukan.
- `ndim` digunakan untuk mengecek jumlah dimensi dari tensor.
- `shape` digunakan untuk mengecek ukuran atau bentuk dari tensor.

- Pada bagian komentar, setiap baris dijelaskan sesuai dengan tugas yang dilakukan oleh kode.

2. Random tensors

Pengguna dapat memilih fungsi yang sesuai dengan kebutuhannya.

```
[91] # Membuat tensor acak dengan ukuran (3, 4)
      random_tensor = torch.rand(size=(3, 4))
      random_tensor, random_tensor.dtype

(tensor([[0.8694, 0.5677, 0.7411, 0.4294],
         [0.8854, 0.5739, 0.2666, 0.6274],
         [0.2696, 0.4414, 0.2969, 0.8317]]),
      torch.float32)
```

- Pada , sebuah tensor acak dengan ukuran (3, 4) dibuat menggunakan torch.rand(size=(3, 4)).
- random_tensor dan tipe data tensor (random_tensor.dtype) dicetak untuk menampilkan hasilnya.

```
# Membuat tensor acak dengan ukuran (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)
```

- Pada , tensor acak dengan ukuran (224, 224, 3) dibuat.
- random_image_size_tensor.shape digunakan untuk mengetahui bentuk (shape) dari tensor, dan random_image_size_tensor.ndim digunakan untuk mengetahui jumlah dimensi dari tensor tersebut.

Analisis Umum:

- Kode menggunakan fungsi `torch.rand()` untuk membuat tensor dengan nilai acak antara 0 dan 1.
- Pada bagian kedua, ukuran tensor (224, 224, 3) mengindikasikan bahwa tensor tersebut mungkin digunakan untuk merepresentasikan suatu citra dengan dimensi tinggi (224x224 piksel) dan 3 saluran warna (RGB).
- Pemahaman mengenai bentuk dan dimensi tensor penting untuk memastikan kesesuaian dalam pemrosesan lebih lanjut, terutama dalam konteks pembelajaran mesin atau pengolahan citra.

3. Zeros and ones

```
[93] # Membuat tensor dengan nilai semua elemen nol dan ukuran (3, 4)
zeros = torch.zeros(size=(3, 4))
zeros, zeros.dtype
```

```
(tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]),
torch.float32)
```

- `torch.zeros(size=(3, 4))` digunakan untuk membuat tensor dengan semua elemen bernilai nol dan ukuran (shape) (3, 4).
- Tensor hasil (`zeros`) dan tipe datanya (`zeros.dtype`) dicetak.

```
[94] # Membuat tensor dengan nilai semua elemen satu dan ukuran (3, 4)
ones = torch.ones(size=(3, 4))
ones, ones.dtype
```

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

- `torch.ones(size=(3, 4))` digunakan untuk membuat tensor dengan semua elemen bernilai satu dan ukuran (shape) (3, 4).
- Tensor hasil (`ones`) dan tipe datanya (`ones.dtype`) dicetak.

Analisis Umum:

- Kode tersebut menggunakan fungsi `'torch.zeros'` dan `'torch.ones'` untuk membuat tensor dengan nilai nol atau satu.
- Penggunaan tensor dengan nilai awal tertentu seringkali penting dalam inisialisasi model atau operasi matematika tertentu.
- Tipe data tensor (`'dtype'`) mencerminkan tipe data elemen di dalam tensor (misalnya, `float32`).

4. Creating a range and tensors like

```
[95] # Menggunakan torch.range(), yang sudah dinyatakan usang (deprecated)
zero_to_ten_deprecated = torch.range(0, 10) # Catatan: ini mungkin menghasilkan kesalahan di masa depan

# Membuat rentang nilai dari 0 hingga 10 dengan langkah 1
zero_to_ten = torch.arange(start=0, end=10, step=1)
zero_to_ten
```

```
<ipython-input-95-a404776195c1>:2: UserWarning: torch.range is deprecated and will be removed in a future release because its behavior is
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])
```

- Menggunakan `torch.range()` untuk membuat tensor dengan nilai dari 0 hingga 10.
- Perlu dicatat bahwa `torch.range()` sudah usang (deprecated) dan mungkin menyebabkan kesalahan di masa depan.
- Menggunakan `torch.arange()` untuk membuat tensor dengan nilai dari 0 hingga 10 dengan langkah 1.

```
[96] # Dapat juga membuat tensor berisi nol dengan bentuk yang sama seperti tensor lain
ten_zeros = torch.zeros_like(input=zero_to_ten) # Akan memiliki bentuk yang sama
ten_zeros
```

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

- Menggunakan `torch.zeros_like()` untuk membuat tensor berisi nol dengan bentuk yang sama seperti `zero_to_ten`.

Analisis:

- Penggunaan fungsi yang sudah usang (`torch.range()`) diberi peringatan tentang kemungkinan penghapusan di masa depan.
- Penggunaan `torch.arange()` dan `torch.zeros_like()` adalah pendekatan yang lebih disarankan dan umum digunakan dalam pembuatan tensor dengan PyTorch.

5. Tensors Data Type

```
[ ] # 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

# Menampilkan bentuk, tipe data, dan perangkat (device) dari tensor float_32_tensor
float_32_tensor.shape, float_32_tensor.dtype, float_32_tensor.device

(torch.Size([3]), torch.float32, device(type='cpu'))
```

- Membuat tensor `float_32_tensor` dengan tipe data default `float32`, bentuk tensor diambil dari data yang diberikan (3 elemen), perangkat (device) default, dan tanpa gradient (`requires_grad=False`).
- Menampilkan bentuk, tipe data, dan perangkat (device) dari tensor `float_32_tensor`.

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

# Menampilkan tipe data dari tensor float_16_tensor
float_16_tensor.dtype

torch.float16
```


Kode tersebut berfokus pada pembuatan tensor dengan tipe data tertentu menggunakan PyTorch. Berikut adalah analisisnya:

Membuat tensor `float_32_tensor` dengan tipe data default `float32`. Penggunaan `dtype=None` membuat tensor menggunakan tipe data default PyTorch, yang dalam hal ini adalah `float32`. `requires_grad=False` menandakan bahwa tensor tidak perlu merekam operasi untuk gradient computation. Menampilkan informasi tentang tensor `float_32_tensor`, termasuk bentuk (shape), tipe data, dan perangkat (device). Membuat tensor `float_16_tensor` dengan tipe data `float16` menggunakan `dtype=torch.float16`. `torch.half` juga dapat digunakan sebagai alternatif untuk menyatakan tipe data `float16`. Menampilkan tipe data dari tensor `float_16_tensor`.

- Default datatype untuk tensor dalam PyTorch adalah `float32`.
- Pemilihan tipe data tensor dapat dilakukan dengan parameter `dtype` saat pembuatan tensor.
- Informasi tentang tensor, seperti bentuk, tipe data, dan perangkat, dapat diakses untuk pemahaman lebih lanjut tentang propertinya.

6. Getting information from tensors

Selain atribut dan metode tensor, Anda juga dapat menggunakan operator untuk mendapatkan informasi dari tensor. Misalnya, Anda dapat menggunakan operator `len()` untuk mendapatkan jumlah elemen dalam tensor.

```
0s  # Membuat tensor acak dengan ukuran (3, 4)
some_tensor = torch.rand(3, 4)

# Menampilkan detail tentang tensor
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}")

# Menampilkan isi tensor
# Menampilkan bentuk (shape) tensor
# Menampilkan tipe data tensor
# Menampilkan perangkat (device) tempat tensor disimpan (akan mengg
```

```
tensor([[0.4416, 0.5443, 0.9440, 0.1085],
        [0.7738, 0.8842, 0.5447, 0.5837],
        [0.6097, 0.1575, 0.2596, 0.9772]])
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

- Membuat tensor `some_tensor` dengan menggunakan `torch.rand` untuk mengisi tensor dengan nilai acak dan memberikan ukuran (shape) (3, 4).
- Menampilkan berbagai detail tentang tensor seperti nilai-nilai tensor, bentuk (shape), tipe data, dan perangkat (device) tempat tensor disimpan.

Kode tersebut dirancang untuk membuat tensor acak dengan PyTorch dan menampilkan beberapa detail tentang tensor tersebut. Berikut adalah analisisnya:

Kode ini menggunakan `torch.rand` untuk membuat tensor (`some_tensor`) dengan ukuran (shape) (3, 4) dan mengisinya dengan nilai acak antara 0 dan 1. Kode ini memanfaatkan fungsi PyTorch untuk menciptakan dan mengelola tensor. Informasi yang ditampilkan sangat penting dalam pemahaman dan pemrosesan tensor, seperti bentuk, tipe data, dan perangkat penyimpanan. Tensor yang dibuat diisi dengan nilai acak, sesuai dengan fungsi `torch.rand()`.

7. Manipulating tensors (tensor operations)

```
[23] # Creating a tensor of values and adding a number to it
      tensor = torch.tensor([1, 2, 3])
      tensor + 10
      # Output: tensor([11, 12, 13])
```

```
tensor([11, 12, 13])
```

```
[24] # Multiplying the tensor by 10
      tensor
```

```
tensor([10, 20, 30])
```

```
[25] # Subtracting and reassigning
      tensor = tensor - 10
      tensor
```

```
tensor([1, 2, 3])
```

```
✓ [26] # Subtract and reassign
0s tensor = tensor - 10
      tensor
```

```
tensor([-9, -8, -7])
```

```
[28] # Using torch.multiply() function
      torch.multiply(tensor, 10)
```

```
tensor([10, 20, 30])
```

```
✓ [29] # Original tensor is still unchanged
0s tensor
```

```
tensor([1, 2, 3])
```

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

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

Analisis:

Pada 3 baris code paling atas, operasi dasar dilakukan pada tensor, seperti penambahan dan perkalian dengan skalar. pada baris ke 4 dan 5 menunjukkan bahwa nilai tensor tidak berubah kecuali variabel tensor di-reassign. Ini menunjukkan sifat immutable dari tensor. pada baris 6 dan 7 menggunakan fungsi bawaan PyTorch seperti `torch.multiply()`. Original tensor tetap tidak berubah. pada baris terakhir menunjukkan penggunaan operator simbol (*), yang lebih umum daripada menggunakan fungsi seperti `torch.mul()`. Operasi perkalian dilakukan secara element-wise (setiap elemen dikalikan dengan elemen yang setara), dan hasilnya adalah tensor baru. Analisis umum menunjukkan kemampuan PyTorch dalam melakukan operasi dasar dan memanipulasi tensor dengan cara yang intuitif. Perhatikan bahwa operasi ini bersifat element-wise jika tidak dinyatakan sebaliknya.

