

Nama : Fasya Hanifah Putti

NIM : 1103200149

KELAS : MACHINE LEARNING

RANGKUMAN MACHINE LEARNING CHAPTER 00.Dasar-Dasar PyTorch

PyTorch merupakan kerangka kerja machine learning yang memungkinkan kita untuk memanipulasi dan memproses data dan menulis algoritma machine learning menggunakan python. PyTorch banyak digunakan Perusahaan teknologi besar seperti Meta (Facebook), Tesla, Microsoft, serta Open AI. PyTorch adalah pembelajaran machine learning yang disukai oleh peneliti karena kerja kerja ini membahas secara mendalam dan paling banyak digunakan di Papers With Code. Disini kita akan memulai dengan pengantar tensor. Apa itu tensor? Tensor adalah blok bangunan dasar dari semua machine learning dengan pembelajaran yang mendalam.

1. Mengimpor Torch

Import modul PyTorch menggunakan syntax dibawah

```
68 [1] import torch  
      torch.__version__  
  
'2.1.0+cu121'
```

Terlihat bahwa modul PyTorch yang kita gunakan memiliki versi 2.1.0+

2. Pengantar Tensor

Tensor adalah elemen dasar machine learning. Tugas tensor mempresentasikan data secara numerik. Dalam bahasa tensor, tensor akan memiliki tiga dimensi, satu untuk colour_channels, height, dan width.

3. Membuat Tensor

Hal yang pertama yang akan dibuat adalah skalar. Skalar adalah angka tunggal. Dari syntax diatas kita akan membuat tensor dengan nilai skalar 7. Terbukti bahwa skalar merupakan nilai tunggal yang biasanya disimpan dan direpresentasikan pada tensor skalar. Jika ingin memeriksa dimensi tensor dapat menggunakan syntax skalar.ndim dan untuk mengambil nomor tensor bisa gunakan fungsi item().

```
✓ [2] # Scalar
os scalar = torch.tensor(7)
scalar

tensor(7)

See how the above printed out tensor(7) ?
That means although scalar is a single number, it's of type torch.Tensor.
We can check the dimensions of a tensor using the ndim attribute.

✓ [3] scalar.ndim
os 0

What if we wanted to retrieve the number from the tensor?
As in, turn it from torch.Tensor to a Python integer?
To do we can use the item() method.

✓ [4] # Get the Python number within a tensor (only works with scalars)
os scalar.item()
os 7
```

Apa itu vektor? Vektor adalah tensor berdimensi tunggal tetapi dapat memuat banyak bilangan. Misalnya dalam syntax dibawah terdapat dua angka dalam array ini berarti vector tersebut berisi dua angka 7.

```
✓ [5] # Vector
os vector = torch.tensor([7, 7])
vector

tensor([7, 7])

✓ [6] # Check the number of dimensions of vector
os vector.ndim
os 1
```

Kira-kira kenapa dimensinya 1? padahal angkanya ada 2?, kita dapat mengetahui jumlah dimensi yang dimiliki tensor dengan jumlah kurung sikunya () dan hanya perlu dihitung satu sisi. Lalu, untuk menentukan bentu dari vektor dapat dilakukan dengan syntax dibawah.

```
✓ [7] # Check shape of vector
os vector.shape
os torch.Size([2])
```

Hasilnya didapatkan [2] ini menandakan bahwa vektor tersebut memiliki bentuk [2] ini dikarenakan terdapat 2 elemen pada vektor tersebut.

MATRIKS

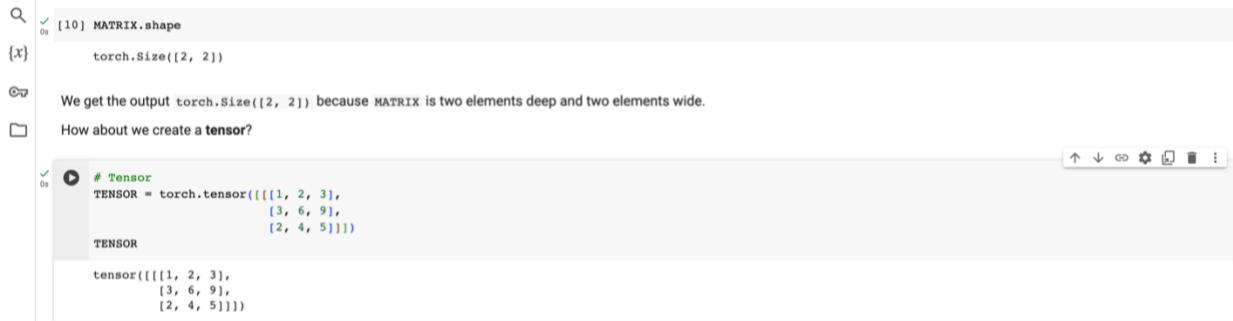
```
✓ [8] # Matrix
os MATRIX = torch.tensor([[7, 8],
                         [9, 10]])
MATRIX

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

Wow! More numbers! Matrices are as flexible as vectors, except they've got an extra dimension.

✓ [9] # Check number of dimensions
os MATRIX.ndim
os 2
```

Matriks sama fleksiblenya dengan vector. Terlihat bahwa diatas merupakan matriks yang memiliki 4 angka. Selanjutnya kita hitung dimensinya, karena matriks tersebut memiliki 2 kurung siku. Lalu, berapa ukuran yang dimiliki matriks tersebut?



```
0s [10] MATRIX.shape
{x}
torch.Size([2, 2])

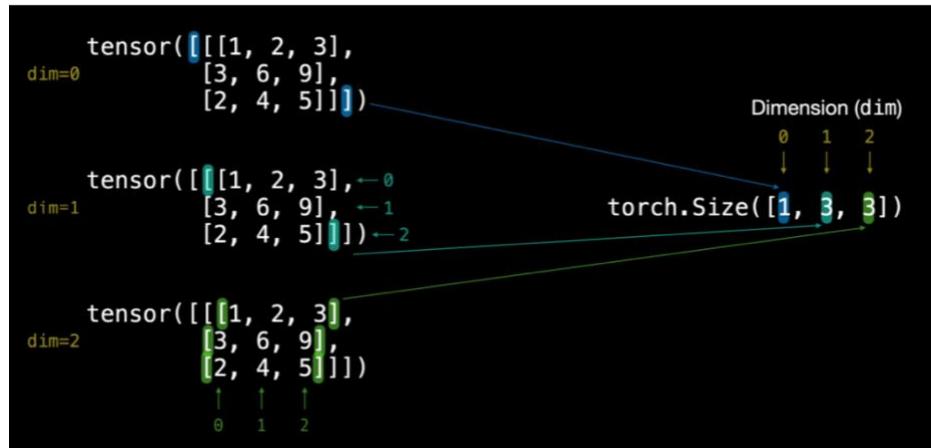
0s We get the output torch.Size([2, 2]) because MATRIX is two elements deep and two elements wide.

0s How about we create a tensor?

0s # Tensor
TENSOR = torch.tensor([[1, 2, 3],
                      [3, 6, 9],
                      [2, 4, 5]])

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

Matriks tersebut memiliki ukuran [2,2], karena matriks memiliki dua elemen kedalam dan dua element lebar. Contoh syntax diatas adalah pembuatan tensor menggunakan matriks dengan data yang diounya adalah dari angka penjualan toko steak dan mentega almond.



Biasanya scalar dan vector dilambangkan dengan huruf kecil seperti y atau a, sedangkan matriks dan tensor biasanya dilambangkan dengan huruf besar seperti X atau W. Jadi, dapat kita simpulkan bahwa skalar merupakan nilai tunggal, vektor adalah angka yang mempunyai arah, matriks adalah deretan angka 2 dimensi dan tensor adalah array bilangan berdimensi h.

Scalar	Vector
7	$\begin{bmatrix} 7 \\ 4 \end{bmatrix}$ or $\begin{bmatrix} 7 & 4 \end{bmatrix}$
Matrix	Tensor
$\begin{bmatrix} 7 & 10 \\ 4 & 3 \\ 5 & 1 \end{bmatrix}$	$\begin{bmatrix} [7 & 4] & [0 & 1] \\ [1 & 9] & [2 & 3] \\ [5 & 6] & [8 & 8] \end{bmatrix}$

Tensor Acak

Sebagai data scientist, kita dapat menentukan cara model machine learning dimulai dari inisialisasi, representasi, dan optimasi angka randomnya. Untuk membuat tensor bilangan random, kita dapat menggunakan fungsi `torch.rand()` dan meneruskan parameter `size`. Contohnya seperti syntax dibawah ini :

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

(tensor([[0.4728, 0.7747, 0.4481, 0.2942],
        [0.2087, 0.6598, 0.1997, 0.4530],
        [0.6139, 0.2833, 0.6973, 0.4668]]),
 torch.float32)
```

Ukuran dari tensor diatas adalah 3, karena syntax tersebut memiliki 3 kurung siku.

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

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

Nol dan Satu

```

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

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

We can do the same to create a tensor of all ones except using torch.ones\(\) instead.

```

```

✓ ⓘ # Create a tensor of all ones
ones = torch.ones(size=(3, 4))
ones, ones.dtype

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

```

Biasanya kita juga hanya ingin mengisi tensor dengan nilai 0 dan 1 , maka kita bisa membuatnya dengan menggunakan fungsi `torch.zeros()` untuk mengisi tensor penuh dengan angka nol dan fungsi `torch.ones()` untuk mengisi tensor penuh dengan angka 1.

Membuat rentang dan tensor sejenisnya

Terkadang kita hanya menginginkan rentang angka tertentu misalnya 1 hingga 20 atau 10 hingga 100, dan sebagainya. Kita dapat menggunakan `torch.arange(start,end,step)` untuk membuat rentang tersebut.

```

 ⓘ ⓘ # Use torch.arange(), torch.range() is deprecated
zero_to_ten_DEPRECATED = torch.range(0, 10) # Note: this may return an error in the future

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

@ <ipython-input-18-a09072c806d9>:2: UserWarning: torch.range is deprecated and will be removed in a future release because its behavior is inconsistent with E
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])

Sometimes you might want one tensor of a certain type with the same shape as another tensor.
For example, a tensor of all zeros with the same shape as a previous tensor.
To do so you can use torch.zeros\_like\(input\) or torch.ones\_like\(input\), which return a tensor filled with zeros or ones in the same
shape as the input respectively.

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

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

```

Dalam syntax tersebut terdapat `range()` untuk membuat rentang kemudian untuk keterangan start yaitu nilai awal rentang, end nilai akhir rentang, dan step berapa banyak langkah di antara setiap nilai rentang tersebut. Lalu outputnya akan menampilkan rentang yang diinginkan yaitu dari 0 hingga 9. Kita juga dapat mengubah tensor yang isi semuanya bernilai 0 dengan menggunakan fungsi `torch.zeros_like(input)`, maka output yang dihasilkan rentangnya akan bernilai 0.

Tipe Data Tensor

Tensor memiliki banyak tipe data, ada yang khusus CPU maupun GPU. Umumnya yang banyak digunakan adalah torch.cuda, tensor yang digunakan untuk GPU, ada juga yang paling umum adalah torch.float32 atau torch.float, berbagai macam tipe data torch.float ada yang 8-bit, 16-bit, 32-bit, maupun 64-bit.

```
✓ [20] # 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'))
```

Mendapat informasi dari tensor

Setelah membuat tensor, ada tiga atribut yang umum untuk diketahui :

- Shape : bentuk tensor yang digunakan
- Dtype : tipe data yang menyimpan elemen dalam tensor
- Device : pada perangkat apa tensor disimpan

Contohnya seperti dibawah ini :

```
✓ [30] # 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([[0.8694, 0.5677, 0.7411, 0.4294],
              [0.8854, 0.5739, 0.2666, 0.6274],
              [0.2696, 0.4414, 0.2969, 0.8317]])
      Shape of tensor: torch.Size([3, 4])
      Datatype of tensor: torch.float32
      Device tensor is stored on: cpu
```

Hasilnya terdapat matriks dengan ukuran tensor [3,4] data type yang digunakan torch.float32 dan perangkat yang digunakan adalah cpu.

Operasi Tensor

Dalam tensor terdapat beberapa operasi-operasi yang sering digunakan ada :

1. Pertambahan
2. Pengurangan
3. Perkalian matriks
4. Pembagian
5. Perkalian (berdasarkan elemen)

Operasi Dasar

Operasi dasar penjumlahan (+), pengurangan (-), dan perkalian (*). Kita dapat menggunakan syntax syntax dibawah ini :

```

 00_pytorch_fundamentals.ipynb
File Edit View Insert Runtime Tools Help Cannot save changes
+ Code + Text ⌂ Copy to Drive
Q [15] # Create a tensor of values and add a number to it
tensor = torch.tensor([1, 2, 3])
tensor + 10
tensor([11, 12, 13])

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

Notice how the tensor values above didn't end up being tensor([110, 120, 130]), this is because the values inside the tensor don't change unless they're reassigned.

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

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

[18] # Subtract and reassign
tensor = tensor - 10
tensor
tensor([1, 2, 3])

[19] # Add and reassign
tensor = tensor + 10
tensor

```

0s completed at 1:17 AM

Selain dengan cara manual, pytorch juga menyediakan fungsi seperti `torch.mul()` untuk perkalian dan `torch.add()` untuk penambahan.

Perkalian Matriks

Salah satu operasi yang paling umum dalam machine learning dan algoritma neural network adalah perkalian matriks. PyTorch mengimplementasikan fungsionalitas perkalian matriks dalam `torch.matmul()`. Ini adalah dua bagian matriks yang harus diingat

1. Dimensi **bagian dalam** harus sesuai:

- $(3, 2) @ (3, 2)$ tidak akan berhasil
- $(2, 3) @ (3, 2)$ akan bekerja
- $(3, 2) @ (2, 3)$ akan bekerja

2. Matriks yang dihasilkan berbentuk **dimensi luar**:

- $(2, 3) @ (3, 2) \rightarrow (2, 2)$
- $(3, 2) @ (2, 3) \rightarrow (3, 3)$

Ini adalah perbedaan antara perkalian elemen dan perkalian matriks dimana dari penjumlahan nilainya.