8. Matrix Multiplication

```
[31] import torch
      # Membuat tensor dengan nilai [1, 2, 3]
      tensor = torch.tensor([1, 2, 3])
      # Menampilkan bentuk (shape) tensor
      tensor.shape
```

```
torch.Size([3])
```

```
[32] # Operasi perkalian elemen-wise pada tensor
      tensor * tensor
```

```
tensor([1, 4, 9])
```

```
[33] # Operasi perkalian matriks (dot product)
      torch.matmul(tensor, tensor)
```

```
tensor(14)
```

```
[34] # Alternatif penggunaan simbol "@" untuk perkalian matriks (not recommended)
      tensor @ tensor
```

```
tensor(14)
```

```
[35] %%time
      # Pengukuran waktu untuk operasi perkalian matriks dengan perulangan
      # (disarankan untuk menghindari penggunaan perulangan for dalam operasi tensor karena komputasionalnya mahal)
      value = 0
      for i in range(len(tensor)):
          value += tensor[i] * tensor[i]
      value
```

```
CPU times: user 1.67 ms, sys: 0 ns, total: 1.67 ms
Wall time: 1.67 ms
tensor(14)
```

```
%%time
# Pengukuran waktu untuk operasi perkalian matriks dengan torch.matmul
torch.matmul(tensor, tensor)
```

```
CPU times: user 33 µs, sys: 7 µs, total: 40 µs
Wall time: 43.2 µs
tensor(14)
```

Kode tersebut fokus pada demonstrasi operasi matriks menggunakan PyTorch dan pengukuran waktu untuk membandingkan efisiensi operasi menggunakan metode yang berbeda. Berikut adalah analisisnya:

1. Pembuatan Tensor dan Menampilkan Bentuk ([1] dan [2]):
 - Tensor `tensor` dibuat dengan nilai [1, 2, 3].
 - `tensor.shape` digunakan untuk menampilkan bentuk (shape) tensor, yang dalam hal ini adalah `[3]`.
2. Operasi Perkalian Elemen-wise dan Matriks ([3] hingga [6]):
 - `tensor * tensor` melakukan perkalian elemen-wise pada tensor dengan dirinya sendiri.

- `torch.matmul(tensor, tensor)` melakukan operasi perkalian matriks (dot product).
- Penggunaan simbol "@" (`tensor @ tensor`) juga digunakan untuk perkalian matriks, meskipun tidak disarankan.

3. Pengukuran Waktu untuk Operasi Perkalian Matriks ([7] dan [9]):

- `%%time` digunakan untuk mengukur waktu eksekusi sel untuk operasi perkalian matriks dengan perulangan for.
- `value = 0` dan perulangan for digunakan untuk melakukan perkalian matriks secara manual.
- `torch.matmul(tensor, tensor)` diukur waktu eksekusinya menggunakan `%%time`. Hasilnya menunjukkan bahwa operasi dengan `torch.matmul` lebih efisien dan lebih cepat.

Catatan dan Kesimpulan:

- Operasi matriks menggunakan fungsi PyTorch seperti `torch.matmul` atau operator `@` lebih disarankan daripada menggunakan perulangan for karena lebih efisien dan cepat.
- Pengukuran waktu eksekusi memberikan pemahaman visual tentang perbandingan kinerja antara dua pendekatan tersebut.
- Dalam praktiknya, menggunakan fungsi PyTorch bawaan akan meningkatkan kinerja dan kejelasan kode.

9. Finding min max etc

```
[43] # Membuat tensor menggunakan torch.arange() dengan nilai awal 0, akhir 100 (tidak termasuk), dan langkah 10
x = torch.arange(0, 100, 10)
x

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

```
[44] # Menampilkan nilai maksimum dari tensor x
print(f"Maximum: {x.max()}")

# Menampilkan mean dari tensor x (perlu diubah tipe datanya menjadi float32)
# print(f"Mean: {x.mean()}") # Baris ini akan menghasilkan error
print(f"Mean: {x.type(torch.float32).mean()}") # Harus mengubah tipe data menjadi float32

# Menampilkan jumlah (sum) dari semua elemen tensor x
print(f"Sum: {x.sum()}")
```

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

```
▶ # Penggunaan fungsi max, min, mean, dan sum dengan torch
torch.max(x), torch.min(x), torch.mean(x.type(torch.float32)), torch.sum(x)

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

- PyTorch menyediakan operasi statistik seperti min, max, mean, dan sum yang dapat diterapkan langsung pada tensor.
- Saat menggunakan mean, perlu diingat bahwa tensor yang berisi bilangan bulat perlu diubah tipe datanya menjadi float32 terlebih dahulu, karena mean tidak dapat dihitung pada tensor integer.

- Fungsi-fungsi PyTorch seperti `torch.min()`, `torch.max()`, `torch.mean()`, dan `torch.sum()` memberikan hasil yang setara dengan metode operasi yang lebih langsung pada tensor.
- Menggunakan fungsi-fungsi PyTorch dapat membuat kode lebih jelas dan dapat dihindari kesalahan konversi tipe data.

✓ Positional min/max

```
# Membuat tensor dengan nilai dari 10 hingga 90 dengan langkah 10
tensor = torch.arange(10, 100, 10)
print(f"Tensor: {tensor}")

# Mendapatkan indeks dari nilai maksimum dan minimum dalam tensor
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

Analisis:

- Tensor yang dibuat adalah `tensor([10, 20, 30, 40, 50, 60, 70, 80, 90])`.
- Nilai maksimum dalam tensor adalah 90 dan terdapat di indeks ke-8.
- Nilai minimum dalam tensor adalah 10 dan terdapat di indeks ke-0.
- Fungsi `argmax()` dan `argmin()` bermanfaat untuk mendapatkan indeks dari nilai maksimum dan minimum dalam tensor, membantu identifikasi lokasi nilai ekstrim dalam data.
- Hasil cetak menunjukkan hasil yang diharapkan dari operasi ini.

✓ Change tensor datatype

```
[47] # Membuat tensor dan memeriksa tipe datanya
tensor = torch.arange(10., 100., 10.)
tensor.dtype
```

torch.float32

```
[48] # Membuat tensor dengan tipe data float16
tensor_float16 = tensor.type(torch.float16)
tensor_float16
```

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

```
# Membuat tensor dengan tipe data int8
tensor_int8 = tensor.type(torch.int8)
tensor_int8
```

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

Analisis:

- Tensor awal (`tensor`) dibuat dengan tipe data float32 secara default.
- Tipe data tensor dapat diperiksa menggunakan atribut `dtype`.
- `tensor.type(torch.float16)` menghasilkan tensor baru dengan tipe data float16 dari tensor awal.
- `tensor.type(torch.int8)` menghasilkan tensor baru dengan tipe data int8 dari tensor awal.
- Perubahan tipe data dapat mempengaruhi presisi dan ukuran memori yang digunakan oleh tensor.
- Penggunaan tipe data yang lebih rendah (seperti float16 atau int8) dapat mengurangi memori yang digunakan, tetapi perlu diperhatikan adanya potensi kehilangan presisi pada nilai tensor.

10. Reshape

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

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

```
▶ # Add an extra dimension
x_reshaped = x.reshape(1, 7)
x_reshaped, x_reshaped.shape

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

```
[52] # 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

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

```
[53] # Changing z changes x
z[:, 0] = 5
z, x
```

```
[54] # 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.]])
```

```
▶ # Menampilkan informasi tensor sebelumnya dan menghapus dimensi tambahan dari x_resaped
print(f"Previous tensor: {x_resaped}")
print(f"Previous shape: {x_resaped.shape}")
x_squeezed = x_resaped.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])
```

```
[56] # Menambah dimensi tambahan dengan unsqueeze
print(f"Previous tensor: {x_squeezed}")
print(f"Previous shape: {x_squeezed.shape}")
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])
```

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

```
[57] # Membuat tensor dengan bentuk tertentu, dan mengubah urutan sumbu
x_original = torch.rand(size=(224, 224, 3))
x_permuted = x_original.permute(2, 0, 1)

print(f"Previous shape: {x_original.shape}")
print(f"New shape: {x_permuted.shape}")
```

```
Previous shape: torch.Size([224, 224, 3])
New shape: torch.Size([3, 224, 224])
```

Analisis:

1. Membuat dan Mereshape Tensor ([1] hingga [4]):
 - Tensor awal `x` dibuat dengan nilai dari 1 hingga 7.
 - Dimensi tambahan ditambahkan dengan menggunakan `reshape` dan `view`.
 - Nilai pada tensor yang diubah (`z`) mempengaruhi tensor awal (`x`).
2. Stacking Tensor ([6]):**

Tensor `x` di-stack secara berulang untuk membentuk `x_stacked`.
3. Menghapus dan Menambah Dimensi ([8] hingga [12]):**
 - Dimensi tambahan dihapus dengan menggunakan `squeeze`.
 - Dimensi tambahan ditambahkan dengan menggunakan `unsqueeze`.

4. Manipulasi Tensor 3D ([14] hingga [18]):**

- Tensor `x_original` dibuat dengan bentuk (shape) 224x224x3.
- Dimensi diubah dengan menggunakan `permute` untuk mendapatkan tensor `x_permuted` dengan bentuk baru (shape) 3x224x224.

11. Indexing

```
[58] # Create a tensor
import torch
x = torch.arange(1, 10).reshape(1, 3, 3)
x, x.shape
```

```
(tensor([[[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]]),
 torch.Size([1, 3, 3]))
```

```
[59] # Let's index bracket by bracket
print(f"First square bracket:\n{x[0]}")
print(f"Second square bracket: {x[0][0]}")
print(f"Third square bracket: {x[0][0][0]}")
```

```
First square bracket:
tensor([[[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]])
Second square bracket: tensor([1, 2, 3])
Third square bracket: 1
```

```
[60] # Mendapatkan semua nilai dari dimensi ke-0 dan indeks ke-0 dari dimensi ke-1
x[:, 0]
```

```
tensor([[1, 2, 3]])
```

```
[61] # Mendapatkan semua nilai dari dimensi ke-0 dan indeks ke-1 dari dimensi ke-2
x[:, :, 1]
```

```
tensor([[2, 5, 8]])
```

```
▶ # Mendapatkan semua nilai dari dimensi ke-0 dan hanya nilai indeks ke-1 dari dimensi ke-1 dan ke-2
x[:, 1, 1]
```

```
🔗 tensor([5])
```

```
[63] # Mendapatkan indeks ke-0 dari dimensi ke-0 dan ke-1 serta semua nilai dari dimensi ke-2
x[0, 0, :] # sama dengan x[0][0]
```

```
tensor([1, 2, 3])
```

Analisis:

1. Pembuatan dan Pemeriksaan Tensor ([58]):

- `torch.arange(1, 10).reshape(1, 3, 3)` digunakan untuk membuat tensor 3D dengan nilai dari 1 hingga 9 dan direshape menjadi ukuran (1, 3, 3).
- `x` adalah tensor hasilnya dan `x.shape` digunakan untuk mendapatkan bentuk tensor tersebut.

2. Indeks dengan Bracket ([59] hingga [63]):

- Penggunaan bracket `[]` untuk melakukan indeksing tensor pada beberapa dimensi.
- Menunjukkan cara mendapatkan nilai pada setiap tingkat dimensi tensor.
- Contoh penggunaan indeksing:
 - `x[0]`: Mendapatkan seluruh matriks pada dimensi ke-0.
 - `x[0][0]`: Mendapatkan baris pertama dari matriks pada dimensi ke-0.
 - `x[0][0][0]`: Mendapatkan nilai pertama dari baris pertama pada matriks dimensi ke-0.
 - `x[:, 0]`: Mendapatkan semua nilai dari dimensi ke-0 dan indeks ke-0 dari dimensi ke-1.
 - `x[:, :, 1]`: Mendapatkan semua nilai dari dimensi ke-0 dan indeks ke-1 dari dimensi ke-2.
 - `x[:, 1, 1]`: Mendapatkan semua nilai dari dimensi ke-0 dan hanya nilai indeks ke-1 dari dimensi ke-1 dan ke-2.
 - `x[0, 0, :]`: Mendapatkan indeks ke-0 dari dimensi ke-0 dan ke-1 serta semua nilai dari dimensi ke-2. Sama dengan `x[0][0]`.

12. Pytorch tensors & Numpy

PyTorch tensors & NumPy

Cell 4/10

```
[64] # Membuat array NumPy dan mengonversinya menjadi tensor
import torch
import numpy as np
array = np.arange(1.0, 8.0) # Membuat array NumPy dari 1.0 hingga 7.0
tensor = torch.from_numpy(array) # Mengonversi array NumPy menjadi tensor
array, tensor
```

```
(array([1., 2., 3., 4., 5., 6., 7.]),
 tensor([1., 2., 3., 4., 5., 6., 7.], dtype=torch.float64))
```

```
[65] # Mengubah array NumPy, tetapi tetap menggunakan tensor yang sama
array = array + 1
array, tensor
```

```
(array([2., 3., 4., 5., 6., 7., 8.]),
 tensor([1., 2., 3., 4., 5., 6., 7.], dtype=torch.float64))
```

```
[66] # Mengonversi tensor menjadi array NumPy
tensor = torch.ones(7) # Membuat tensor yang berisi tujuh elemen dengan nilai satu dan tipe data float32
numpy_tensor = tensor.numpy() # Mengonversi tensor menjadi array NumPy
tensor, numpy_tensor
```

```
(tensor([1., 1., 1., 1., 1., 1., 1.]),
 array([1., 1., 1., 1., 1., 1., 1.], dtype=float32))
```

```
[67] # Mengubah tensor, tetapi tetap menggunakan array NumPy yang sama
tensor = tensor + 1
tensor, numpy_tensor
```

```
(tensor([2., 2., 2., 2., 2., 2., 2.]),
 array([1., 1., 1., 1., 1., 1., 1.], dtype=float32))
```

1. Membuat Array NumPy dan Mengonversinya Menjadi Tensor ([3] dan [4]):

- Dengan menggunakan `np.arange(1.0, 8.0)`, sebuah array NumPy dibuat dari 1.0 hingga 7.0.
- Melalui `torch.from_numpy(array)`, array NumPy diubah menjadi tensor PyTorch.

2. Mengubah Array NumPy Tetapi Tetap Menggunakan Tensor ([6] dan [7]):
 - Dengan ``array = array + 1``, setiap elemen array NumPy diubah dengan menambahkannya 1.
 - Meskipun array NumPy diubah, tensor PyTorch tetap terpengaruh karena keduanya berbagi memori.
3. Mengonversi Tensor Menjadi Array NumPy ([9] dan [10]):
 - ``torch.ones(7)`` digunakan untuk membuat tensor PyTorch yang berisi tujuh elemen dengan nilai satu dan tipe data float32.
 - Melalui ``tensor.numpy()``, tensor PyTorch diubah menjadi array NumPy.
4. Mengubah Tensor Tetapi Tetap Menggunakan Array NumPy yang Sama ([12] dan [13]):**
 - Dengan ``tensor = tensor + 1``, setiap elemen tensor PyTorch diubah dengan menambahkannya 1.
 - Meskipun tensor PyTorch diubah, array NumPy yang dibuat dari tensor tersebut tetap terpengaruh karena keduanya berbagi memori.

13. Reproducibility

✓ Reproducibility (trying to take the random out of random)

```
[68] import torch
# Membuat dua tensor acak
random_tensor_A = torch.rand(3, 4) # Membuat tensor acak A berukuran 3x4
random_tensor_B = torch.rand(3, 4) # Membuat tensor acak B berukuran 3x4

print(f"Tensor A:\n{random_tensor_A}\n") # Mencetak tensor A
print(f"Tensor B:\n{random_tensor_B}\n") # Mencetak tensor B
print(f"Does Tensor A equal Tensor B? (anywhere)")
random_tensor_A == random_tensor_B # Memeriksa apakah nilai elemen di Tensor A sama dengan 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]])
```

```
# Set the random seed
RANDOM_SEED=42 # Menetapkan seed acak untuk reproduktibilitas
torch.manual_seed(seed=RANDOM_SEED) # Menetapkan seed untuk generator angka acak PyTorch
random_tensor_C = torch.rand(3, 4) # Membuat tensor acak C berukuran 3x4

# Harus mereset seed setiap kali rand() baru dipanggil
# Tanpa ini, tensor_D akan berbeda dari tensor_C
torch.random.manual_seed(seed=RANDOM_SEED) # Menetapkan seed untuk generator angka acak PyTorch
random_tensor_D = torch.rand(3, 4) # Membuat tensor acak D berukuran 3x4

print(f"Tensor C:\n{random_tensor_C}\n") # Mencetak tensor C
print(f"Tensor D:\n{random_tensor_D}\n") # Mencetak tensor D
print(f"Does Tensor C equal Tensor D? (anywhere)")
random_tensor_C == random_tensor_D # Memeriksa apakah nilai elemen di Tensor C sama dengan 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]])
```

Pada potongan kode pertama, dua tensor acak, yaitu `random_tensor_A` dan `random_tensor_B`, dibuat menggunakan fungsi `torch.rand(3, 4)`. Kedua tensor tersebut kemudian dicetak, dan dilakukan perbandingan elemen antara keduanya menggunakan operator `==`. Hasilnya adalah sebuah tensor Boolean yang menunjukkan apakah nilai elemen di `random_tensor_A` sama dengan nilai elemen di `random_tensor_B`.

Pada potongan kode kedua, seed acak (`RANDOM_SEED=42`) diatur untuk memastikan reproduktibilitas hasil acak. Seed ini digunakan untuk menetapkan seed pada generator angka acak PyTorch menggunakan `torch.manual_seed` dan `torch.random.manual_seed`. Selanjutnya, dua tensor acak baru, yaitu `random_tensor_C` dan `random_tensor_D`, dibuat dengan menggunakan seed yang sama. Setelah itu, dilakukan perbandingan elemen antara kedua tensor tersebut. Hasilnya juga berupa tensor Boolean yang menunjukkan kecocokan elemen.

Analisis ini menunjukkan bahwa dengan menggunakan seed acak yang sama, tensor-tensor acak yang dihasilkan oleh PyTorch akan memiliki nilai elemen yang identik. Sebaliknya, jika seed berbeda, nilai elemen akan bervariasi antar tensor. Pengaturan seed acak adalah teknik penting dalam penelitian dan pengembangan yang melibatkan eksperimen dengan nilai acak, memastikan reproduktibilitas dan konsistensi dalam hasil.

14. Running Tensors on Gpu

1. Getting a GPU

Note:

```
0s [!nvidia-smi]

Thu Jan  4 14:13:18 2024

+-----+
| NVIDIA-SMI 535.104.05                 Driver Version: 535.104.05   CUDA Version: 12.2   |
+-----+-----+
| GPU  Name      Persistence-M   Bus-Id        Disp.A    Volatile Uncorr. ECC  |
| Fan  Temp  Perf    Pwr:Usage/Cap     Memory-Usage  GPU-Util  Compute M.  |
|                                           MIG M.       |
+-----+-----+
|   0   Tesla T4          Off          00000000:00:04:0  Off          0          |
| N/A   35C    P8             9W / 70W           0MiB / 15360MiB      0%      Default |
|                                           N/A          |
+-----+-----+

+-----+
| Processes:                               GPU Memory  |
|  GPU   GI    CI        PID   Type   Process name                  Usage      |
|  ID     ID     ID                                 |
+-----+-----+
| No running processes found               |
+-----+-----+
```

2. Getting PyTorch to run on the GPU

Note:

```
0s [71] # Check for GPU
import torch
torch.cuda.is_available()

True

0s [72] # Set device type
device = "cuda" if torch.cuda.is_available() else "cpu"
device

'cuda'

0s [73] # Count number of devices
torch.cuda.device_count()

1
```

3. Putting tensors (and models) on the GPU

Note:

```
[74] # Create tensor (default on CPU)
      tensor = torch.tensor([1, 2, 3])

      # Tensor not on GPU
      print(tensor, tensor.device)

      # Move tensor to GPU (if available)
      tensor_on_gpu = tensor.to(device)
      tensor_on_gpu

      tensor([1, 2, 3]) cpu
      tensor([1, 2, 3], device='cuda:0')
```

4. Moving tensors back to the CPU

Note:

```
✓ [75] import torch
0s

      # Assuming tensor_on_gpu is your tensor on GPU
      tensor_on_gpu = tensor_on_gpu.to('cpu') # Move tensor from GPU to CPU
      numpy_array = tensor_on_gpu.numpy()

      # Now you can use numpy_array as a NumPy array
      numpy_array = tensor_on_gpu.cpu().numpy()

✓ [76] # Instead, copy the tensor back to cpu
0s      tensor_back_on_cpu = tensor_on_gpu.cpu().numpy()
      tensor_back_on_cpu

      array([1, 2, 3])

✓ [77] tensor_on_gpu
0s      tensor([1, 2, 3])
```

Analisis:

Potongan kode di atas menunjukkan langkah-langkah untuk menggunakan GPU dengan PyTorch. Pada bagian pertama, [70], digunakan perintah `!nvidia-smi` untuk menampilkan informasi tentang GPU yang tersedia. Outputnya menunjukkan adanya GPU Tesla T4 yang dapat digunakan. Berikutnya, [71], kode Python menggunakan perpustakaan PyTorch untuk memeriksa ketersediaan GPU dengan `torch.cuda.is_available()`. Hasilnya adalah `True`, menandakan bahwa GPU dapat digunakan. Kemudian, [72], sebuah variabel `device` ditetapkan sebagai `"cuda"` jika

GPU tersedia, dan "cpu" jika tidak. Pada bagian [73], dilakukan pengecekan jumlah perangkat GPU dengan ``torch.cuda.device_count()``, dan outputnya menunjukkan satu perangkat GPU yang terdeteksi. Selanjutnya, [74], sebuah tensor PyTorch dibuat secara default di CPU. Kemudian, dengan menggunakan metode ``to(device)``, tensor tersebut dipindahkan ke GPU jika tersedia. Hasilnya adalah tensor yang sekarang berada di GPU, dan perangkatnya ditampilkan. Bagian [75] dan [76] menunjukkan cara memindahkan kembali tensor dari GPU ke CPU. Ini dapat dilakukan dengan menggunakan metode ``to('cpu')`` atau ``cpu()``. Juga, cara lainnya adalah dengan mengonversi tensor menjadi array NumPy dengan ``numpy()``. Pada akhirnya, [77], menunjukkan output dari tensor yang kini berada di GPU, tetapi dapat dipindahkan kembali ke CPU jika diperlukan.

Analisis ini menyajikan langkah-langkah yang diperlukan untuk memeriksa ketersediaan GPU, menentukan perangkat yang digunakan (GPU atau CPU), dan memindahkan tensor antara GPU dan CPU. Penggunaan GPU dalam komputasi tensor dapat meningkatkan kinerja, terutama untuk tugas-tugas yang melibatkan perhitungan berat seperti pelatihan model deep learning.

O1_pytorch_workflow_exercises

1. Pytorch Workflow Exercise Template

✓ 01. PyTorch Workflow Exercise Template