Operasi	Perhitungan	Kode
Perkalian berdasarkan elemen	$[1*1, 2*2, 3*3] = [1, 4, 9]$	<code>tensor * tensor</code>
Perkalian matriks	$[1*1 + 2*2 + 3*3] = [14]$	<code>tensor.matmul(tensor)</code>

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

CPU times: user 1.29 ms, sys: 104 µs, total: 1.4 ms
Wall time: 1.31 ms
tensor(14)

[27]: torch.matmul(tensor, tensor)

CPU times: user 178 µs, sys: 39 µs, total: 217 µs
Wall time: 144 µs
tensor(14)
```

Syntax pertama digunakan untuk mengukur waktu eksekusi operasi perkalian matriks secara manual. Kode tersebut melakukan iterasi melalui elemen-elemen suatu tensor menggunakan perulangan for dan menghitung jumlah dari kuadrat setiap elemen. Syntax kedua untuk mengukur waktu eksekusi perkalian matriks menggunakan fungsi `torch.matmul()`. Fungsi ini dirancang untuk untuk menangani operasi perkalian matriks dengan efisien dan memberikan kinerja yang lebih baik dibandingkan dengan iterasi manual.

Salah satu kesalahan paling umum dalam pembelajaran mendalam (kesalahan bentuk)

Dalam konteks pembelajaran mendalam, kesalahan bentuk yang sering terjadi adalah kesalahan dalam perkalian matriks.

```

os [28] # Shapes need to be in the right way
tensor_A = torch.tensor([[1, 2],
                       [3, 4],
                       [5, 6]], dtype=torch.float32)

tensor_B = torch.tensor([[7, 10],
                       [8, 11],
                       [9, 12]], dtype=torch.float32)

torch.matmul(tensor_A, tensor_B) # (this will error)

-----
RuntimeError Traceback (most recent call last)
<ipython-input-28-aceec990e652> in <cell line: 10>(8)
      8         [9, 12]], dtype=torch.float32)
      9
----> 10 torch.matmul(tensor_A, tensor_B) # (this will error)
     11
RuntimeError: mat1 and mat2 shapes cannot be multiplied (3x2 and 3x2)

```

Disini kita melihat contoh dimana terjadi kesalahan bentuk ketika mencoba mengalikan kedua tensor 'tensor_A' dan 'tensor_B' yang keduanya memiliki bentuk [3,2].

```

os [36] # View tensor_A and tensor_B
print(tensor_A)
print(tensor_B)

tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
tensor([[ 7., 10.],
        [ 8., 11.],
        [ 9., 12.]])  

os [35] # View tensor_A and tensor_B.T
print(tensor_A)
print(tensor_B.T)

tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
tensor([[ 7.,  8.],
        [10., 11.],
        [12., 13.]])  

os [37] # The operation works when tensor_B is transposed
print("Original shapes: tensor_A = (%s), tensor_B = (%s)\n" % (str(tensor_A.shape), str(tensor_B.shape)))
print("New shapes: tensor_A = (%s) (same as above), tensor_B.T = (%s)\n" % (str(tensor_A.shape), str(tensor_B.T.shape)))
print("Multiplying: (%s) * (%s) <- inner dimensions match\n" % (str(tensor_A.shape), str(tensor_B.T.shape)))
print("Output:\n")
output = torch.matmul(tensor_A, tensor_B.T)
print(output)
print("\nOutput shape: (%s)" % str(output.shape))

Original shapes: tensor_A = torch.Size([3, 2]), tensor_B = torch.Size([3, 2])
New shapes: tensor_A = torch.Size([3, 2]) (same as above), tensor_B.T = torch.Size([2, 3])
Multiplying: torch.Size([3, 2]) * torch.Size([2, 3]) <- inner dimensions match

```

Untuk memperbaiki error, kita dapat menggunakan transposisi salah satu tensor sehingga bentuk atau dimensi matriksnya cocok untuk perkalian matriks. Dalam PyTorch, transposisi dapat dilakukan dengan `torch.transpose(input, dim0, dim1)` atau menggunakan `.T` pada tensor. Setelah transposisi `tensor_B`, dimensi yang cocok adalah [3, 2] untuk `tensor_A` dan [2, 3] untuk `tensor_B.T`, memungkinkan perkalian matriks berlangsung tanpa kesalahan. Hasilnya adalah tensor dengan bentuk [3, 3].

The screenshot shows two Jupyter Notebooks in Google Colab. The top notebook, titled '00_pytorch_fundamentals.ipynb', demonstrates matrix multiplication. It shows code to create two tensors and multiply them using the dot product. Below the code, there's a visual representation of the multiplication of two 3x3 matrices:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 27 & 30 & 33 \\ 61 & 68 & 75 \\ 95 & 106 & 117 \end{bmatrix}$$

The bottom notebook, also titled '00_pytorch_fundamentals.ipynb', shows code for a linear layer. It creates a Linear module with `in_features=2` and `out_features=6`, and applies it to a tensor `x`. The output shape is `(3, 6)`.

Untuk memperbaiki error, kita dapat menggunakan transposisi salah satu tensor sehingga bentuk atau dimensi matriksnya cocok untuk perkalian matriks. Dalam PyTorch, transposisi dapat dilakukan dengan `torch.transpose(input, dim0, dim1)` atau menggunakan `.T` pada tensor. Setelah transposisi `tensor_B`, dimensi yang cocok adalah `[3, 2]` untuk `tensor_A` dan `[2, 3]` untuk `tensor_B.T`, memungkinkan perkalian matriks berlangsung tanpa kesalahan. Hasilnya adalah tensor dengan bentuk `[3, 3]`. Kesalahan bentuk ini juga berlaku untuk lapisan linier dalam PyTorch. Misalnya, `torch.nn.Linear(in_features, out_features)` menggunakan perkalian matriks, di mana `in_features` dari input harus cocok dengan dimensi dalam matriks bobot lapisan. Jika `in_features` tidak cocok, akan terjadi kesalahan bentuk. Contoh yang diberikan

menunjukkan bagaimana mengubah `in_features` dan `out_features` dapat mempengaruhi hasil. Misalnya, jika `in_features` diubah dari 2 menjadi 3 tetapi input masih memiliki bentuk [3, 2], akan terjadi kesalahan karena ketidakcocokan bentuk.

Menemukan min, max, mean, jumlah, dll (agregasi)

Untuk menentukan nilai min,max,mean, dll. Pertama, kita akan membuat tensornya terlebih dahulu.

The screenshot shows a Jupyter Notebook interface with the following code in cell 46:

```
[46] # Create a tensor
x = torch.arange(0, 100, 10)
x
tensor([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Cell 47 contains a note and some print statements:

```
[47] 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()}")
```

The output of cell 47 is:

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

A note in the cell states: "Note: You may find some methods such as `torch.mean()` require tensors to be in `torch.float32` (the most common) or another specific datatype, otherwise the operation will fail."

Cell 48 shows a comparison:

```
[48] torch.max(x), torch.min(x), torch.mean(x.type(torch.float32)), torch.sum(x)
(tensor(90), tensor(0), tensor(45.), tensor(450))
```

Below cell 48, there is a section titled "Positional min/max".

Disini isi dalam tensor adalah nilai dari 0 hingga 90 dengan kelipatan 10. Kita dalam menentukan nilai min yaitu 0, max 90, mean 45.0 dan jumlahnya adalah 450 jika dengan menggunakan manual. Tetapi jika ingin menggunakan fungsi dari tensornya, kita dapat menggunakan fungsi `torch.mean()` yang berada di `torch.float32`

Posisi min/maks

Kita dapat menemukan index tensor nilai maks dan min menggunakan fungsi `torch.argmax()` dan `torch.argmin()`.

```

49] # 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

```

Ubah Tipe Data Tensor

Masalah umum dengan operasi pembelajaran mendalam adalah memiliki tensor dalam tipe data yang berbeda, jika satu tensor masuk `torch.float64` dan tensor lainnya masuk `torch.float32` maka akan terdapat error. Semakin rendah angka tipe datanya maka semakin kurang tepat computer menyimpan nilainya dan dengan jumlah penyimpanan yang lebih rendah, menghasilkan komputasi yang lebih cepat dan model keseluruhan yang lebih kecil.

The screenshot shows a Jupyter Notebook interface in Google Colab. The notebook has a single cell containing Python code related to tensors. The code includes creating a tensor, checking its datatype, and creating a float16 tensor. It also includes a note about precision and an exercise link.

```

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

torch.float32

Now we'll create another tensor the same as before but change its datatype to torch.float16.

[51] # Create a float16 tensor
tensor_float16 = tensor.type(torch.float16)
tensor_float16

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

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

[52] # 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)

Note: Different datatypes can be confusing to begin with. But think of it like this, the lower the number (e.g. 32, 16, 8), the less precise a computer stores the value. And with a lower amount of storage, this generally results in faster computation and a smaller overall model. Mobile-based neural networks often operate with 8-bit integers, smaller and faster to run but less accurate than their float32 counterparts. For more on this, I'd read up about precision in computing.
Exercise: So far we've covered a fair few tensor methods but there's a bunch more in the torch.Tensor documentation, I'd

```

Membentuk Kembali, menumpuk, meremas, dan melepaskan

Beberapa metode yang popular :

- `Torch.reshape(input,shape)` : dibentuk ulang input menjadi shape
- `Tensor.view(shape)` : Mengembalikan tampilan tensor asli dalam data berbeda shape
- `Torch.stack(tensors, dim=0)` : Menggabungkan barisan tensors sepanjang dimensi baru (dim).

Model pembelajaran mendalam (jaringan saraf) adalah tentang memanipulasi tensor dengan cara tertentu. Dan karena aturan perkalian matriks, jika ada ketidakcocokan bentuk, kita akan mengalami kesalahan.

```
[53] # Create a tensor
import torch
x = torch.arange(1., 8.)
x, x.shape
(tensor([1., 2., 3., 4., 5., 6., 7.]), torch.Size([7]))  
  
Now let's add an extra dimension with torch.reshape().  
  
[54] # 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]))  
  
We can also change the view with torch.view().  
  
[55] # 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]))  
  
Remember though, changing the view of a tensor with torch.view() really only creates a new view of the same tensor.  
So changing the view changes the original tensor too.  
  
[56] # Changing z changes x
z[:, 0] = 5
```

Kode dimulai dengan membuat tensor **x** menggunakan fungsi **torch.arange(1., 8.)**, yang menghasilkan tensor dengan nilai dari 1 hingga 7. Selanjutnya, kode menunjukkan cara menambahkan dimensi tambahan pada tensor menggunakan metode **reshape** dan **view**. **x_reshaped** dan **z** adalah dua tensor yang dihasilkan setelah manipulasi dimensi, dan **z** menunjukkan bahwa perubahan pada tensor **z** juga memengaruhi tensor aslinya, **x**.

```

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

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

  If we wanted to stack our new tensor on top of itself five times, we could do so with torch.stack().

  [57] # 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.]]])

  How about removing all single dimensions from a tensor?
  To do so you can use torch.squeeze() (I remember this as squeezing the tensor to only have dimensions over 1).

  [58] print(f"Previous tensor: {x_reshaped}")
  print(f"Previous shape: {x_reshaped.shape}")

  # Remove extra dimension from x_reshaped
  x_squeezed = x_reshaped.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])

```

Selanjutnya, kode menggunakan **torch.stack** untuk menggabungkan tensor **x** secara berulang ke dalam satu tensor baru, **x_stacked**. Perubahan dimensi dapat dieksplorasi dengan mengubah nilai parameter **dim** pada fungsi **stack**.

```

  [59] 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])

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

  You can also rearrange the order of axes values with torch.permute(input, dims), where the input gets turned into a view with new dims.

  [60] # Create tensor with specific shape
  x_original = torch.randn(size=(224, 224, 3))

  # Permute the original tensor to rearrange the axis order
  x_permuted = x_original.permute(2, 0, 1) # shifts axis 0->1, 1->2, 2->0

  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])

  Note: Because permuting returns a view (shares the same data as the original), the values in the permuted tensor will be the same as the original tensor and if you change the values in the view, it will change the values of the original.

```

Kemudian, terdapat operasi untuk menghapus dimensi tambahan dari tensor menggunakan metode **squeeze**. Tensor yang telah diubah dimensinya disimpan dalam variabel **x_squeezed**. Selanjutnya, dimensi tambahan dapat ditambahkan kembali menggunakan metode **unsqueeze**, dan tensor yang dihasilkan disimpan dalam variabel **x_unsqueezed**.

```

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

{x}
  ## 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])

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

You can also rearrange the order of axes values with torch.permute(input, dims), where the input gets turned into a view with new dims.

{60}
  # Create tensor with specific shape
  x_original = torch.rand(size=(224, 224, 3))

  # Permute the original tensor to rearrange the axis order
  x_permuted = x_original.permute(2, 0, 1) # shifts axis 0->1, 1->2, 2->0

  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])

Note: Because permuting returns a view (shares the same data as the original), the values in the permuted tensor will be the same as the original tensor and if you change the values in the view, it will change the values of the original.

```

Selanjutnya, kode membuat tensor dengan bentuk tertentu menggunakan `torch.rand(size=(224, 224, 3))`. Tensor tersebut kemudian diubah urutan sumbu menggunakan fungsi `permute` untuk menghasilkan tensor `x_permuted` dengan sumbu yang diatur ulang

Pengindexan (memilih data dari tensor)

```

  [61] # 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])

Indexing values goes outer dimension -> inner dimension (check out the square brackets).

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

You can also use : to specify 'all values in this dimension' and then use a comma (,) to add another dimension.

{63} # Get all values of 0th dimension and the 0 index of 1st dimension
  x[:, 0]

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

{64} # Get all values of 0th & 1st dimensions but only index 1 of 2nd dimension

```

Pertama, sebuah tensor `x` dibuat dengan menggunakan `torch.arange(1, 10).reshape(1, 3, 3)`. Tensor ini memiliki bentuk (shape) [1, 3, 3]. Selanjutnya, dilakukan berbagai operasi pengindeksan untuk memahami konsep tersebut.

Dalam contoh pengindeksan, kode menunjukkan cara menggunakan kurung siku bertingkat untuk mengakses elemen tensor pada dimensi yang berbeda. Misalnya, `x[0]` memberikan semua nilai pada dimensi pertama (indeks 0), `x[0][0]` memberikan nilai pada dimensi pertama dan kedua (indeks 0), dan `x[0][0][0]` memberikan nilai pada semua dimensi.

```
✓  [64] # Get all values of 0th & 1st dimensions but only index 1 of 2nd dimension
      x[:, :, 1]

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

✓  [65] # Get all values of the 0 dimension but only the 1 index value of the 1st and 2nd dimension
      x[:, 1, 1]

      tensor([5])

✓  [66] # Get index 0 of 0th and 1st dimension and all values of 2nd dimension
      x[0, 0, :] # same as x[0][0]

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

Selanjutnya, dilakukan beberapa operasi pengindeksan yang lebih kompleks, seperti `x[:, 0]` untuk mendapatkan semua nilai pada dimensi pertama dan hanya indeks 0 pada dimensi kedua, `x[:, :, 1]` untuk mendapatkan semua nilai pada dimensi pertama dan kedua, namun hanya indeks 1 pada dimensi ketiga, dan sebagainya.

Pengindeksan dapat terasa membingungkan, terutama pada tensor yang lebih besar, namun dengan latihan dan memvisualisasikannya, pemahaman terhadap konsep ini akan meningkat. Motto "visualize, visualize, visualize" menggarisbawahi pentingnya menggambarkan atau memvisualisasikan data untuk membantu memahami operasi pengindeksan pada tensor PyTorch.

Tensor PyTorch dan NumPy

Dua metode utama yang digunakan untuk mengonversi antara keduanya adalah `torch.from_numpy(ndarray)` untuk mengubah NumPy array menjadi PyTorch tensor, dan `torch.Tensor.numpy()` untuk mengubah PyTorch tensor menjadi NumPy array.

```
[68] # NumPy array to tensor
import torch
import numpy as np
array = np.arange(1.0, 8.0)
tensor = torch.from_numpy(array)
array, tensor

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

Note: By default, NumPy arrays are created with the datatype float64 and if you convert it to a PyTorch tensor, it'll keep the same datatype (as above).

However, many PyTorch calculations default to using float32.

So if you want to convert your NumPy array (float64) -> PyTorch tensor (float64) -> PyTorch tensor (float32), you can use tensor = torch.from_numpy(array).type(torch.float32).

Because we reassigned tensor above, if you change the tensor, the array stays the same.

[69] # Change the array, keep the tensor
array = array + 1
array, tensor

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

And if you want to go from PyTorch tensor to NumPy array, you can call tensor.numpy().

[70] # Tensor to NumPy array
tensor = torch.ones(7) # create a tensor of ones with dtype=float32
```

Dalam contoh pertama, kita membuat sebuah NumPy array menggunakan **np.arange(1.0, 8.0)** dan kemudian mengonversinya menjadi PyTorch tensor menggunakan **torch.from_numpy(array)**. Perlu diperhatikan bahwa default dtype dari NumPy array akan dijaga saat dikonversi menjadi PyTorch tensor. Namun, PyTorch umumnya menggunakan float32 sebagai default dtype. Jika Anda ingin mengonversi NumPy array dengan float64 ke PyTorch tensor dengan float32, Anda dapat menggunakan **tensor = torch.from_numpy(array).type(torch.float32)**

```
[70] # Tensor to NumPy array
tensor = torch.ones(7) # create a tensor of ones with dtype=float32
numpy_tensor = tensor.numpy() # will be dtype=float32 unless changed
tensor, numpy_tensor

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

And the same rule applies as above, if you change the original tensor, the new numpy_tensor stays the same.

[71] # Change the tensor, keep the array the same
tensor = tensor + 1
tensor, numpy_tensor

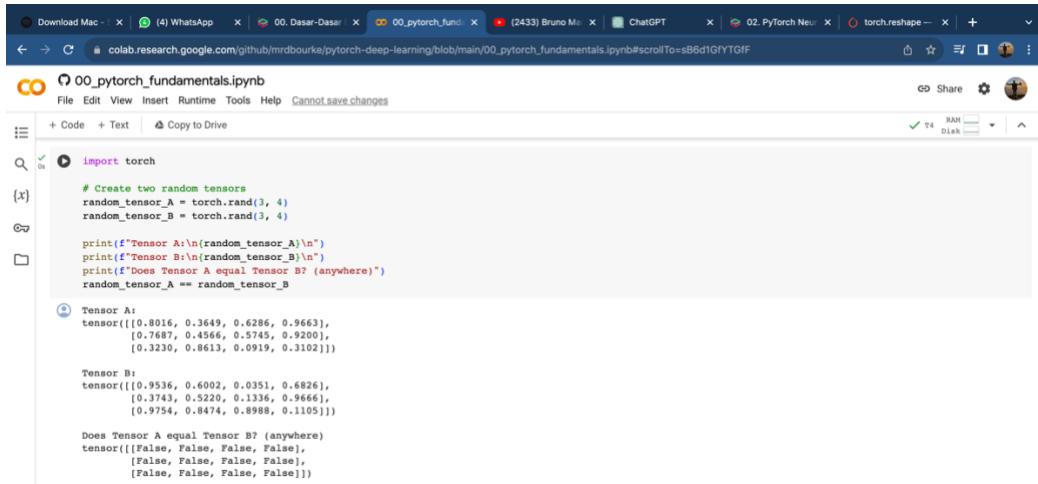
(tensor([2., 2., 2., 2., 2., 2., 2.]),
 array([1., 1., 1., 1., 1., 1., 1.], dtype=float32))
```

Selanjutnya, kode menunjukkan bahwa perubahan pada array NumPy juga memengaruhi tensor PyTorch yang telah dihasilkan sebelumnya.

Kemudian, dijelaskan cara mengubah PyTorch tensor menjadi NumPy array menggunakan metode **tensor.numpy()**. Seperti sebelumnya, perubahan pada tensor PyTorch juga akan mempengaruhi NumPy array yang dihasilkan.

Reproduksibilitas (mencoba mengambil yang acak dari yang acak)

Sebuah komputer pada dasarnya bersifat deterministik (setiap langkah dapat diprediksi) sehingga keacakan yang dihasilkannya adalah keacakan. Pentingnya reproduksibilitas dalam eksperimen ilmiah komputer, khususnya dalam konteks pembelajaran mesin dan jaringan saraf. Reproduksibilitas memungkinkan hasil eksperimen untuk diulang dan diverifikasi, yang sangat penting dalam penelitian.



```
# Import torch
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)\n")
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, 0.9988, 0.1105]])

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

Pada awalnya, kode tersebut menunjukkan bagaimana membuat dua tensor acak di PyTorch menggunakan **torch.rand()**. Seperti yang diharapkan, kedua tensor tersebut memiliki nilai yang berbeda karena keacakan. Namun, dalam banyak kasus, Anda ingin hasil yang dapat diprediksi dan dapat diulangi ketika menggunakan fungsi acak. Untuk ini, PyTorch menyediakan fungsi **torch.manual_seed()**. Fungsi ini menetapkan benih (seed) untuk generator angka acak, sehingga setiap kali program dijalankan dengan benih yang sama, ia menghasilkan urutan angka acak yang sama.

The screenshot shows a Google Colab notebook titled '00_pytorch_fundamentals.ipynb'. The code cell contains Python code demonstrating tensor operations and random number generation. It sets a random seed, creates tensors C and D, prints their values, and checks if they are equal. The output shows that with a fixed seed, both tensors have identical values.

```
import torch
import random

# Set the random seed
RANDOM_SEED=42 # try changing this to different values and see what happens 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(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 contoh di atas, tensor C dan D akan memiliki nilai yang sama karena benih yang digunakan identik. Ini menunjukkan bagaimana menggunakan `torch.manual_seed()` untuk menciptakan kondisi yang reproduksibel dalam eksperimen yang melibatkan keacakan. Menggunakan benih yang sama memastikan bahwa setiap panggilan ke fungsi acak seperti `torch.rand()` akan menghasilkan hasil yang sama, sehingga memudahkan proses verifikasi dan validasi hasil eksperimental.

Menjalankan Tensor pada GPU (dan membuat komputasi lebih cepat)

Algoritma pembelajaran mendalam memerlukan banyak operasi numerik dan secara default operasi ini sering dilakukan pada CPU. Namun, ada perangkat keras umum lainnya yang disebut GPU (unit pemrosesan grafis), yang seringkali jauh lebih cepat dalam melakukan jenis operasi tertentu yang dibutuhkan jaringan saraf (perkalian matriks) dibandingkan CPU.

1. Mendapatkan GPU

Untuk mendapatkan GPU banyak sekali cara untuk mengaksesnya sepeeti dengan menggunakan google colab, menggunakan platform komputasi awan (AWS,GCP, dan Azure).

2. Menjalankan PyTorch di GPU

Setelah GPU bisa untuk diakses, selanjutnya menggunakan PyTorch untuk menyimpan data(tensor) dan menghitung data (melakukan operasi pada tensor). Untuk melakukannya

menggunakan `torch.cuda` paket. Kita dapat menguji apakah PyTorch sudah memiliki akses ke GPU menggunakan fungsi `torch.cuda.is_available()`. Kita dapat menulis kode yang agnostik perangkat, artinya kode tersebut akan berjalan baik pada CPU maupun GPU, tergantung pada ketersediaan. Untuk ini, buat variabel `device` yang secara otomatis memilih 'cuda' (GPU) jika tersedia, atau 'cpu' jika tidak. 'cuda' adalah antarmuka yang PyTorch gunakan untuk berinteraksi dengan CUDA (Compute Unified Device Architecture) NVIDIA, yang memungkinkan komputasi paralel pada GPU. Ketika kita, memiliki beberapa GPU, Anda bisa mendapatkan jumlah GPU yang tersedia menggunakan `torch.cuda.device_count()`. Ini berguna untuk beban kerja yang memerlukan distribusi tugas di antara beberapa GPU atau untuk paralelisasi data.

3. Menempatkan tensor (dan model) pada GPU

dengan menggunakan metode `to(device)` pada tensor PyTorch. Hal ini dilakukan untuk memanfaatkan kemampuan komputasi yang lebih cepat yang ditawarkan oleh GPU dibandingkan dengan CPU. Pertama, sebuah tensor dibuat dengan nilai **[1, 2, 3]** dan secara default disimpan di CPU. Untuk memeriksa lokasi tensor, kita dapat menggunakan properti `tensor.device`. Outputnya akan menunjukkan bahwa tensor tersebut berada di CPU. Tensor kemudian dipindahkan ke GPU dengan memanggil `to(device)`. Jika GPU tersedia, tensor akan disalin ke GPU dan lokasinya akan menjadi GPU. Outputnya akan menunjukkan tensor di kedua perangkat, dengan tensor di GPU ditandai dengan `device='cuda:0'` (dengan asumsi GPU diindeks 0). Metode `to(device)` mengembalikan salinan tensor dan untuk menimpannya, Anda dapat menetapkan ulang seperti `tensor = tensor.to(device)`. Pemindahan tensor ke GPU berguna ketika melakukan operasi komputasi berat, terutama saat melibatkan model jaringan saraf yang besar.

4. Memindahkan tensor kembali ke CPU

Jika kita ingin memindahkan tensor kembali ke CPU gunakan fungsi `torch.Tensor.numpy()` metode di `tensor_on_gpu`. ebagai gantinya, untuk mengembalikan tensor ke CPU dan dapat digunakan dengan NumPy kita dapat menggunakan `Tensor.cpu()`. Ini menyalin tensor ke memori CPU sehingga dapat digunakan dengan CPU

Nama : Fasya Hanifah
NIM : 1103200149
KELAS : MACHINE LEARNING

RANGKUMAN MACHINE LEARNING CHAPTER 01 PyTorch Workflow Fundamentals

01. Dasar-Dasar Alur Kerja PyTorch

Dasar-dasar alur kerja dalam menggunakan PyTorch untuk pembelajaran mesin dan pembelajaran mendalam, dengan fokus khusus pada membangun model untuk mempelajari dan memprediksi pola garis lurus.

Tema dan Isi:

- **Mempersiapkan Data:** Ini melibatkan pengambilan dan persiapan data. Dalam kasus ini, data yang digunakan adalah garis lurus sederhana.
- **Membangun Model:** Ini mencakup pembuatan model yang akan mempelajari pola dalam data, termasuk pemilihan fungsi kerugian (loss function), pengoptimal (optimizer), dan pembangunan loop pelatihan.
- **Menyesuaikan Model dengan Data (Pelatihan):** Proses ini melibatkan pelatihan model untuk mengenali pola dalam data.
- **Membuat Prediksi dan Mengevaluasi Model (Inferensi):** Setelah model dilatih, tahap ini membandingkan hasil prediksi model dengan data aktual untuk mengevaluasi kinerjanya.
- **Menyimpan dan Memuat Model:** Bagian ini berfokus pada cara menyimpan dan memuat model untuk digunakan di masa mendatang.
- **Menyatukan Semuanya:** Tahap akhir ini adalah menggabungkan semua langkah di atas dalam sebuah proyek lengkap.