```
[ ] # Import library yang diperlukan
import torch # Import library PyTorch
import matplotlib.pyplot as plt # Import library matplotlib untuk visualisasi data
from torch import nn # Import modul neural network dari PyTorch

[ ] # Setup kode agar dapat berjalan pada perangkat apa pun (CPU atau GPU)
device = "cuda" if torch.cuda.is_available() else "cpu" # Tentukan perangkat yang akan digunakan (GPU jika tersedia, jika tidak, gunakan CPU)
device # Tampilkan perangkat yang dipilih (cuda untuk GPU, cpu untuk CPU)

'cuda'
```

✓ 1. Create a straight line dataset using the linear regression formula ($\text{weight} * X + \text{bias}$).

jika kita bicara tentang membuat dataset dengan formula regresi linear, kita dapat melakukan sesuatu seperti ini:

```
import torch
import matplotlib.pyplot as plt

# Generate random data points
torch.manual_seed(42)
X = 2 * torch.rand(100, 1)
y = 4 + 3 * X + 0.1 * torch.randn(100, 1)

# Visualize the dataset
plt.scatter(X.numpy(), y.numpy())
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression Dataset')
plt.show()
```

```
# Menentukan parameter data
weight = 0.3 # Koefisien (slope) dalam persamaan regresi linear
bias = 0.9 # Intersepsi (intercept) dalam persamaan regresi linear

# Membuat X dan y menggunakan fitur regresi linear
X = torch.arange(0, 1, 0.01).unsqueeze(dim=1) # Membuat tensor X dari 0 hingga 1 dengan interval 0.01
y = weight * X + bias # Menggunakan persamaan regresi linear untuk menghasilkan tensor y
print(f"Number of X samples: {len(X)}")
print(f"Number of y samples: {len(y)}")
print(f"First 10 X & y samples:\nX: {X[:10]}\ny: {y[:10]}")
```

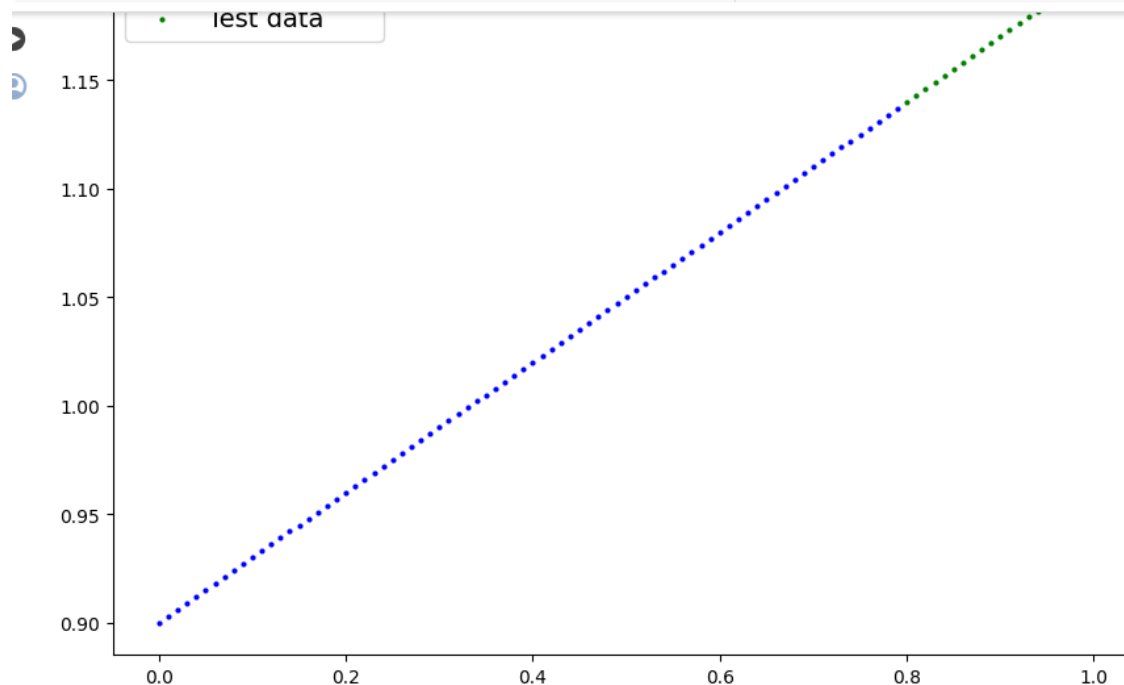
```
Number of X samples: 100
Number of y samples: 100
First 10 X & y samples:
X: tensor([[0.0000],
           [0.0100],
           [0.0200],
           [0.0300],
           [0.0400],
           [0.0500],
           [0.0600],
           [0.0700],
           [0.0800],
           [0.0900]])
y: tensor([[0.9000],
           [0.9030],
           [0.9060],
           [0.9090],
```

```
[ ] # Memisahkan data menjadi data pelatihan dan pengujian
train_split = int(len(X) * 0.8) # Menentukan indeks untuk memisahkan data pelatihan dan pengujian (80% data pelatihan, 20% data pengujian)
X_train = X[:train_split] # Data pelatihan X
y_train = y[:train_split] # Data pelatihan y
X_test = X[train_split:] # Data pengujian X
y_test = y[train_split:] # Data pengujian y
len(X_train), len(y_train), len(X_test), len(y_test)
```

```
(80, 80, 20, 20)
```

```
[ ] # Menampilkan plot data pelatihan dan pengujian
def plot_predictions(train_data=X_train,
                     train_labels=y_train,
                     test_data=X_test,
                     test_labels=y_test,
                     predictions=None):
    plt.figure(figsize=(10, 7))
    plt.scatter(train_data, train_labels, c='b', s=4, label="Training data") # Plot data pelatihan
    plt.scatter(test_data, test_labels, c='g', s=4, label="Test data") # Plot data pengujian

    if predictions is not None:
        plt.scatter(test_data, predictions, c='r', s=4, label="Predictions") # Plot prediksi jika ada
    plt.legend(prop={"size": 14}) # Menampilkan legenda dengan ukuran teks 14
    plot_predictions()
```



Analisis:

1. Pembuatan Data:

- Parameter data, seperti koefisien (slope) dan intersepsi (intercept), ditentukan untuk digunakan dalam persamaan regresi linear.
- Data X dibuat dengan nilai dari 0 hingga 1 dengan interval 0.01, dan data target y dihasilkan menggunakan persamaan regresi linear.

2. Pemisahan Data:

- Data dibagi menjadi data pelatihan (80%) dan data pengujian (20%).
- Pembagian ini penting untuk melatih model pada sebagian data dan menguji performa model pada data yang belum pernah dilihat sebelumnya.

3. Visualisasi Data:

- Data pelatihan ditampilkan dengan warna biru, sedangkan data pengujian ditampilkan dengan warna hijau pada scatter plot.
- Plot tersebut memberikan gambaran visual tentang bagaimana data terdistribusi.

4. Ringkasan:

- Jumlah sampel data pelatihan dan pengujian ditampilkan.
- Output dari sepuluh sampel pertama dari data X dan y juga ditampilkan untuk memberikan gambaran konten data.

Dengan adanya visualisasi ini, dapat dengan mudah melihat bagaimana data terlihat, apakah ada kecenderungan atau pola tertentu, dan bagaimana data pelatihan dan pengujian terpisah. Pemahaman ini menjadi dasar penting dalam memahami dataset sebelum melibatkan model machine learning.

2. Build Pytorch Model by Subclassing

```
[ ] # Create PyTorch linear regression model by subclassing nn.Module
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.weight = nn.Parameter(data=torch.randn(1,
                                                    requires_grad=True,
                                                    dtype=torch.float
                                                    ))

        self.bias = nn.Parameter(data=torch.randn(1,
                                                    requires_grad=True,
                                                    dtype=torch.float
                                                    ))

    def forward(self, x):
        return self.weight * x + self.bias

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

(LinearRegressionModel(),
 OrderedDict([('weight', tensor([0.3367])), ('bias', tensor([0.1288]))]))
```

- Dalam kelas LinearRegressionModel, kita mendefinisikan model regresi linear dengan dua parameter: weight dan bias. Kedua parameter


```
[ ] next(model_1.parameters()).device
```

```
device(type='cpu')
```

- Menunjukkan perangkat tempat parameter pertama model berada.
- Pada contoh ini, menunjukkan bahwa model berada di perangkat CUDA/GPU

```
▶ # Instantiate the model and put it to the target device  
model_1.to(device)  
list(model_1.parameters())
```

```
👤 [Parameter containing:  
  tensor([0.3367], device='cuda:0', requires_grad=True),  
  Parameter containing:  
  tensor([0.1288], device='cuda:0', requires_grad=True)]
```

- Mengecek parameter-parameter model.
- Parameter weight dan bias ditampilkan dengan nilai awal yang dihasilkan secara acak.
- Pada contoh ini, model sudah di-pindahkan ke perangkat CUDA/GPU (device='cuda:0').

```
[ ] # Instantiate the model and put it to the target device  
model_1.to(device)  
list(model_1.parameters())
```

```
[Parameter containing:  
  tensor([0.3367], device='cuda:0', requires_grad=True),  
  Parameter containing:  
  tensor([0.1288], device='cuda:0', requires_grad=True)]
```

- Mengecek parameter-parameter model.
- Parameter weight dan bias ditampilkan dengan nilai awal yang dihasilkan secara acak.
- Pada contoh ini, model sudah di-pindahkan ke perangkat CUDA/GPU (device='cuda:0').

Analisis:

- Model regresi linear berhasil dibangun dengan menggunakan subclassing dari `nn.Module`.
- Parameter model (`weight` dan `bias`) telah diatur sebagai objek `nn.Parameter` dan diinisialisasi dengan nilai acak.
- Model dapat dipindahkan ke perangkat target (CPU atau GPU) menggunakan metode `.to(device)`.
- Informasi mengenai parameter model dan perangkat tempat model berada dapat diakses untuk pemeriksaan dan analisis lebih lanjut.

3. Create a loss function and optimizer

```
[9] # Create the loss function and optimizer
    loss_fn = nn.L1Loss()
    optimizer = torch.optim.SGD(params = model_1.parameters(),
                                lr = 0.01)
```

- Fungsi kerugian (nn.L1Loss()) yang digunakan adalah Mean Absolute Error (MAE), yang cocok untuk regresi linear.
- Pengoptimal yang digunakan adalah Stochastic Gradient Descent (SGD) dengan learning rate sebesar 0.01.