```
✓ 4 import torch
   from torch import nn # nn contains all of PyTorch's building blocks for neural networks
   import matplotlib.pyplot as plt

# Check PyTorch version
torch.__version__
↳ '2.1.0+cu121'
```

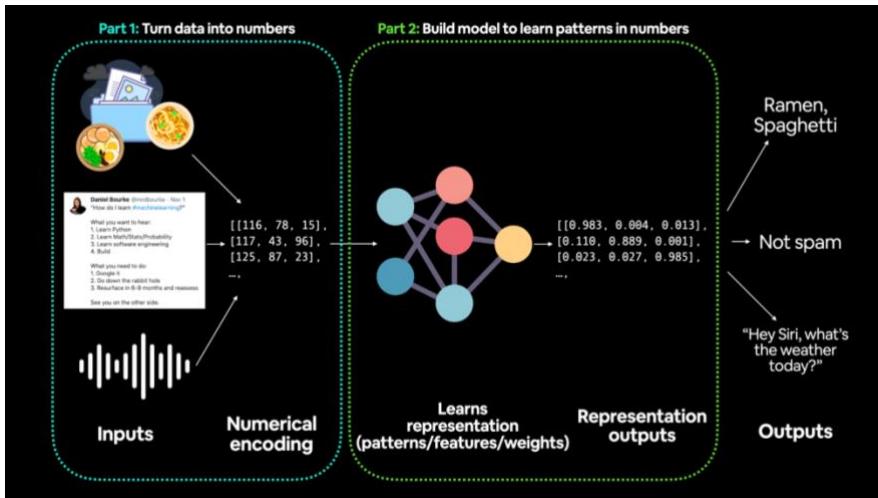
1. Impor Library yang Diperlukan:

- **torch:** Library utama PyTorch.
- **from torch import nn:** Impor modul **nn** dari PyTorch yang berisi blok bangunan untuk jaringan saraf.
- **import matplotlib.pyplot as plt:** Impor **matplotlib.pyplot** untuk visualisasi data dan hasil.

2. Cek Versi PyTorch:

Menjalankan **torch.__version__** untuk mengetahui versi PyTorch yang digunakan.

1. Data (Persiapan dan Pemuatan)



Kita kan membuat data sintetis yang mengikuti model garis lurus. Ini adalah pendekatan yang umum dalam pembelajaran mesin ketika data aktual tidak tersedia atau ingin menguji algoritma dengan data yang memiliki pola yang diketahui. Ini membantu dalam memahami bagaimana model belajar dan validasi keakuratan model. Konsep utama di sini adalah regresi linier, di mana menciptakan hubungan linier antara variabel input (X) dan output (y) dengan parameter yang diketahui (berat dan bias). Tujuan dari pembelajaran mesin dalam kasus ini adalah untuk memperkirakan parameter ini (berat dan bias) seakurat mungkin dari data.

```
# 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]
```

(`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]]))`

- Terdapat parameter **weight = 0.7**, Ini adalah kemiringan garis lurus. Dalam konteks regresi linier, ini adalah koefisien yang menggambarkan hubungan antara X dan y . **bias = 0.3**: Ini adalah intersep garis, yaitu nilai y ketika X adalah 0. Kita akan membuat data dengan menentukan rentang nilai X dari **start** ke **end** dengan **step**. Dalam kasus ini, Anda membuat nilai X dari 0 hingga 1 dengan interval 0.02.

- **X = torch.arange(start, end, step).unsqueeze(dim=1)**: Ini membuat tensor 1D dari nilai-nilai tersebut dan kemudian menambahkan dimensi tambahan untuk membuatnya menjadi tensor 2D (kolom). Ini penting karena PyTorch umumnya bekerja dengan data dalam bentuk batch atau matriks.
- **y = weight * X + bias**: Ini menghitung nilai y yang sesuai untuk setiap nilai X berdasarkan hubungan garis lurus. Ini adalah data target yang akan digunakan model untuk belajar.
- Melihat sampel data **X[:10], y[:10]**: Ini menampilkan 10 nilai pertama dari X dan y yang sesuai.

Pisahkan data menjadi set pelatihan dan pengujian

Membagi dataset menjadi set pelatihan dan pengujian, yang merupakan langkah umum machine learning untuk mengevaluasi seberapa baik model bekerja pada data yang tidak terlihat selama pelatihan.

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

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

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

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

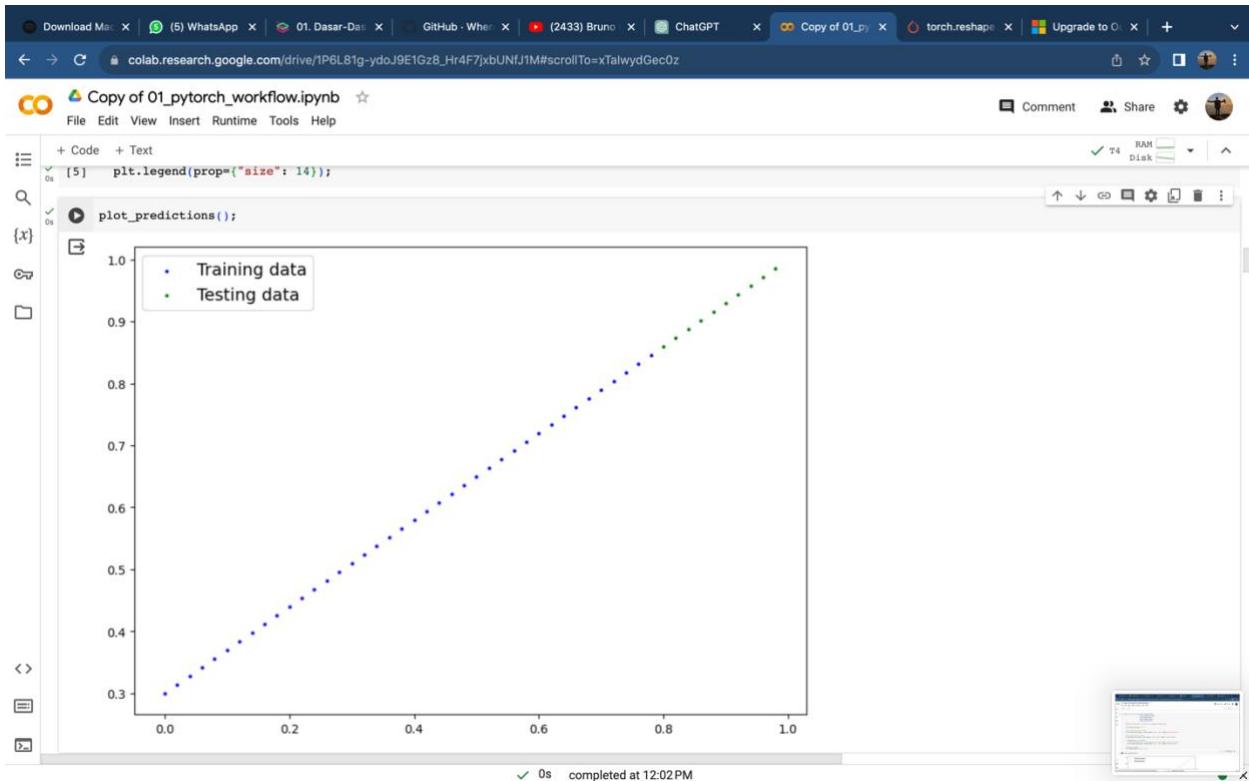
    # Show the legend
    plt.legend(prop={"size": 14});

plot_predictions();

```

- Membuat pembagian data **train_split = int(0.8 * len(X))**: Membuat pembagian data dengan mengambil 80% data untuk pelatihan dan 20% untuk pengujian. **len(X)** memberikan panjang total dataset, dan **int(0.8 * len(X))** memberikan indeks pemisahan antara data pelatihan dan pengujian.
- **X_train, y_train = X[:train_split], y[:train_split]**: Memisahkan data pelatihan untuk **X** dan **y** berdasarkan indeks yang dihitung sebelumnya.

- **X_test, y_test = X[train_split:], y[train_split]:** Memisahkan data pengujian untuk X dan y berdasarkan indeks yang dihitung sebelumnya.



plot_predictions(): Fungsi ini memvisualisasikan data pelatihan dan pengujian menggunakan matplotlib. Data pelatihan ditampilkan sebagai titik biru, data pengujian sebagai titik hijau.

2. Membangun Model

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

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

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

PyTorch memiliki empat modul yang dapat digunakan untuk membuat semua jenis jaringan saraf.

[torch.nn](#)

Berisi semua elemen penyusun grafik komputasi (pada dasarnya serangkaian komputasi yang dilakukan dengan cara tertentu).

torch.nn.Parameter	Menyimpan tensor yang dapat digunakan dengan nn.Module. Jika requires_grad=Truegradien (digunakan untuk memperbarui parameter model melalui penurunan gradien) dihitung secara otomatis, hal ini sering disebut sebagai "autograd".
torch.nn.Module	Kelas dasar untuk semua modul jaringan saraf, semua blok penyusun jaringan saraf adalah subkelas. Jika Anda membangun jaringan saraf di PyTorch, model Anda harus membuat subkelas nn.Module. Membutuhkan forward()metode untuk diimplementasikan.
torch.optim	Berisi berbagai algoritme pengoptimalan (ini memberi tahu parameter model yang disimpan dalam nn.Parametercara terbaik untuk mengubah guna meningkatkan penurunan gradien dan pada gilirannya mengurangi kerugian).
def forward()	Semua nn.Modulesubkelas memerlukan suatu forward()metode, ini mendefinisikan perhitungan yang akan dilakukan pada data yang diteruskan ke data tertentu nn.Module(misalnya rumus regresi linier di atas).

Memeriksa konten Model PyTorch

```
[ ] # Set manual seed since nn.Parameter are randomly initialized
torch.manual_seed(42)

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

# Check the nn.Parameter(s) within the nn.Module subclass we created
list(model_0.parameters())
[Parameter containing:
tensor([0.3367], requires_grad=True),
Parameter containing:
tensor([0.1288], requires_grad=True)]
```

We can also get the state (what the model contains) of the model using [.state_dict\(\)](#).

```
➊ # List named parameters
model_0.state_dict()
➋ OrderedDict([('weights', tensor([0.3367])), ('bias', tensor([0.1288]))])
```

Inisialisasi Model:

- Menciptakan sebuah instance dari model menggunakan kelas **LinearRegressionModel**

- Menampilkan daftar parameter (berupa `nn.Parameter`) yang ada dalam model menggunakan `model_0.parameters()`. Menampilkan status (nilai) dari model menggunakan `model_0.state_dict()`

Membuat Prediksi dengan `torch.inference_mode()`

```
✓ 0s ① # Make predictions with model
    with torch.inference_mode():
        y_preds = model_0(X_test)

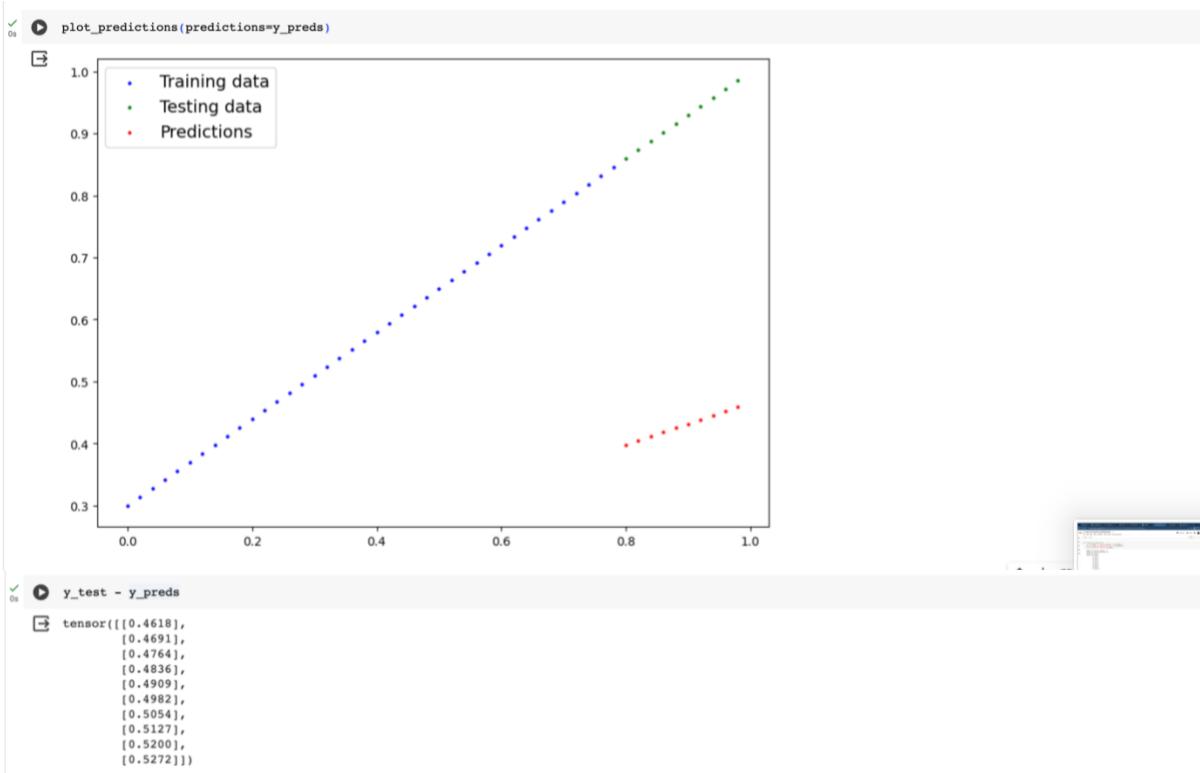
    # Note: in older PyTorch code you might also see torch.no_grad()
    # with torch.no_grad():
    #     y_preds = model_0(X_test)
```

Membuat prediksi dengan meneruskan data pengujian (`X_test`) melalui model menggunakan `torch.inference_mode()` (sekarang disarankan daripada `torch.no_grad()`).

```
✓ [11] # Check the predictions
0s 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]])
```

Menampilkan jumlah sampel pengujian dan jumlah prediksi yang dibuat. Terdapat 10 testing sample dan 10 prediction sample.



```

✓ 0s y_test - y_preds

```

y_test	y_preds
0.4618	0.4691
0.4764	0.4836
0.4836	0.4909
0.4909	0.4982
0.4982	0.5054
0.5054	0.5127
0.5127	0.5200
0.5200	0.5272

Output prediksi ditampilkan, plot grafik menunjukkan bahwa warna biru merupakan data training, warna hijau merupakan data testing, dan merah adalah prediksi. Pada akhirnya, perbedaan antara nilai sebenarnya (**y_test**) dan prediksi (**y_preds**) diperiksa untuk melihat seberapa baik model melakukan prediksi.