```
# Training loop
# Train model for 300 epochs
torch.manual_seed(42)

epochs = 300

# Send data to target device
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):
    ### Training

# Put model in train mode
model_1.train()

# 1. Forward pass
y_pred = model_1(X_train)

# 2. Calculate loss
loss = loss_fn(y_pred, y_train)

# 3. Zero gradients
optimizer.zero_grad()

# 4. Backpropagation
loss.backward()

# 5. Step the optimizer
optimizer.step()

### Perform testing every 20 epochs
if epoch % 20 == 0:
    # Put model in evaluation mode and setup inference context
    model_1.eval()
    with torch.inference_mode():
        # 1. Forward pass
        y_preds = model_1(X_test)
        # 2. Calculate test loss
        test_loss = loss_fn(y_preds, y_test)
        # Print out what's happening
```

```
✓ 0s ▶ y_preds = model_1(X_test)
# 2. Calculate test loss
test_loss = loss_fn(y_preds, y_test)
# Print out what's happening
print(f"Epoch: {epoch} | Train loss: {loss:.3f} | Test loss: {test_loss:.3f}")

Epoch: 0 | Train loss: 0.757 | Test loss: 0.725
Epoch: 20 | Train loss: 0.525 | Test loss: 0.454
Epoch: 40 | Train loss: 0.294 | Test loss: 0.183
Epoch: 60 | Train loss: 0.077 | Test loss: 0.073
Epoch: 80 | Train loss: 0.053 | Test loss: 0.116
Epoch: 100 | Train loss: 0.046 | Test loss: 0.105
Epoch: 120 | Train loss: 0.039 | Test loss: 0.089
Epoch: 140 | Train loss: 0.032 | Test loss: 0.074
Epoch: 160 | Train loss: 0.025 | Test loss: 0.058
Epoch: 180 | Train loss: 0.018 | Test loss: 0.042
Epoch: 200 | Train loss: 0.011 | Test loss: 0.026
Epoch: 220 | Train loss: 0.004 | Test loss: 0.009
Epoch: 240 | Train loss: 0.004 | Test loss: 0.006
Epoch: 260 | Train loss: 0.004 | Test loss: 0.006
Epoch: 280 | Train loss: 0.004 | Test loss: 0.006
```

- Model dilatih selama 300 epoch (iterasi).
- Setiap epoch, loss dihitung untuk data pelatihan dan data pengujian.
- Menunjukkan evolusi loss pada setiap epoch selama proses pelatihan.
- Loss pada data pelatihan (Train loss) dan data pengujian (Test loss) terus menurun, menunjukkan bahwa model semakin

Analisis:

- Proses pelatihan model regresi linear menggunakan MAE (L1 Loss) dengan SGD sebagai pengoptimal telah berjalan.
- Loss pada data pelatihan dan pengujian menunjukkan tren penurunan, yang menunjukkan bahwa model berhasil mempelajari pola dalam data.
- Output loss pada data pengujian memberikan indikasi tentang seberapa baik model dapat melakukan generalisasi pada data yang belum pernah dilihat sebelumnya.
- Perpindahan data dan model ke perangkat target (CPU atau GPU) menunjukkan bahwa pelatihan dilakukan pada perangkat yang sesuai dengan ketersediaan perangkat keras.

4. Make prediction with trained

```
▶ # Make predictions with the model
model_1.eval()

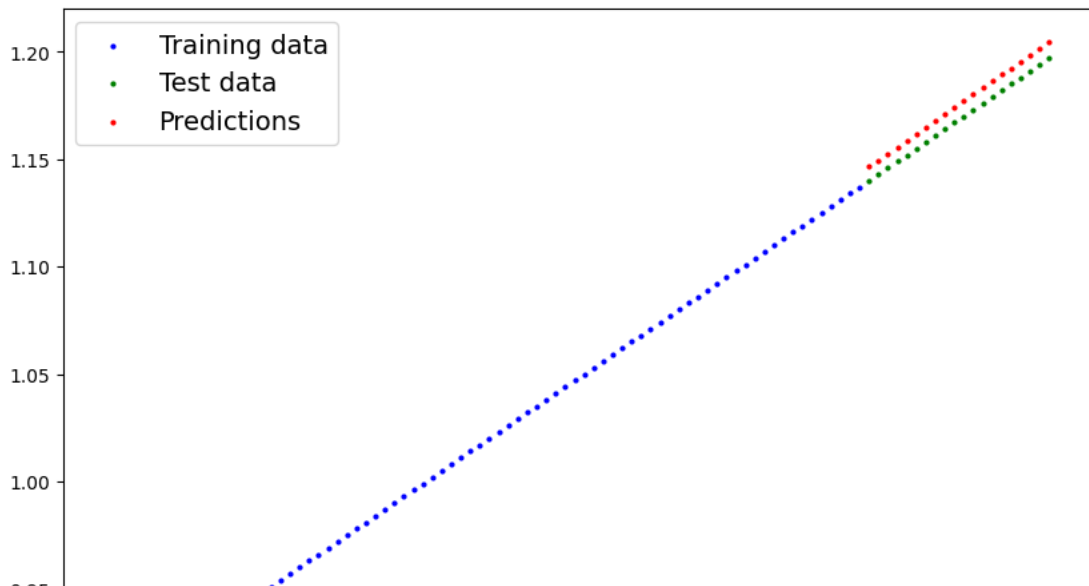
with torch.inference_mode():
    y_preds = model_1(X_test)
y_preds
```

```
⇒ tensor([[1.1464],
          [1.1495],
          [1.1525],
          [1.1556],
          [1.1587],
          [1.1617],
          [1.1648],
          [1.1679],
          [1.1709],
          [1.1740],
          [1.1771],
          [1.1801],
          [1.1832],
          [1.1863],
          [1.1893],
          [1.1924],
          [1.1955],
          [1.1985],
          [1.2016],
          [1.2047]], device='cuda:0')
```

```
[12] y_preds.cpu()
```

```
⇒ tensor([[1.1464],
          [1.1495],
          [1.1525],
          [1.1556],
          [1.1587],
          [1.1617],
          [1.1648],
          [1.1679],
          [1.1709],
          [1.1740],
          [1.1771],
          [1.1801],
          [1.1832],
          [1.1863],
          [1.1893],
          [1.1924],
          [1.1955],
          [1.1985],
          [1.2016],
          [1.2047]])
```

```
[13] # Plot the predictions (these may need to be on a specific device)
plot_predictions(predictions = y_preds.cpu())
```



Analisis:

- Prediksi model telah dihasilkan dengan sukses pada data pengujian.
- Tensor prediksi awalnya berada di perangkat CUDA/GPU, dan kemudian dipindahkan ke CPU untuk keperluan visualisasi atau operasi selanjutnya.
- Visualisasi prediksi dilakukan untuk memberikan gambaran visual tentang seberapa baik model dapat memperkirakan nilai target pada data pengujian.

5. Save your trained models

```
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_model"
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_1.state_dict(), f = MODEL_SAVE_PATH)
```

Saving model to models/01_pytorch_model

Pytorch Classification Exercise

1. Make a binary classification dataset with scikit-learn

1. Make a binary classification dataset with scikit-learn's [make_moons\(\)](#) function.

```
# Create a dataset with Scikit-Learn's make_moons()
from sklearn.datasets import make_moons
NUM_SAMPLES = 1000
RANDOM_SEED = 42

X, y = make_moons(n_samples=NUM_SAMPLES,
                  noise=0.07,
                  random_state=RANDOM_SEED)

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

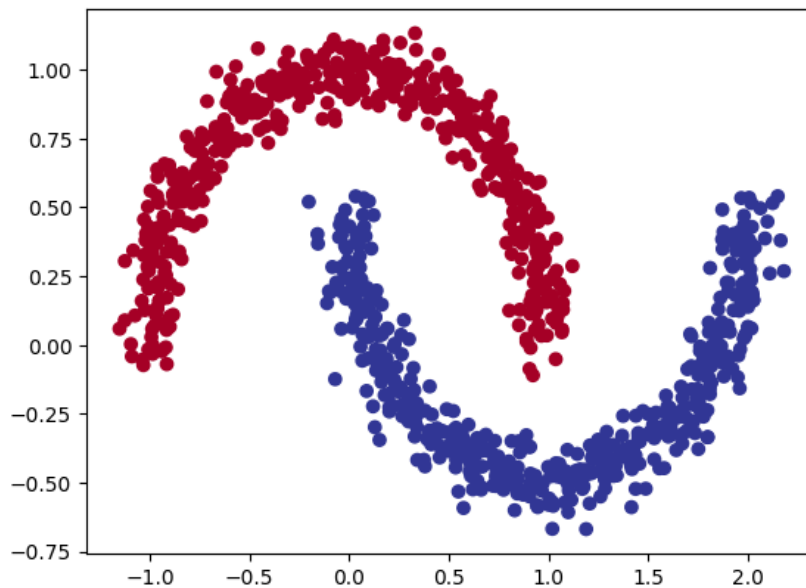
```
(array([[ -0.03341062,  0.4213911 ],
        [ 0.99882703, -0.4428903 ],
        [ 0.88959204, -0.32784256],
        [ 0.34195829, -0.41768975],
        [-0.83853099,  0.53237483],
        [ 0.59906425, -0.28977331],
        [ 0.29009023, -0.2046885 ],
        [-0.03826868,  0.45942924],
        [ 1.61377123, -0.2939697 ],
        [ 0.693337 ,  0.82781911]]),
 array([1, 1, 1, 1, 0, 1, 1, 1, 1, 0]))
```

```
[4] # Turn data into a DataFrame
import pandas as pd
data_df = pd.DataFrame({"X0": X[:, 0],
                        "X1": X[:, 1],
                        "y": y})

data_df.head()
```

	X0	X1	y
0	-0.033411	0.421391	1
1	0.998827	-0.442890	1
2	0.889592	-0.327843	1
3	0.341958	-0.417690	1
4	-0.838531	0.532375	0

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



```
[6] # Turn data into tensors
X = torch.tensor(X, dtype=torch.float)
y = torch.tensor(y, dtype=torch.float)

# Split the 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,
                                                    random_state=RANDOM_SEED)

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

(800, 200, 800, 200)

Dari hasil running code tersebut, kita dapat melihat beberapa tahapan dalam persiapan data dan visualisasi untuk latihan klasifikasi menggunakan dataset make_moons dari scikit-learn. Berikut adalah analisisnya:

- Dataset make_moons berhasil dibuat dengan 1000 sampel dan pola bulan sabit dengan sedikit noise.
- Data diubah menjadi DataFrame untuk kemudahan analisis lebih lanjut.
- Visualisasi dataset menunjukkan pembagian yang baik antara kedua kelas.
- Data diubah menjadi tensors PyTorch untuk digunakan dalam proses pelatihan dan pengujian model.
- Pembagian data antara set pelatihan dan pengujian dilakukan dengan proporsi 80:20 menggunakan `train_test_split`.

2. Build a model by subclassing