Train Model

```

✓ [54] # Create the loss function
loss_fn = nn.L1Loss() # MAE loss is same as L1Loss

# Create the optimizer
optimizer = torch.optim.SGD(params=model_0.parameters(), # parameters of target model to optimize
                           lr=0.01) # learning rate (how much the optimizer should change parameters at each step, higher=more (less stable), lower=less (more stable))

```

Membuat Fungsi Loss dan Optimizer:

- `loss_fn = nn.L1Loss()`: Ini mendefinisikan fungsi loss (kesalahan) yang digunakan, yaitu Mean Absolute Error (MAE), juga dikenal sebagai L1Loss.
- `optimizer = torch.optim.SGD(...)`: Menginisialisasi optimizer (Stochastic Gradient Descent) dengan parameter dari model (`model_0.parameters()`) dan learning rate 0.01.

```

✓ 0s .orch.manual_seed(42)

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

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

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

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

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

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

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

    # 4. Loss backwards
    loss.backward()

    # 5. Progress the optimizer
    optimizer.step()

    ### Testing

```

1. Mengatur seed dan epochs

- **torch.manual_seed(42)**: Menetapkan seed untuk mengontrol randomness, sehingga hasilnya dapat direproduksi.
- **epochs = 100**: Jumlah kali model akan melalui data pelatihan selama pelatihan.

2. Loop Pelatihan (**for epoch in range(epochs)**):

- Menggunakan loop untuk mengiterasi sebanyak epochs yang telah ditentukan.
- Dalam setiap epoch, model akan melewati data pelatihan dan diuji menggunakan data pengujian.

3. Pelatihan:

- **model_0.train()**: Mengatur model ke mode pelatihan. Ini penting karena beberapa lapisan (seperti dropout) berperilaku berbeda saat pelatihan dan evaluasi.
- **y_pred = model_0(X_train)**: Melakukan forward pass pada data pelatihan, menghasilkan prediksi.
- **loss = loss_fn(y_pred, y_train)**: Menghitung loss antara prediksi (**y_pred**) dan nilai sebenarnya (**y_train**).
- **optimizer.zero_grad()**: Mengatur gradien parameter model ke nol sebelum melakukan backpropagation.
- **loss.backward()**: Melakukan backpropagation, menghitung gradien loss terhadap parameter model.
- **optimizer.step()**: Melakukan langkah optimisasi untuk memperbarui parameter model berdasarkan gradien yang dihitung.

```

0s # Put the model in evaluation mode
model_0.eval()

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

    # 2. Calculate loss on test data
    test_loss = loss_fn(test_pred, y_test.type(torch.float)) # predictions come in torch.float datatype, so comparisons need to be done with tensors of the same type

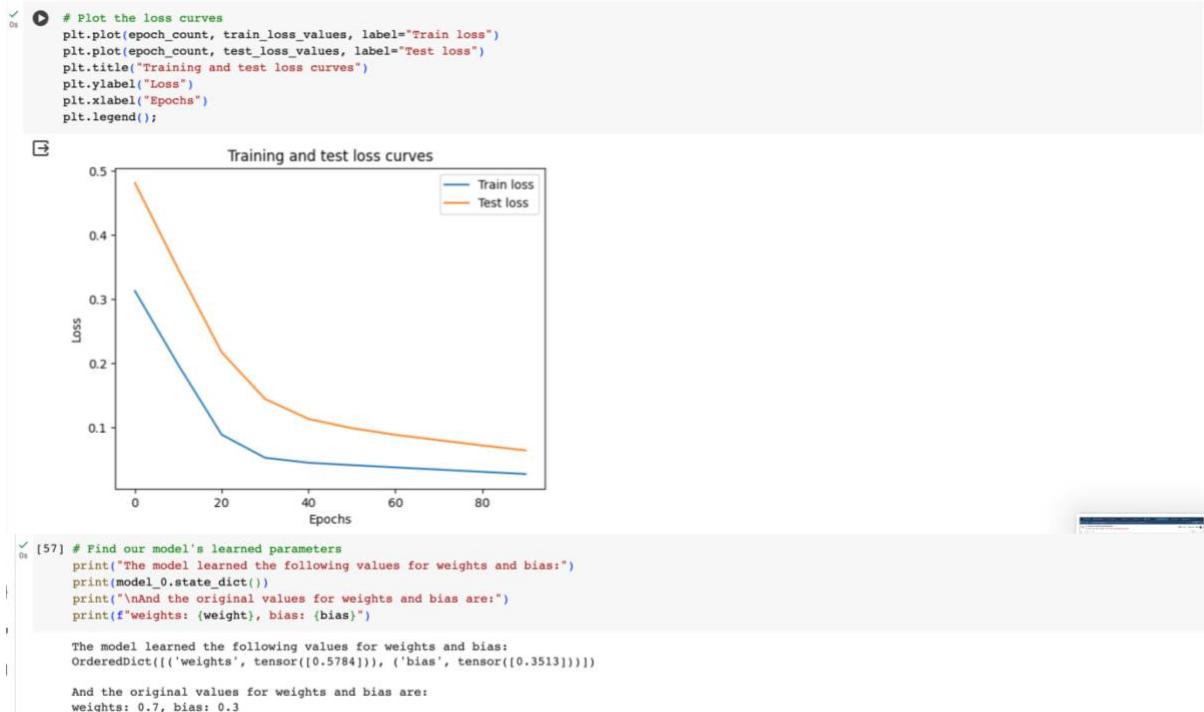
    # Print out what's happening
    if epoch % 10 == 0:
        epoch_count.append(epoch)
        train_loss_values.append(loss.detach().numpy())
        test_loss_values.append(test_loss.detach().numpy())
        print(f"Epoch: {epoch} | MAE Train Loss: {loss} | MAE Test Loss: {test_loss}")

```

Epoch: 0 | MAE Train Loss: 0.31288118031959534 | 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: 0.04543799554207802 | 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

Evaluasi (Testing):

- **model_0.eval():** Mengatur model ke mode evaluasi. Ini mematikan dropout atau layer lain yang dapat mempengaruhi hasil inferensi.
- **with torch.inference_mode():** Menggunakan **torch.inference_mode()** sebagai pengelola konteks untuk inferensi (membuat prediksi).
- **test_pred = model_0(X_test):** Melakukan forward pass pada data pengujian, menghasilkan prediksi untuk evaluasi.
- **test_loss = loss_fn(test_pred, y_test.type(torch.float)):** Menghitung loss pada data pengujian.
- Menyimpan nilai loss pada interval tertentu (**if epoch % 10 == 0**) untuk pemantauan dan mencetaknya.



Grafik Kurva Loss:

- plt.plot(epoch_count, train_loss_values, label="Train loss"): Menambahkan kurva loss pada data pelatihan ke plot.
- plt.plot(epoch_count, test_loss_values, label="Test loss"): Menambahkan kurva loss pada data pengujian ke plot.
- plt.title("Training and test loss curves"): Menetapkan judul plot.
- plt.ylabel("Loss"): Menetapkan label sumbu y.
- plt.xlabel("Epochs"): Menetapkan label sumbu x.
- plt.legend(): Menampilkan legenda yang menjelaskan kurva mana yang mewakili loss pelatihan dan pengujian.
- Terlihat pada grafik garis kuning merupakan test loss dan garis biru merupakan training loss. Grafik ini menunjukkan penurunan kerugian seiring berjalannya waktu. Kerugian adalah ukuran seberapa salah model yang anda buat.

Menampilkan Parameter Model:

- print("The model learned the following values for weights and bias:"): Mencetak teks informatif.
- print(model_0.state_dict()): Mencetak nilai parameter model (bobot dan bias) setelah pelatihan.
- print("\nAnd the original values for weights and bias are:"): Mencetak teks informatif.
- print(f"weights: {weight}, bias: {bias}"): Mencetak nilai parameter model (bobot dan bias) yang sebenarnya (diasumsikan untuk membuat data) sebelum pelatihan dimulai.
- Didapatkan nilai bobotnya adalah 0,7 dan bias : 0,3

4. Membuat Prediksi dengan model PyTorch terlatih (inferensi)

```

✓ 0 # 1. Set the model in evaluation mode
model_0.eval()

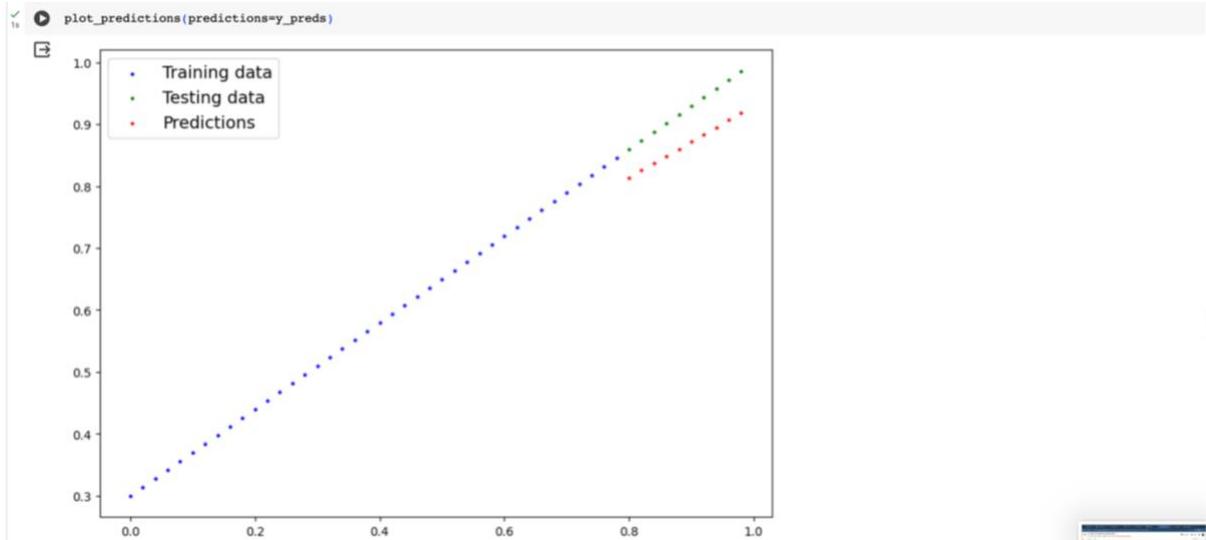
# 2. Setup the inference mode context manager
with torch.inference_mode():
    # 3. Make sure the calculations are done with the model and data on the same device
    # in our case, we haven't setup device-agnostic code yet so our data and model are
    # on the CPU by default.
    # model_0.to(device)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

y_preds

```

Berfokus pada evaluasi model menggunakan mode inferensi (`torch.inference_mode()`).

`model_0.eval()` Mengubah mode model ke mode evaluasi. Ini mematikan operasi yang tidak diperlukan selama evaluasi, seperti pelacakan gradien, yang hanya dibutuhkan selama pelatihan. `with torch.inference_mode():` Menggunakan pengelola konteks `torch.inference_mode()` untuk mengeksekusi blok kode tertentu dalam mode inferensi. Dalam hal ini, membuat prediksi pada data uji. `y_preds = model_0(X_test)` Meneruskan data uji (`X_test`) ke model (`model_0`) untuk membuat prediksi. Ini dilakukan dalam mode inferensi yang mematikan beberapa fitur yang tidak diperlukan untuk prediksi, membuatnya lebih efisien.



Terlihat pada grafik plot muncul titik merah dan titik tersebut lebih dekat dibandingkan grafik sebelumnya.

5. Menyimpan dan memuat model PyTorch

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

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

<All keys matched successfully>

Excellent! It looks like things matched up.

Now to test our loaded model, let's perform inference with it (make predictions) on the test data.

Remember the rules for performing inference with PyTorch models?

If not, here's a refresher:
▶ PyTorch inference rules

[63] # 1. Put the loaded model into evaluation mode
loaded_model_0.eval()

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

Now we've made some predictions with the loaded model, let's see if they're the same as the previous predictions.

[64] # Compare previous model predictions with loaded model predictions (these should be the same)
y_preds == loaded_model_preds

tensor([True],
      [True],
```

1. Membuat Instance model baru : Ini membuat instance baru dari **LinearRegressionModel**. Model ini pada awalnya akan memiliki parameter (weights dan biases) yang diinisialisasi secara acak.

2. Menempatkan model yang dimuat dalam mode evaluasi : Memanggil **.eval()** pada model mengatur model ke mode evaluasi. Ini penting untuk beberapa jenis layer (seperti dropout dan batch normalization) yang memiliki perilaku berbeda selama pelatihan dan inferensi. Walaupun dalam kasus model regresi linier sederhana ini mungkin tidak memiliki efek, ini merupakan praktik yang baik.

3. Membuat prediksi dengan model yang dimuat : Di sini, kita menggunakan **`torch.inference_mode()`** untuk membuat prediksi dengan model yang dimuat menggunakan data **`X_test`**. Ini adalah cara efisien untuk membuat prediksi karena tidak menghitung gradient, yang tidak diperlukan untuk inferensi.