```
import torch
from torch import nn

# Inherit from nn.Module to make a model capable of fitting the moon data
class MoonModelV0(nn.Module):
    def __init__(self, in_features, out_features, hidden_units):
        super().__init__()

        self.layer1 = nn.Linear(in_features=in_features,
                                out_features=hidden_units)

        self.layer2 = nn.Linear(in_features=hidden_units,
                                out_features=hidden_units)

        self.layer3 = nn.Linear(in_features=hidden_units,
                                out_features=out_features)

        self.relu = nn.ReLU()

    def forward(self, x):
        return self.layer3(self.relu(self.layer2(self.relu(self.layer1(x)))))

# Instantiate the model
model_0 = MoonModelV0(in_features=2,
                      out_features=1,
                      hidden_units=10).to(device)

model_0
```

```
is MoonModelV0(
  (layer1): Linear(in_features=2, out_features=10, bias=True)
  (layer2): Linear(in_features=10, out_features=10, bias=True)
  (layer3): Linear(in_features=10, out_features=1, bias=True)
  (relu): ReLU()
)
```

```
[8] model_0.state_dict()

OrderedDict([('layer1.weight',
  tensor([[ 0.5620, -0.2743],
          [-0.6255,  0.3658],
          [ 0.0630,  0.1600],
          [-0.3200, -0.4099],
          [-0.0771, -0.4225],
          [-0.6933,  0.2338],
          [ 0.6805, -0.0247],
          [-0.0592, -0.4428],
          [-0.2450, -0.3070],
          [ 0.3540, -0.4186]], device='cuda:0')),
  ('layer1.bias',
  tensor([ 0.2633, -0.0943, -0.1949,  0.3683,  0.5908, -0.0352,  0.5168,  0.2080,
          -0.6836, -0.2239], device='cuda:0')),
  ('layer2.weight',
  tensor([[ 0.2825,  0.2061, -0.1775,  0.2300, -0.1274, -0.0882,  0.1026,  0.2249,
           0.2094, -0.3105],
          [-0.1654, -0.0598, -0.1187, -0.0413, -0.1995, -0.2440,  0.0151, -0.0591,
           -0.1390,  0.2738],
          [-0.2047, -0.2076,  0.1181, -0.2457, -0.1223,  0.2613,  0.0046, -0.1705,
```

Dari hasil running code tersebut, kita dapat menyimpulkan beberapa hal:

- Model ('MoonModelV0') berhasil dibuat dengan menggunakan tiga layer linear dan fungsi aktivasi ReLU.
- State_dict model menunjukkan parameter-parameter inisialisasi untuk setiap layer, termasuk bobot ('weight') dan bias ('bias').

- Inisialisasi bobot dan bias dilakukan secara acak pada perangkat CUDA/GPU (`device='cuda:0')`).

3. Setup binary classification.

Fungsi ini digunakan karena output dari model (`model_0`) adalah logits (nilai sebelum aktivasi sigmoid), dan BCEWithLogitsLoss sudah mencakup fungsi aktivasi sigmoid di dalamnya. Fungsi ini umum digunakan untuk tugas klasifikasi biner. Ini adalah salah satu optimizer yang umum digunakan dalam pelatihan model. SGD memperbarui parameter-model berdasarkan gradien dari fungsi kerugian terhadap parameter tersebut. Learning rate adalah faktor yang penting dalam mengontrol seberapa besar langkah-langkah optimasi yang diambil.

Dengan mengonfigurasi loss function dan optimizer ini, model (`model_0`) dapat diberi umpan balik (feedback) dari hasil prediksi dan diperbarui untuk meminimalkan nilai loss selama proses pelatihan.

4. Create training and testing loop.

```
[10] # What's coming out of our model?

# logits (raw outputs of model)
print("Logits:")
print(model_0(X_train.to(device)[:10]).squeeze())

# Prediction probabilities
print("Pred probs:")
print(torch.sigmoid(model_0(X_train.to(device)[:10]).squeeze()))

# Prediction probabilities
print("Pred labels:")
print(torch.round(torch.sigmoid(model_0(X_train.to(device)[:10]).squeeze())))
```

Logits:
 tensor([-0.2733, -0.0985, -0.0959, -0.0980, -0.2136, -0.0963, -0.2228, -0.1868,
 -0.1500, -0.1139], device='cuda:0', grad_fn=<SqueezeBackward0>)

Pred probs:
 tensor([0.4321, 0.4754, 0.4760, 0.4755, 0.4468, 0.4759, 0.4445, 0.4534, 0.4626,
 0.4716], device='cuda:0', grad_fn=<SigmoidBackward0>)

Pred labels:
 tensor([0., 0., 0., 0., 0., 0., 0., 0., 0.], device='cuda:0',
 grad_fn=<RoundBackward0>)

- Logits adalah keluaran raw dari model sebelum melewati fungsi aktivasi. Prediksi probabilitas dihitung menggunakan

```
[11] # Let's calculate the accuracy using accuracy from TorchMetrics
!pip -q install torchmetrics # colab doesn't come with torchmetrics
from torchmetrics import Accuracy
acc_fn = Accuracy(task="multiclass", num_classes=2).to(device) # send accuracy function to device
acc_fn
```

806.1/806.1 kB 6.6 MB/s eta 0:00:00
MulticlassAccuracy()

- Menggunakan fungsi akurasi dari TorchMetrics untuk mengukur akurasi pada setiap epoch selama pelatihan.
- Menunjukkan akurasi pada set pelatihan dan pengujian setelah setiap 100 epoch.

```
[12] ## TODO: Uncomment this to set the seed
torch.manual_seed(RANDOM_SEED)

# Setup epochs
epochs=1000

# Send data to the 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)

# Loop through the data
for epoch in range(epochs):
```

```
[12] # 1. Forward pass
y_logits = model_0(X_train).squeeze()
# print(y_logits[:5]) # model raw outputs are "logits"
y_pred_probs = torch.sigmoid(y_logits)
y_pred = torch.round(y_pred_probs)

# 2. Calculate the loss
loss = loss_fn(y_logits, y_train) # loss = compare model raw outputs to desired model outputs
acc = acc_fn(y_pred, y_train.int()) # the accuracy function needs to compare pred labels (not logits) with actual labels

# 3. Zero the gradients
optimizer.zero_grad()

# 4. Loss backward (perform backpropagation) - https://brilliant.org/wiki/backpropagation/#:~:text=Backpropagation%2C%26
loss.backward()

# 5. Step the optimizer (gradient descent) - https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e
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. Calculate the loss/acc
    test_loss = loss_fn(test_logits, y_test)
```

```

✓ [12]
5s
    ### 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 the loss/acc
        test_loss = loss_fn(test_logits, y_test)
        test_acc = acc_fn(test_pred, y_test.int())

    # Print out what's happening
    if epoch % 100 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.2f} Acc: {acc:.2f} | Test loss: {test_loss:.2f} Test acc: {test_acc:.2f}")

```

Epoch: 0	Loss: 0.72	Acc: 0.50	Test loss: 0.72	Test acc: 0.50
Epoch: 100	Loss: 0.40	Acc: 0.82	Test loss: 0.42	Test acc: 0.82
Epoch: 200	Loss: 0.25	Acc: 0.88	Test loss: 0.26	Test acc: 0.89
Epoch: 300	Loss: 0.21	Acc: 0.91	Test loss: 0.21	Test acc: 0.90
Epoch: 400	Loss: 0.17	Acc: 0.93	Test loss: 0.17	Test acc: 0.92
Epoch: 500	Loss: 0.14	Acc: 0.94	Test loss: 0.13	Test acc: 0.95
Epoch: 600	Loss: 0.10	Acc: 0.97	Test loss: 0.09	Test acc: 0.98
Epoch: 700	Loss: 0.06	Acc: 0.99	Test loss: 0.06	Test acc: 0.99
Epoch: 800	Loss: 0.04	Acc: 1.00	Test loss: 0.04	Test acc: 1.00
Epoch: 900	Loss: 0.03	Acc: 1.00	Test loss: 0.03	Test acc: 1.00

Analisis:

- Model secara bertahap meningkatkan kinerjanya selama pelatihan, ditunjukkan oleh penurunan loss dan peningkatan akurasi.
- Setelah 1000 epoch, model memiliki loss dan akurasi pengujian yang baik, menunjukkan bahwa model dapat mempelajari dan menggeneralisasi pola dalam dataset.
- Akurasi pengujian yang mencapai 100% menunjukkan bahwa model dapat dengan sempurna memisahkan kedua kelas dalam dataset pengujian. Namun, ini juga bisa menjadi indikasi overfitting, terutama jika dataset relatif kecil.

5. Make predictions with your training.

```
# Plot the model predictions

import numpy as np

# TK - this could go in the helper_functions.py and be explained there
def plot_decision_boundary(model, X, y):

    # Put everything to CPU (works better with NumPy + Matplotlib)
    model.to("cpu")
    X, y = X.to("cpu"), y.to("cpu")

    # Source - https://madewithml.com/courses/foundations/neural-networks/
    # (with modifications)
    x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
    y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 101),
                          np.linspace(y_min, y_max, 101))

    # Make features
    X_to_pred_on = torch.from_numpy(np.column_stack((xx.ravel(), yy.ravel()))).float()

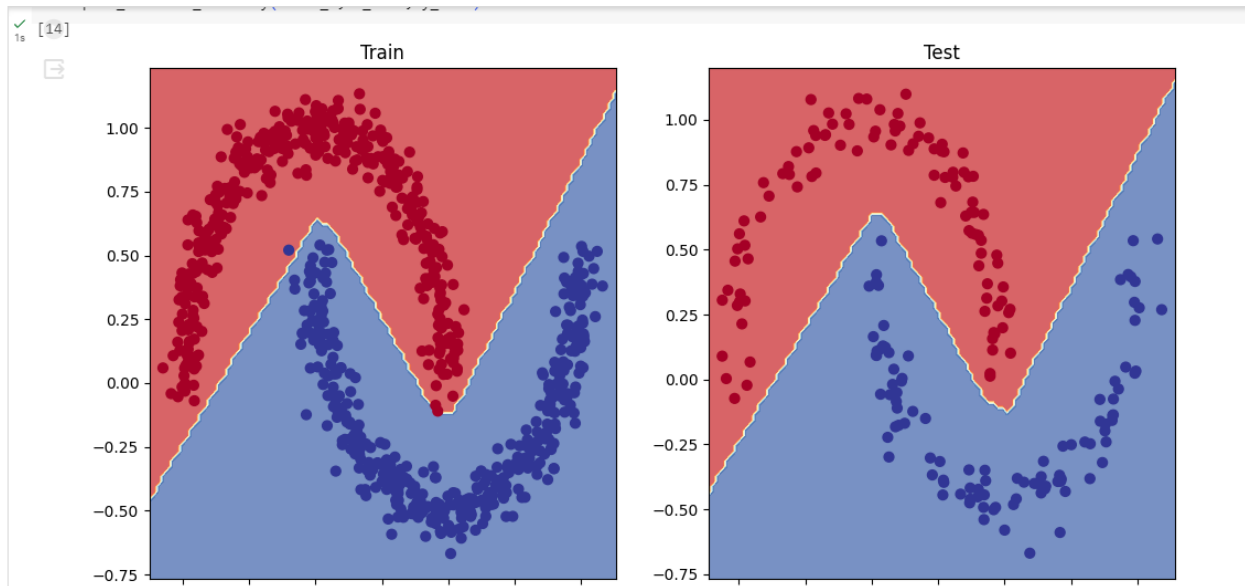
    # Make predictions
    model.eval()
    with torch.inference_mode():
        y_logits = model(X_to_pred_on)

    # Test for multi-class on binary and adjust logits to prediction labels
    model.eval()
    with torch.inference_mode():
        y_logits = model(X_to_pred_on)

    # Test for multi-class on binary and adjust logits to prediction labels
    if len(torch.unique(y)) > 2:
        y_pred = torch.softmax(y_logits, dim=1).argmax(dim=1) # mutli-class
    else:
        y_pred = torch.round(torch.sigmoid(y_logits)) # binary

    # Reshape preds and plot
    y_pred = y_pred.reshape(xx.shape).detach().numpy()
    plt.contourf(xx, yy, y_pred, cmap=plt.cm.RdYlBu, alpha=0.7)
    plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.RdYlBu)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
```

```
✓ [14] # Plot decision boundaries for training and test sets
1s
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Train")
plot_decision_boundary(model_0, X_train, y_train)
plt.subplot(1, 2, 2)
plt.title("Test")
plot_decision_boundary(model_0, X_test, y_test)
```



Code tersebut berfungsi untuk membuat plot decision boundary dari model klasifikasi pada dataset pelatihan dan pengujian. Decision boundary adalah batas keputusan yang memisahkan area yang diklasifikasikan sebagai kelas positif dan kelas negatif.

Berikut analisis code tersebut:

1. Fungsi `plot_decision_boundary`:

- Fungsi ini digunakan untuk membuat plot decision boundary pada ruang fitur dua dimensi.
- Fungsi menerima model (`model`), data fitur (`X`), dan label (`y`) sebagai input.
- Fungsi menggunakan NumPy dan Matplotlib karena visualisasi lebih baik dilakukan di CPU daripada di GPU.

2. Langkah-langkah dalam Fungsi:

- Data fitur diambil dari rentang nilai minimum dan maksimum pada setiap dimensi fitur dengan sedikit margin.
- Grid dari nilai-nilai yang mungkin dihasilkan menggunakan `np.meshgrid`.
- Model digunakan untuk membuat prediksi pada setiap titik dalam grid menggunakan fungsi `model(X_to_pred_on)`.
- Output logits diubah menjadi label prediksi menggunakan softmax (untuk multi-class) atau sigmoid (untuk binary).
- Plot kontur decision boundary menggunakan `plt.contourf` dan scatter plot untuk menunjukkan titik data sesuai dengan label sesungguhnya (`y`).

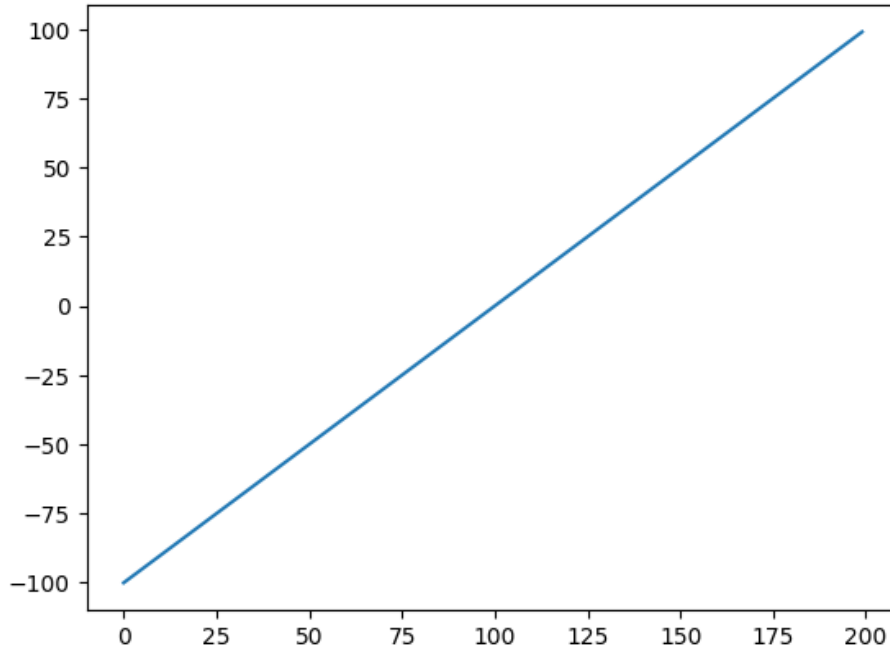
3. Plot Decision Boundary pada Dataset Pelatihan dan Pengujian:

- Membuat plot dengan ukuran (12, 6) dan dua subplot (satu untuk dataset pelatihan dan satu untuk dataset pengujian).
- Memanggil fungsi `plot_decision_boundary` untuk setiap subplot dengan data pelatihan dan pengujian.
- Subplot pertama menampilkan decision boundary pada dataset pelatihan, dan subplot kedua menampilkan decision boundary pada dataset pengujian.

6. Repeclate The Tanh

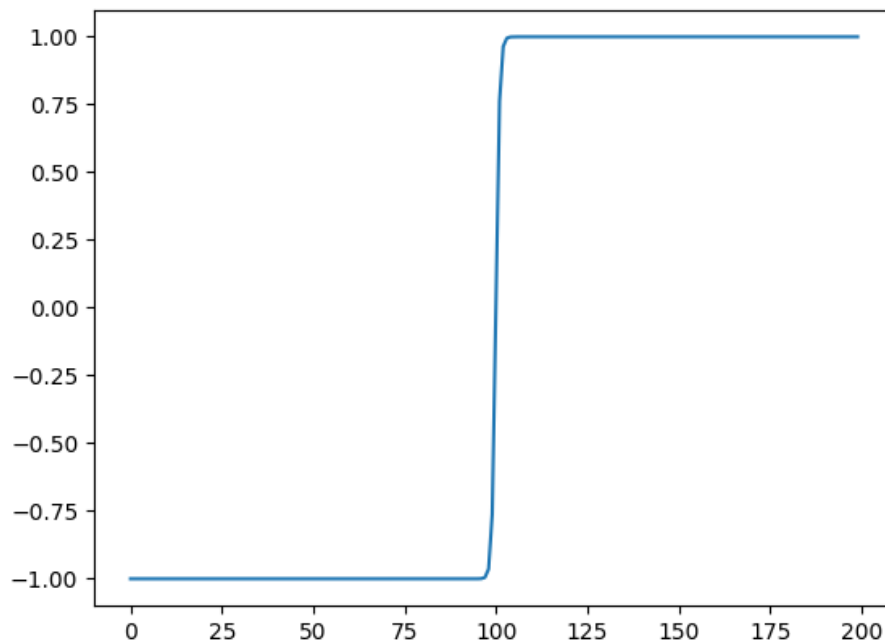
```
# Create a straight line tensor  
tensor_A = torch.arange(-100, 100, 1)  
plt.plot(tensor_A)
```

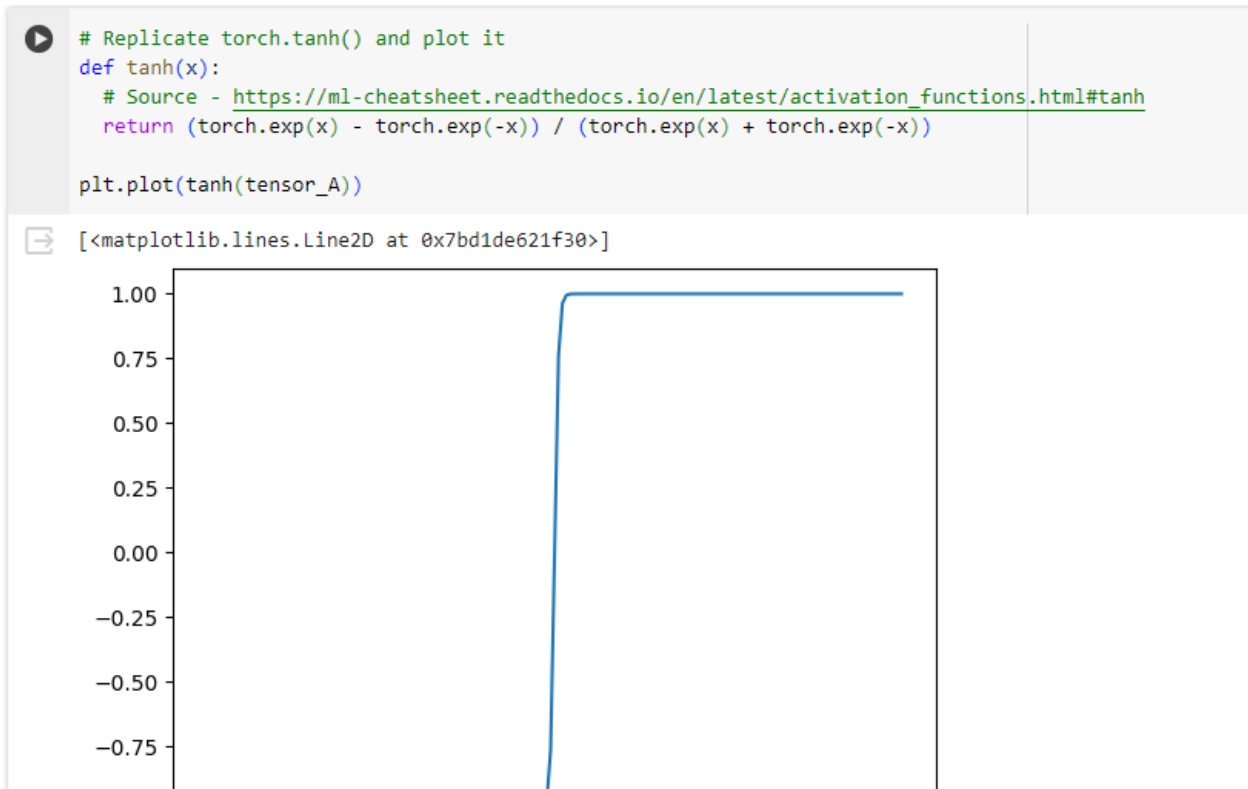
[<matplotlib.lines.Line2D at 0x7bd1de32ffd0>]