```
✓ [64] # Compare previous model predictions with loaded model predictions (these should be the same)
y_preds == loaded_model_preds
tensor([True],
      [True],
      [True],
      [True],
      [True],
      [True],
      [True],
      [True],
      [True])
```

Langkah terakhir adalah membandingkan prediksi yang dibuat oleh model asli (**`y_preds`**) dengan prediksi yang dibuat oleh model yang dimuat (**`loaded_model_preds`**). Jika semua langkah di atas dijalankan dengan benar, kedua set prediksi ini harus sama, menunjukkan bahwa model yang disimpan dan dimuat kembali berhasil mempertahankan informasi yang diperlukan untuk membuat prediksi yang sama.

Nama : Fasya Hanifah
NIM : 1103200149

RESUME 02 PyTorch Neural Network Classification

0. Architecture of a classification neural network

The screenshot shows a Jupyter Notebook interface with the following code:

```
from sklearn.datasets import make_circles

# Make 1000 samples
n_samples = 1000

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

print("First 5 X features:\n", X[:5])
print("First 5 y labels:\n", y[:5])

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

The output of the code is displayed below the code cell:

```
First 5 X features:
[[ 0.75424625  0.23148074]
 [-0.75615988  0.15325988]
 [-0.81539193  0.17328203]
 [-0.39373073  0.69286277]
 [ 0.44220765 -0.89672343]]

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

   X1      X2  label
0  0.75  0.2314  1
1 -0.75  0.1533  1
2 -0.81  0.1733  1
3 -0.39  0.6929  1
4  0.44  -0.8967  0
```

- Menggunakan library Scikit-learn untuk menghasilkan dataset sintetis dengan fungsi **make_circles**. Fungsi ini membuat 1000 sampel (**n_samples=1000**) yang terdiri dari titik-titik yang terorganisir dalam dua lingkaran konsentris. Parameter **noise=0.03** menambahkan sedikit variasi acak ke lokasi titik-titik tersebut, membuatnya tidak terlalu teratur, sedangkan **random_state=42** memastikan bahwa setiap kali kode dijalankan, kami mendapatkan set data yang sama. Tujuan dari menghasilkan data ini adalah untuk menciptakan sebuah masalah klasifikasi sederhana yang sering digunakan untuk menguji algoritma pembelajaran mesin.
- Menampilkan lima contoh pertama fitur (**X**) dan label terkait (**y**) dari kumpulan data yang dihasilkan menggunakan **make_circles** fungsi tersebut. Mengeksekusi kode ini akan membantu Anda memahami struktur data Anda. **X[:5]** akan menampilkan lima set fitur pertama (setiap set fitur kemungkinan besar merupakan sepasang koordinat dalam hal ini, karena **make_circles** menghasilkan data melingkar 2D), dan **y[:5]** akan menampilkan lima label pertama, yang menunjukkan yang mana dari dua kemungkinan lingkaran (kelas) yang masing-masingnya set fitur milik.

	x1	x2	label
0	0.754246	0.231481	1
1	-0.756159	0.153259	1
2	-0.815392	0.173282	1
3	-0.393731	0.692863	1
4	0.442208	-0.896723	0
5	-0.479646	0.676435	1
6	-0.013648	0.803349	1
7	0.771513	0.147760	1
8	-0.169322	-0.793456	1
9	-0.121486	1.021509	0

```

[4]: # Check different labels
circles.label.value_counts()

1    500
0    500
Name: label, dtype: int64

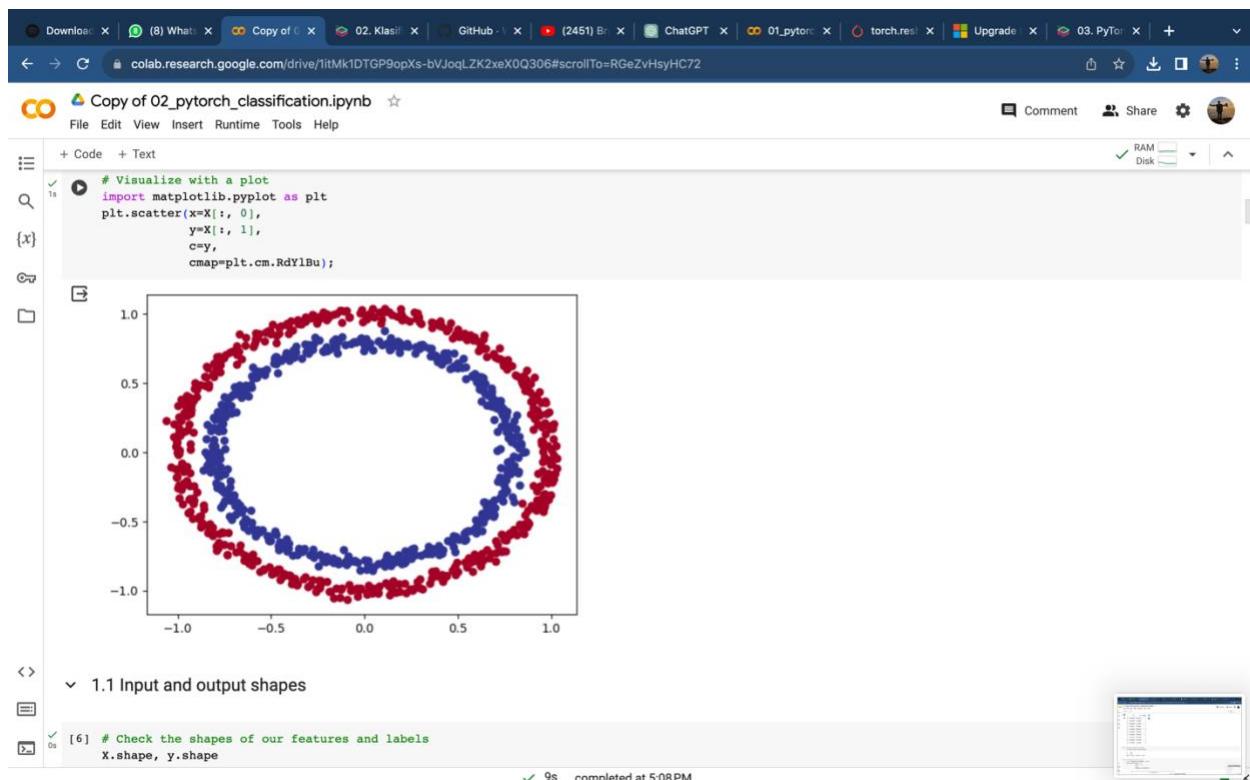
[5]: # Visualize with a plot
import matplotlib.pyplot as plt
plt.scatter(x=X[:, 0],
            y=X[:, 1],
            c=y,
            cmap=plt.cm.RdYlBu);

```

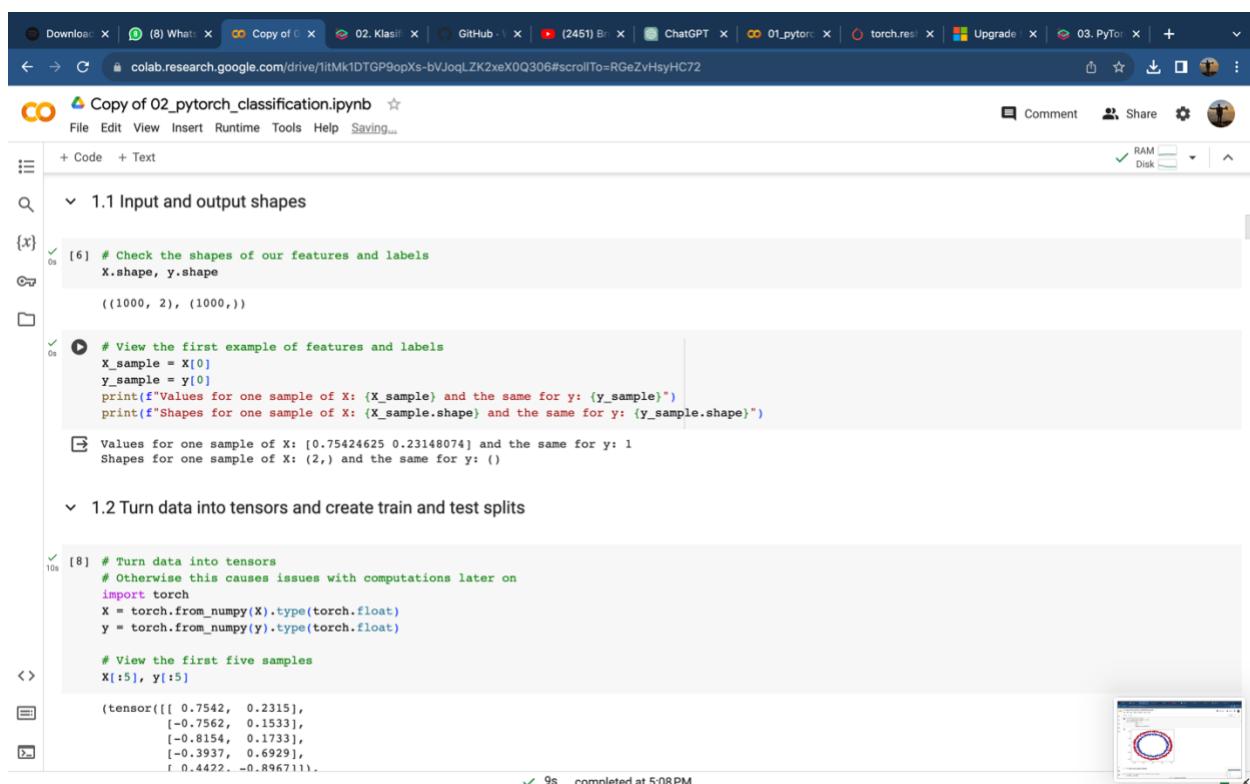
Membuat DataFrame menggunakan Pandas, pustaka manipulasi data populer dengan Python. DataFrame ini, bernama **circles**, disusun untuk menyertakan data yang dihasilkan oleh **make_circles** fungsi. Secara khusus, ini memiliki tiga kolom:

1. "X1": Kolom ini mewakili fitur pertama dari setiap sampel di **X**. Hal ini diperoleh dengan mengiris semua baris (:) dan kolom pertama (0) dari **X**(yaitu, **X[:, 0]**).
2. "X2": Ini adalah fitur kedua dari setiap sampel di **X**. Mirip dengan "X1", ini diturunkan dengan memotong semua baris dan kolom kedua **X**(yaitu, **X[:, 1]**).
3. "label": Kolom ini berisi label **y** dari kumpulan data, yang menunjukkan kelas milik setiap sampel.

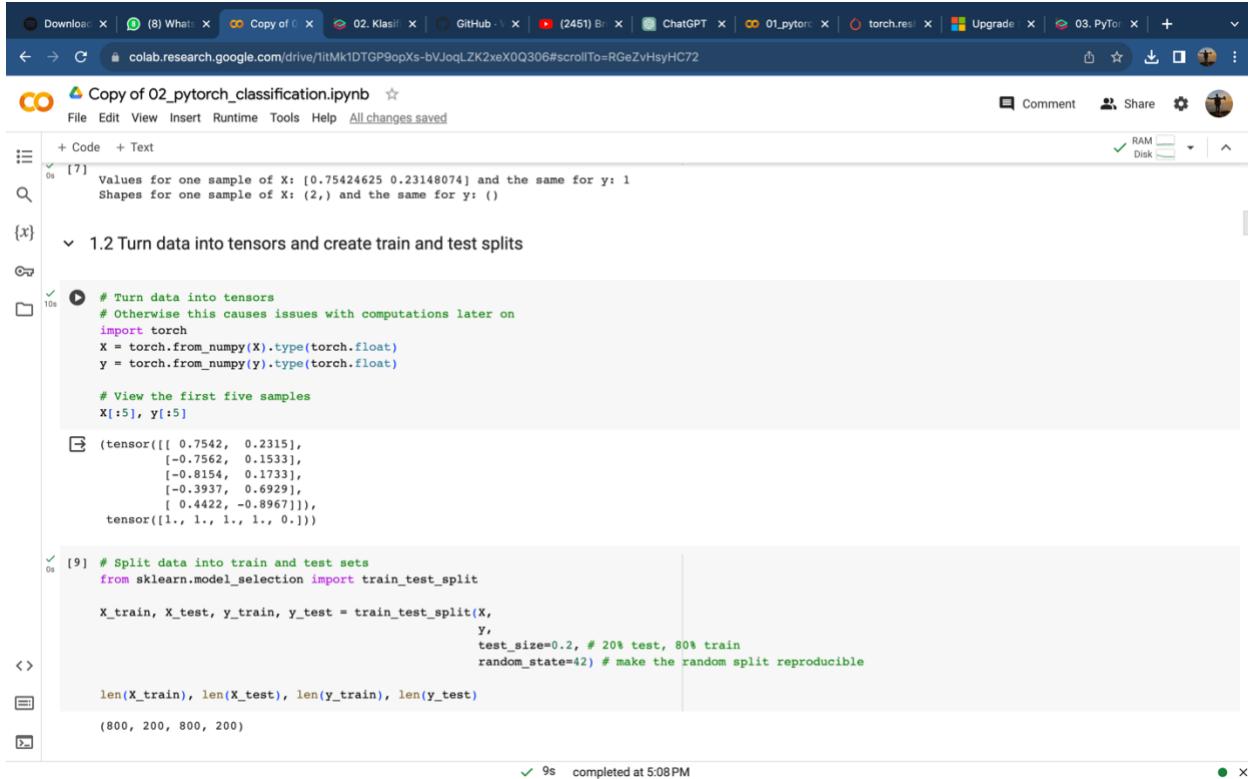
Terakhir, **circles.head(10)** tampilkan 10 baris pertama DataFrame ini. Hal ini berguna untuk mendapatkan gambaran singkat tentang data, menampilkan dua fitur pertama dan label yang sesuai untuk 10 sampel pertama.



Visualisasi menggunakan plot, terlihat pada grafik terdapat warna biru dan merah.



Lakukan pengecekan untuk ukuran feature dan labelnya. Disini dihasilkan nilai dari 1 sample x adalah [0.75424625 0.2318074].



The screenshot shows a Google Colab notebook titled "Copy of 02_pytorch_classification.ipynb". The code cell at line 7 displays the values for one sample of X and y, both being arrays of shape (2,). The code cell at line 10s shows the creation of tensors from NumPy arrays X and y, and then displays the first five samples of the tensor X. The code cell at line 9s splits the data into training and testing sets using train_test_split, resulting in four tensors: X_train, X_test, y_train, and y_test, all of size (800, 200).

```
[7] 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: ()

{x}
    1.2 Turn data into tensors and create train and test splits

[10s]
    # Turn data into tensors
    # Otherwise this causes issues with computations later on
    import torch
    X = torch.from_numpy(X).type(torch.float)
    y = torch.from_numpy(y).type(torch.float)

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

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

[9s]
    [9] # Split data into train and test sets
    from sklearn.model_selection import train_test_split

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

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

    (800, 200, 800, 200)
```

Ubah data menjadi tensor dan lihat 5 sample pertama disini dihasilkan isi dari tensor adalah [1, 1, 1, 1, 0]. Lalu bagi data menjadi data set pelatihan dan pengujian. Gunakan fungsi train_test_split().

2. Membangun Model

Copy of 02_pytorch_classification.ipynb

```
[70] # Standard PyTorch imports
import torch
from torch import nn

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

'cpu'

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

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

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

0s completed at 5:44PM

Copy of 02_pytorch_classification.ipynb

```
[72] # Replicate CircleModelV0 with nn.Sequential
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)
)

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

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

First 10 predictions:
tensor([-0.0338,
       [-0.0309],
       [ 0.0894],
       [-0.0692],
       [ 0.2967],
       [ 0.2968],
       [ 0.1405],
       [ 0.2178],
       [ 0.0805],
       [-0.0284]], grad_fn=<SliceBackward0>)

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

0s completed at 5:44PM

```

Copy of 02_pytorch_classification.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Q [74] # Create a loss function
# loss_fn = nn.BCELoss() # BCELoss = no sigmoid built-in
loss_fn = nn.BCEWithLogitsLoss() # BCEWithLogitsLoss = sigmoid built-in

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

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

3. Train model
+ Code + Text
3.1 Going from raw model outputs to predicted labels (logits -> prediction probabilities -> prediction labels)

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

tensor([[-0.0338],
       [-0.0309],
       [ 0.0894],
       [-0.0692],
       [ 0.2967]], grad_fn=<SliceBackward0>)

[77] # Use sigmoid on model logits
y_pred_probs = torch.sigmoid(y_logits)

```

Analisis :

- Impor PyTorch dan torch.nn menyiapkan kode agnostic
- Buat kelas model menggunakan subkelas nn.Module, alli buat 2 nn.Linear lapisan dalam konstruktor, definisikan fungsi forward() dan buat instance kelas model.
- Lalu replikasi CircleModelV0 dengan nn.Sequential. nn.Sequential jauh lebih sedergana daripada subkelas nn.Module.
- Lalu buat prediksi dengan model, didaoatkan outout panjang prediksi 200 dan panjang sample uji 200
- PyTorch memiliki dua implementasi entropi silang biner:
[torch.nn.BCELoss\(\)](#)- Membuat fungsi kerugian yang mengukur entropi silang biner antara target (label) dan input (fitur).
[torch.nn.BCEWithLogitsLoss\(\)](#)- Ini sama seperti di atas, hanya saja ia memiliki lapisan sigmoid ([nn.Sigmoid](#)) bawaan (kita akan segera melihat artinya).
- Terakhir buat marik evakuasi , matriks evaluasi dibuat untuk mengevaluasi model apakah sudah baik atau belum.

3. Train Model

Copy of 02_pytorch_classification.ipynb

```

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

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

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

```

3. Train model

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

```

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

tensor([[-0.0338],
       [-0.0309],
       [ 0.0894],
       [-0.0692],
       [ 0.2967]], grad_fn=<SliceBackward0>)

[77] # Use sigmoid on model logits
y_pred_probs = torch.sigmoid(y_logits)

```

0s completed at 5:44PM

```

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

tensor([[-0.0338],
       [-0.0309],
       [ 0.0894],
       [-0.0692],
       [ 0.2967]], grad_fn=<SliceBackward0>)

[77] # Use sigmoid on model logits
y_pred_probs = torch.sigmoid(y_logits)
y_pred_probs

tensor([[0.4916],
       [0.4923],
       [0.5223],
       [0.4827],
       [0.5736]], grad_fn=<SigmoidBackward0>)

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

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

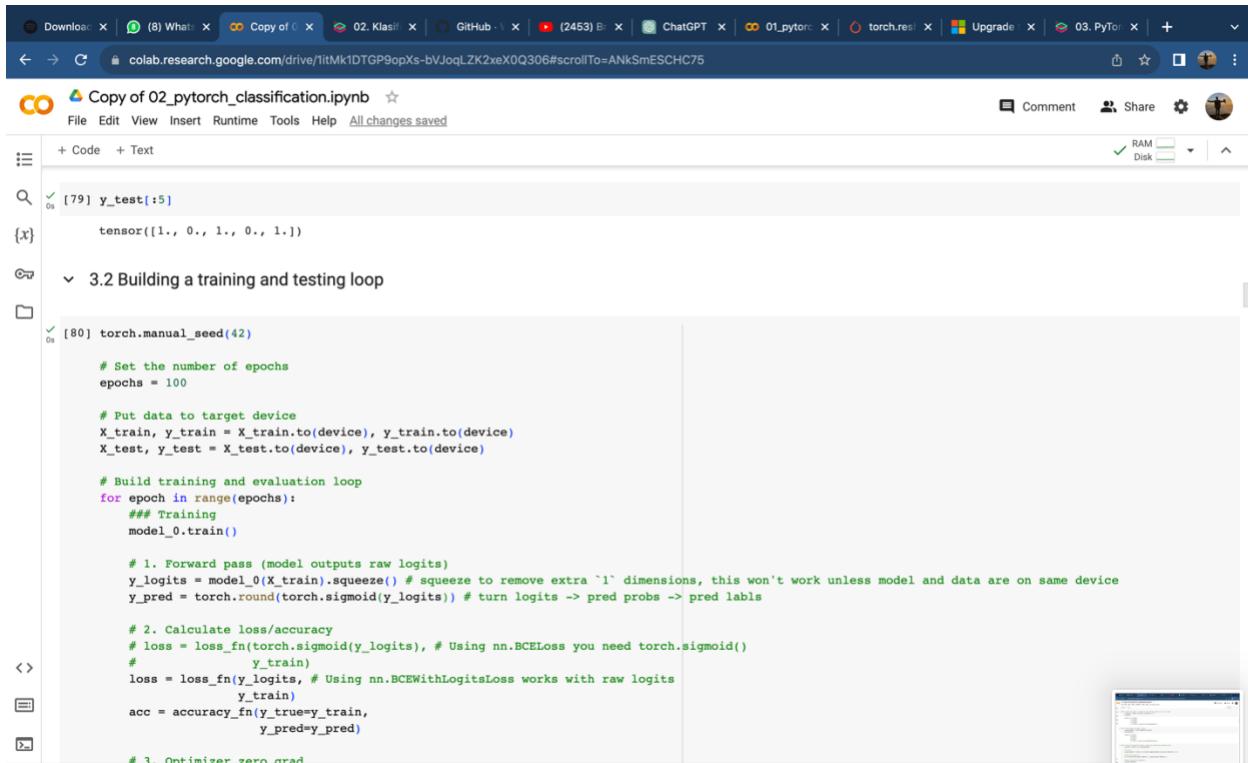
# Check for equality
print(torch.eq(y_preds.squeeze(), y_pred_labels.squeeze()))

# Get rid of extra dimension
y_preds.squeeze()

tensor([True, True, True, True, True])
tensor([0., 0., 1., 0., 1.], grad_fn=<SqueezeBackward0>)

```

0s completed at 5:44PM



```

Copy of 02_pytorch_classification.ipynb
[79] y_test[:5]
tensor([1., 0., 1., 0., 1.])

3.2 Building a training and testing loop

[80] torch.manual_seed(42)

# Set the number of epochs
epochs = 100

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

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

    # 1. Forward pass (model outputs raw logits)
    y_logits = model_0(X_train).squeeze() # squeeze to remove extra '1' dimensions, this won't work unless model and data are on same device
    Y_Pred = torch.round(torch.sigmoid(y_logits)) # turn logits -> pred probs -> pred labs

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

    # 3. Optimizer zero grad
    optimizer.zero_grad()

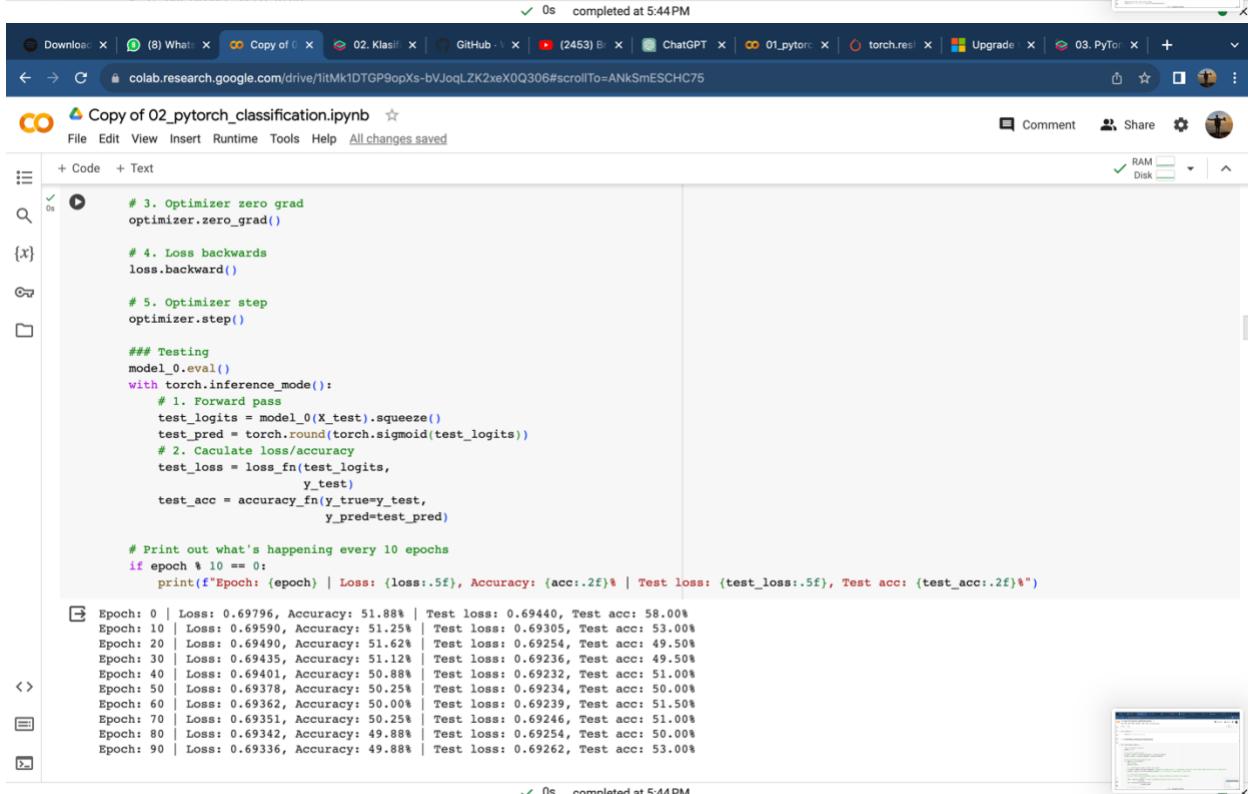
    # 4. Loss backwards
    loss.backward()

    # 5. Optimizer step
    optimizer.step()

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

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

```



```

Epoch: 0 | Loss: 0.69796, Accuracy: 51.88% | Test loss: 0.69440, Test acc: 58.00%
Epoch: 10 | Loss: 0.69590, Accuracy: 51.25% | Test loss: 0.69305, Test acc: 53.00%
Epoch: 20 | Loss: 0.69490, Accuracy: 51.62% | Test loss: 0.69254, Test acc: 49.50%
Epoch: 30 | Loss: 0.69435, Accuracy: 51.12% | Test loss: 0.69236, Test acc: 49.50%
Epoch: 40 | Loss: 0.69401, Accuracy: 50.88% | Test loss: 0.69232, Test acc: 51.00%
Epoch: 50 | Loss: 0.69378, Accuracy: 50.25% | Test loss: 0.69234, Test acc: 50.00%
Epoch: 60 | Loss: 0.69362, Accuracy: 50.00% | Test loss: 0.69239, Test acc: 51.50%
Epoch: 70 | Loss: 0.69351, Accuracy: 50.25% | Test loss: 0.69246, Test acc: 51.00%
Epoch: 80 | Loss: 0.69342, Accuracy: 49.88% | Test loss: 0.69254, Test acc: 50.00%
Epoch: 90 | Loss: 0.69336, Accuracy: 49.88% | Test loss: 0.69262, Test acc: 53.00%

```

Analisis :

- Lihat 5 keluaran pertama dari forward pass data penguji terlihat bahwa didapatkan tensor([[-0.4279], [-0.3417], [-0.5975], [-0.3801], [-0.5078]])

- Untuk mendapatkan keluaran logit model diubah menggunakan fungsi aktivasi sigmoid. Nilai yang terdapat pada output sekarang dalam bentuk probabilitas prediksi. Pada kasus kita, kita akan berurusan dengan klasifikasi biner yaitu 0 dan 1, Jika $y_pred_probs \geq 0,5$, $y=1$ (kelas 1), Jika $y_pred_probs < 0,5$, $y=0$ (kelas 0)
- Ketika dilakukan pengujian y kita membandingkan prediksi model dengan label pengujian untuk melihat seberapa baik kinerjanya.
- Selanjutnya, mengambil keluaran model mentah dan mengonversinya menjadi label prediksi. Mulai dengan melatih selama 100 epoch. Lalu masukan data ke device target dan bangun loop pelatihan dan evaluasi. Lakukan training.
- Output yang dihasilkan menghasilkan akurasi yang hampir tidak melebihi 50% pada setiap pemisahan data.

4. Membuat Prediksi dan Mengevaluasi Model

The screenshot shows two instances of Google Colab notebooks. Both notebooks are titled "Copy of 02_pytorch_classification.ipynb".

Code Snippet:

```

[201] import requests
      from pathlib import Path

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

      from helper_functions import plot_predictions, plot_decision_boundary

```

Plot 1 (Top Notebook): A 2x2 grid of plots. The top row is labeled "Train" and the bottom row is labeled "Test". Each row has two subplots. The left subplot shows a circular distribution of blue and red points with a pink shaded decision boundary. The right subplot shows a similar distribution with a pink shaded decision boundary.