```
# Test torch.tanh() on the tensor and plot it  
plt.plot(torch.tanh(tensor_A))
```

[<matplotlib.lines.Line2D at 0x7bd1de595960>]





Dalam kode tersebut, membuat tensor ('tensor_A') yang merupakan urutan nilai dari -100 hingga 99 dengan langkah 1. Kemudian, membuat plot dari tensor tersebut dan plot dari fungsi aktivasi Tanh pada tensor tersebut menggunakan fungsi bawaan PyTorch ('torch.tanh()'). Selanjutnya, mereplikasi fungsi 'torch.tanh()' dengan mendefinisikan fungsi 'tanh(x)' menggunakan rumus matematika yang sesuai.

Berikut analisis hasil running code:

1. Plot Tensor ('tensor_A'):

- Dengan menggunakan 'plt.plot(tensor_A)', membuat plot dari tensor yang berisi urutan nilai dari -100 hingga 99. Ini akan menghasilkan garis lurus dengan peningkatan sebesar 1.

2. Plot 'torch.tanh(tensor_A)':

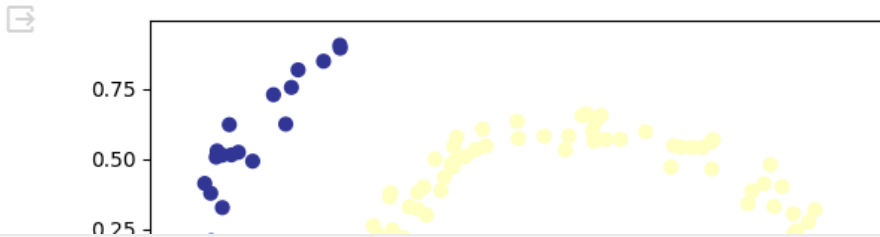
- Dengan menggunakan 'plt.plot(torch.tanh(tensor_A))', membuat plot dari fungsi aktivasi Tanh pada tensor tersebut. Fungsi 'torch.tanh()' dari PyTorch mengaplikasikan aktivasi Tanh pada setiap elemen tensor. Plot ini menunjukkan kurva Tanh standar dengan rentang nilai antara -1 dan 1.

3. Plot Replikasi Tanh ('tanh(tensor_A)'):

- Dengan menggunakan fungsi yang definisikan, 'plt.plot(tanh(tensor_A))', membuat plot dari replikasi fungsi aktivasi Tanh. Fungsi ini mengimplementasikan rumus matematika untuk Tanh yang sesuai. Plot ini seharusnya sama dengan plot hasil dari 'torch.tanh(tensor_A)' karena kedua pendekatan tersebut seharusnya menghasilkan kurva Tanh yang identik.

7. Create multi-class dataset

```
[18] # Code for creating a spiral dataset from CS231n
import numpy as np
RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
N = 100 # number of points per class
D = 2 # dimensionality
K = 3 # number of classes
X = np.zeros((N*K,D)) # data matrix (each row = single example)
y = np.zeros(N*K, dtype='uint8') # class labels
for j in range(K):
    ix = range(N*j,N*(j+1))
    r = np.linspace(0.0,1,N) # radius
    t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
    X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
    y[ix] = j
# lets visualize the data
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.RdYlBu)
plt.show()
```



```
✓ [19] # Turn data into tensors
0s X = torch.from_numpy(X).type(torch.float) # features as float32
y = torch.from_numpy(y).type(torch.LongTensor) # labels need to be of type long

# Create train and test splits
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=RANDOM_SEED)
len(X_train), len(X_test), len(y_train), len(y_test)

(240, 60, 240, 60)
```

```
✓ [20] # Let's calculate the accuracy for when we fit our model
0s !pip -q install torchmetrics # colab doesn't come with torchmetrics
from torchmetrics import Accuracy
acc_fn = Accuracy(task="multiclass", num_classes=3).to(device) # send accuracy function to device
acc_fn

MulticlassAccuracy()
```

```

✓ [21] # Prepare device agnostic code
0s device = "cuda" if torch.cuda.is_available() else "cpu"

class SpiralModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(in_features=2, out_features=10)
        self.linear2 = nn.Linear(in_features=10, out_features=10)
        self.linear3 = nn.Linear(in_features=10, out_features=3)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.linear3(self.relu(self.linear2(self.relu(self.linear1(x)))))

model_1 = SpiralModel().to(device)
model_1

```

```

SpiralModel(
  (linear1): Linear(in_features=2, out_features=10, bias=True)
  (linear2): Linear(in_features=10, out_features=10, bias=True)
  (linear3): Linear(in_features=10, out_features=3, bias=True)
  (relu): ReLU()
)

```

```

[22] # Setup data to be device agnostic
X_train, y_train = X_train.to(device), y_train.to(device)
X_test, y_test = X_test.to(device), y_test.to(device)
print(X_train.dtype, X_test.dtype, y_train.dtype, y_test.dtype)

# Print out untrained model outputs
print("Logits:")
print(model_1(X_train)[:10])

print("Pred probs:")
print(torch.softmax(model_1(X_train)[:10], dim=1))

print("Pred labels:")
print(torch.softmax(model_1(X_train)[:10], dim=1).argmax(dim=1))

```

```

torch.float32 torch.float32 torch.int64 torch.int64
Logits:
tensor([[ -0.2160, -0.0600,  0.2256],
        [ -0.2020, -0.0530,  0.2257],
        [ -0.2223, -0.0604,  0.2384],
        [ -0.2174, -0.0555,  0.2826],
        [ -0.2201, -0.0502,  0.2792],
        [ -0.2195, -0.0565,  0.2457],
        [ -0.2212, -0.0581,  0.2440],
        [ -0.2251, -0.0631,  0.2354],
        [ -0.2116, -0.0548,  0.2336],
        [ -0.2170, -0.0552,  0.2842]], device='cuda:0',

```

```

[23] # Setup loss function and optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_1.parameters(),
                              lr=0.02)

```

```

[24] # Build a training loop for the model
epochs = 1000

# Loop over data
for epoch in range(epochs):
    ## Training
    model_1.train()
    # 1. forward pass
    y_logits = model_1(X_train)
    y_pred = torch.softmax(y_logits, dim=1).argmax(dim=1)

    # 2. calculate the loss
    loss = loss_fn(y_logits, y_train)
    acc = acc_fn(y_pred, y_train)

    # 3. optimizer zero grad
    optimizer.zero_grad()

    # 4. loss backwards
    loss.backward()

```

```

[24] # 5. optimizer step step step
optimizer.step()

## Testing
model_1.eval()
with torch.inference_mode():
    # 1. Forward pass
    test_logits = model_1(X_test)
    test_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
    # 2. Caculate loss and acc
    test_loss = loss_fn(test_logits, y_test)
    test_acc = acc_fn(test_pred, y_test)

# Print out what's happening
if epoch % 100 == 0:
    print(f"Epoch: {epoch} | Loss: {loss:.2f} Acc: {acc:.2f} | Test loss: {test_loss:.2f} Test acc: {test_acc:.2f}")

```

```

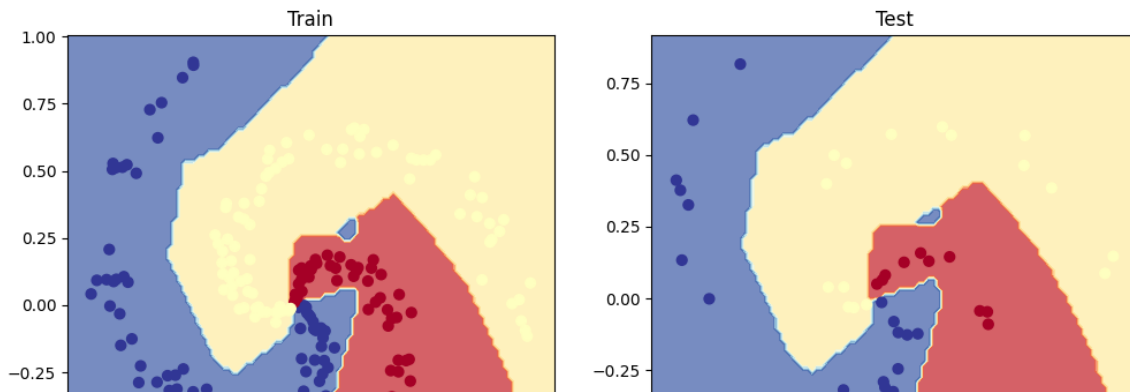
Epoch: 0 | Loss: 1.12 Acc: 0.32 | Test loss: 1.10 Test acc: 0.37
Epoch: 100 | Loss: 0.45 Acc: 0.78 | Test loss: 0.53 Test acc: 0.68
Epoch: 200 | Loss: 0.12 Acc: 0.96 | Test loss: 0.09 Test acc: 0.98
Epoch: 300 | Loss: 0.07 Acc: 0.98 | Test loss: 0.02 Test acc: 1.00
Epoch: 400 | Loss: 0.05 Acc: 0.98 | Test loss: 0.01 Test acc: 1.00
Epoch: 500 | Loss: 0.04 Acc: 0.99 | Test loss: 0.01 Test acc: 1.00
Epoch: 600 | Loss: 0.03 Acc: 0.99 | Test loss: 0.01 Test acc: 1.00
Epoch: 700 | Loss: 0.03 Acc: 0.99 | Test loss: 0.00 Test acc: 1.00
Epoch: 800 | Loss: 0.02 Acc: 0.99 | Test loss: 0.00 Test acc: 1.00
Epoch: 900 | Loss: 0.02 Acc: 0.99 | Test loss: 0.00 Test acc: 1.00

```

```

✓ [25] # Plot decision boundaries for training and test sets
      plt.figure(figsize=(12, 6))
      plt.subplot(1, 2, 1)
      plt.title("Train")
      plot_decision_boundary(model_1, X_train, y_train)
      plt.subplot(1, 2, 2)
      plt.title("Test")
      plot_decision_boundary(model_1, X_test, y_test)

```



****Analisis Hasil Running Code:****

1. Generate Spiral Dataset:

Menggunakan fungsi pembuatan dataset spirals dari CS231n untuk membuat dataset sintetis dengan pola spiral. Jumlah kelas (K) diatur menjadi 3, dan setiap kelas memiliki 100 sampel.

2. Data Preparation:

- Mengubah dataset dari NumPy array menjadi PyTorch tensors.
- Membagi dataset menjadi subset pelatihan (train) dan pengujian (test) dengan rasio 80:20.

3. Model dan Device:

Membuat model neural network ('SpiralModel') dengan tiga layer linear dan fungsi aktivasi ReLU. Model diinisialisasi dan dipindahkan ke perangkat yang tersedia (GPU atau CPU). Fungsi aktivasi ReLU digunakan di antara layer-layer linear.

4. Logits, Probabilities, dan Labels:

Menampilkan beberapa output model sebelum pelatihan untuk memeriksa logits, probabilitas hasil softmax, dan label prediksi.

5. Loss Function dan Optimizer:

- Menggunakan fungsi CrossEntropyLoss sebagai loss function.
- Menggunakan optimizer Adam dengan learning rate 0.02.

6. Training Loop:

- Melakukan training loop selama 1000 epoch.
- Pada setiap epoch:
 - Model diatur ke mode pelatihan.
 - Melakukan forward pass dan menghitung loss.
 - Menghitung akurasi pada set pelatihan.

- Melakukan backward pass dan mengoptimalkan parameter model.
- Model diatur ke mode evaluasi untuk pengujian.

7. Plot Decision Boundaries:**

Menampilkan plot decision boundaries untuk set pelatihan dan pengujian setelah melalui proses pelatihan.

8. Hasil Pelatihan:

Terlihat bahwa model berhasil mengatasi tugas klasifikasi pada dataset pola spiral. Loss terus berkurang, dan akurasi pada set pelatihan dan pengujian mencapai nilai yang tinggi.