Plot 2 (Bottom Notebook): A 2x2 grid of plots. The top row is labeled "Train" and the bottom row is labeled "Test". Each row has two subplots. The left subplot shows a circular distribution of blue and red points with a pink shaded decision boundary. The right subplot shows a similar distribution with a pink shaded decision boundary.

Section: 5. Improving a model (from a model perspective)

Menggunakan fungsi `helper_functions.py` dan `plot_decision_boundary()` yang membuat meshgrid numpy untuk memplot secara visual berbagai titik dimana model kita memprediksi kelas tertentu. Ditemukan penyebab masalah peforma model, karena data kita berbentuk

lingkaran, untuk memisahkan kedua data tersebut gambar garis lurus ditengah. Dalam istilah machine learning model kita kurang cocok, artinya model tersebut tidak mempelajari pola prediktif dari data.

5. Memperbaiki model (Dari perspektif model)

Ada beberapa Teknik peningkatan model :

1. Tambahkan lebih banyak lapisan

Setiap lapisan *berpotensi* meningkatkan kemampuan pembelajaran model dengan setiap lapisan mampu mempelajari beberapa jenis pola baru dalam data, lebih banyak lapisan sering disebut sebagai membuat jaringan saraf Anda *lebih dalam*.

2. Tambahkan lebih banyak unit tersembunyi

Mirip dengan hal di atas, lebih banyak unit tersembunyi per lapisan berarti *potensi* peningkatan kemampuan pembelajaran model, lebih banyak unit tersembunyi sering disebut sebagai membuat jaringan saraf Anda *lebih luas*.

3. Pemasangan lebih lama (lebih banyak periode)

Model Anda mungkin belajar lebih banyak jika memiliki lebih banyak peluang untuk melihat data.

Langkah untuk memecahkan masalah berguna saat membuat model pembelajaran mendalam adalah memulai dari hal kecil, ini bisa dimulai dengan jaringan neural sederhana dan kumpulan data kecil dan lakukan overfitting. Jumlah data atau ukuran/desain model untuk mengurangi overfitting.

6. Bagian yang hilang : non-linearitas

Untuk memberi kemampuan menggambar garis tidak lurus, buat ulang datanya untuk memulai dari awal. Untuk meletakkan fungsi aktivasi non-linier saat membangun jaringan saraf yaitu menempatkannya di antara lapisan tersembunyi dan tepat setelah lapisan keliaran, namun tidak ada opsi set in stone.

7. Mereplikasi fungsi aktivasi non-linier :

Menggunakan fungsi non linier ReLU PyTorch untuk membantu memodelkan data non-linier

8. Menyatukan berbagai hal membangun model PyTorch multikelas :

Menggabungkan semuanya menggunakan masalah klasifikasi kelas jamak. Klasifikasi biner berhubungan dengan mengklasifikasikan sesuatu sebagai salah satu dari dua pilihan dan klasifikasi kelas jamak berkaitan dengan mengklasifikasikan sesuatu dari daftar lebih dari dua pilihan.

9. Lebih banyak metrik evaluasi klasifikasi

Beberapa cara mengevaluasi model klasifikasi (akurasi, kerugian, dan visualisasi prediksi). Ketepatan Dari 100 prediksi, berapa banyak model Anda yang benar? Misalnya akurasi 95% berarti 95/100 prediksinya benar. Presisi Proporsi hasil positif sejati terhadap jumlah total sampel. Presisi yang lebih tinggi menghasilkan lebih sedikit kesalahan positif (model memprediksi

1 padahal seharusnya 0). Skor F1 Menggabungkan presisi dan perolehan menjadi satu metrik. 1 adalah yang terbaik, 0 adalah yang terburuk.

PyTorch Computer Vision

PyTorch computer vision adalah bidang penggunaan PyTorch yang digunakan untuk memperoleh, memproses, menganalisis dan memahami gambar digital, dan ekstraksi data dimensi tinggi dari dunia nyata untuk menghasilkan informasi numerik atau simbolis, misalnya dalam bentuk Keputusan. Komputer vision biasanya digunakan pada smartphone, kamera, dan foto.

Apa yang dibahas?

1. Libraries computer vision in PyTorch : PyTorch memiliki banyak pustaka visi komputer bawaan yang bermanfaat.
2. Load data : Untuk mempraktikkan visi komputer, kita akan mulai dengan beberapa gambar pakaian yang berbeda dari [FashionMNIST](#).
3. Prepare data : Kita memiliki beberapa gambar yang akan di muat dengan [DataLoader PyTorch](#) sehingga kita dapat menggunakan dengan loop pelatihan kita.
4. Model 0 : Building a baseline model : Di sini kita akan membuat model klasifikasi multi-kelas untuk mempelajari pola dalam data, kita juga akan memilih **fungsi kerugian, pengoptimal**, dan membangun **loop pelatihan**
5. Making Prediction and evaluating model 0 : Mari kita membuat beberapa prediksi dengan model dasar kita dan mengevaluasinya.
6. Setup device agnostic code for future models : Ini adalah praktik terbaik untuk menulis kode perangkat-agnostik, jadi mari kita siapkan.
7. Model 1 : Adding non linearity : Bereksperimen adalah bagian besar dari pembelajaran mesin, mari kita coba dan tingkatkan model dasar kita dengan menambahkan lapisan non-linear.
8. Model 2 : Convolutional Neural Network (CNN) : Saatnya untuk mendapatkan visi komputer spesifik dan memperkenalkan arsitektur jaringan saraf konvolusional yang kuat.
9. Evaluating our best model : Mari kita membuat beberapa prediksi pada gambar acak dan mengevaluasi model terbaik kami.
10. Making a confusiom matrix : Matriks kebingungan adalah cara yang bagus untuk mengevaluasi model klasifikasi, mari kita lihat bagaimana kita bisa membuatnya.
11. Saving and loading the bet performing model : Karena kita mungkin ingin menggunakan model kita untuk nanti, mari kita simpan dan pastikan itu dimuat kembali dengan benar.

Computer Vision adalah seni mengajar komputer untuk melihat :

1. Binary Classification : Proses di mana sebuah model komputer, biasanya berbasis pembelajaran mesin atau pembelajaran mendalam (deep learning), membedakan antara dua kelas atau kategori berdasarkan input berupa gambar atau video. Tujuannya adalah untuk mengklasifikasikan setiap input ke dalam satu dari dua kategori yang tersedia.

2. Multiclass Classification : Sebuah model harus membedakan antara lebih dari dua kelas atau kategori. Dalam hal ini, setiap sampel data diberikan label yang menunjukkan kategori atau kelas tertentu ke mana sampel tersebut termasuk.
3. Object Detection : Melibatkan identifikasi dan penempatan objek dalam sebuah gambar atau video. Tujuan utama dari object detection adalah untuk menentukan lokasi relatif dan kelas (jenis) dari objek-objek yang hadir dalam sebuah input visual. Berbeda dengan klasifikasi yang hanya mencoba mengidentifikasi jenis objek dalam gambar, object detection juga memberikan informasi tentang lokasi atau batasan (bounding box) setiap objek.
4. Segmentation : Membagi gambar atau video menjadi bagian-bagian yang saling terkait atau serupa, yang disebut sebagai segmen. Pada dasarnya, segmentation mencoba untuk mengidentifikasi dan memahami batas-batas antara objek atau wilayah yang berbeda dalam gambar. Terdapat dua jenis utama segmentation: semantic segmentation dan instance segmentation.

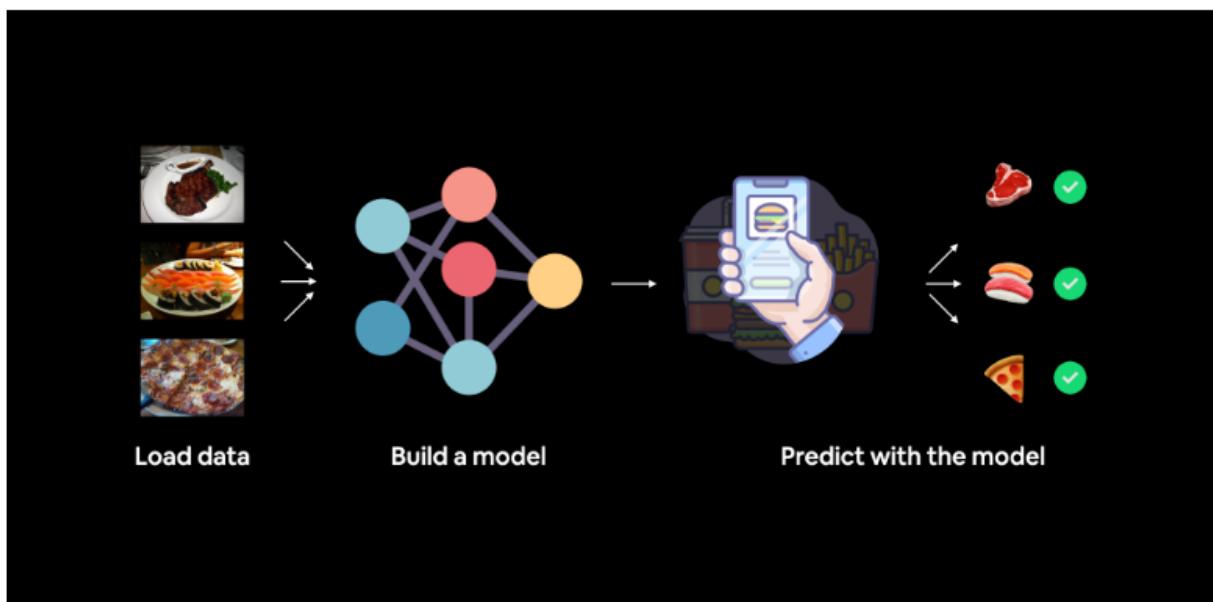
Library dari Computer Vision

1. torchvision : Berisi kumpulan data, arsitektur model, dan transformasi gambar yang sering digunakan untuk masalah penglihatan komputer.
2. torchvision.datasets : Kita akan menemukan banyak contoh kumpulan data visi komputer untuk berbagai masalah mulai dari klasifikasi gambar, deteksi objek, keterangan gambar, klasifikasi video, dan lainnya. Ini juga berisi serangkaian kelas dasar untuk membuat himpunan data kustom.
3. torchvision.models : Modul ini berisi arsitektur model visi komputer yang berkinerja baik dan umum digunakan yang diimplementasikan di PyTorch. Anda dapat menggunakananya dengan masalah Anda sendiri.
4. torchvision.transforms : Seringkali gambar perlu diubah (diubah menjadi angka / diproses / ditambah) sebelum digunakan dengan model, transformasi gambar umum ditemukan di sini.
5. torch.utils.data.Dataset : Kelas himpunan data dasar untuk PyTorch.
6. torch.utils.data.DataLoader : Membuat Python yang dapat diulang melalui himpunan data (dibuat dengan).torch.utils.data.Dataset

PyTorch Custom Datasets

PyTorch Custom Datasets adalah kumpulan data yang berkaitan dengan masalah tertentu yang sedang dikerjakan. Misalnya jika sedang membuat aplikasi klasifikasi gambar makanan, himpunan data khusus yang ada mungkin berupa gambar makanan. Atau jika membuat aplikasi klasifikasi suara, himpunan data kustomnya merupakan sampel suara di samping label sampelnya.

Apa yang akan dibahas?



1. **Importing PyTorch and setting up device-agnostic code** : Mari kita muat PyTorch dan kemudian ikuti praktik terbaik untuk mengatur kode kita agar menjadi perangkat-agnostik.
2. **Get Data** : Kita akan menggunakan **dataset kustom** gambar pizza, steak, dan sushi kita sendiri.
3. **Become one with the data** : Pada awal masalah pembelajaran mesin baru, sangat penting untuk memahami data yang dikerjakan. Di sini kita akan mengambil beberapa langkah untuk mencari tahu data apa yang kita miliki.
4. **Transforming data** : Seringkali, data yang Anda dapatkan tidak akan 100% siap digunakan dengan model pembelajaran mesin, di sini kita akan melihat beberapa langkah yang dapat kita ambil untuk *mengubah* gambar kita sehingga siap digunakan dengan model.
5. **Loading data with imagefolder** : PyTorch memiliki banyak fungsi pemuatan data bawaan untuk jenis data umum. sangat membantu jika gambar kita dalam format klasifikasi gambar standar.ImageFolder
6. **Loading image data with a custom dataset** : Bagaimana jika PyTorch tidak memiliki fungsi bawaan untuk memuat data? Di sinilah kita dapat membangun subkelas kustom kita sendiri dari `.torch.utils.data.Dataset`

7. **Other forms of transforms (data augmentation)** : Penambahan data adalah teknik umum untuk memperluas keragaman data pelatihan. Di sini kita akan mengeksplorasi beberapa fungsi augmentasi data bawaan.torchvision
8. **Model 0** : TinyVGG without data augmentation : Pada tahap ini, kita akan menyiapkan data kita, mari kita membangun model yang mampu menyesuaikannya. Kami juga akan membuat beberapa fungsi pelatihan dan pengujian untuk melatih dan mengevaluasi model kami.
9. **Exploring loss curves** : Kurva kerugian adalah cara yang bagus untuk melihat bagaimana model Anda melatih / meningkatkan dari waktu ke waktu. Mereka juga merupakan cara yang baik untuk melihat apakah model Anda underfitting atau overfitting.
10. **Model 1 : TinyVGG with data augmentation** : Sekarang, kami sudah mencoba model *tanpa*, bagaimana kalau kami mencobanya *dengan* augmentasi data?
11. **Compare model results** : Mari kita bandingkan kurva kerugian model kami yang berbeda dan lihat mana yang berkinerja lebih baik dan diskusikan beberapa opsi untuk meningkatkan kinerja.
12. **Making a prediction on a custom image** : Model kami dilatih untuk pada dataset gambar pizza, steak dan sushi. Di bagian ini, kami akan membahas cara menggunakan model terlatih kami untuk memprediksi gambar di *luar* himpunan data kami yang ada.