#Buatlah data klasifikasi dan siapkan

Kita akan menggunakan metode make_circles() dari Scikit-Learn untuk menghasilkan dua lingkaran dengan titik berwarna berbeda.

```
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
```

Baiklah, sekarang mari kita lihat 5 nilai X dan y yang pertama.

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

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

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

Mari terus ikuti moto penjelajah data yaitu pandas dan memasukkannya ke dalam DataFrame.

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

         X1        X2  label
0  0.754246  0.231481      1
1 -0.756159  0.153259      1
2 -0.815392  0.173282      1
3 -0.393731  0.692883      1
4  0.442208 -0.896723      0
5 -0.479646  0.676435      1
```

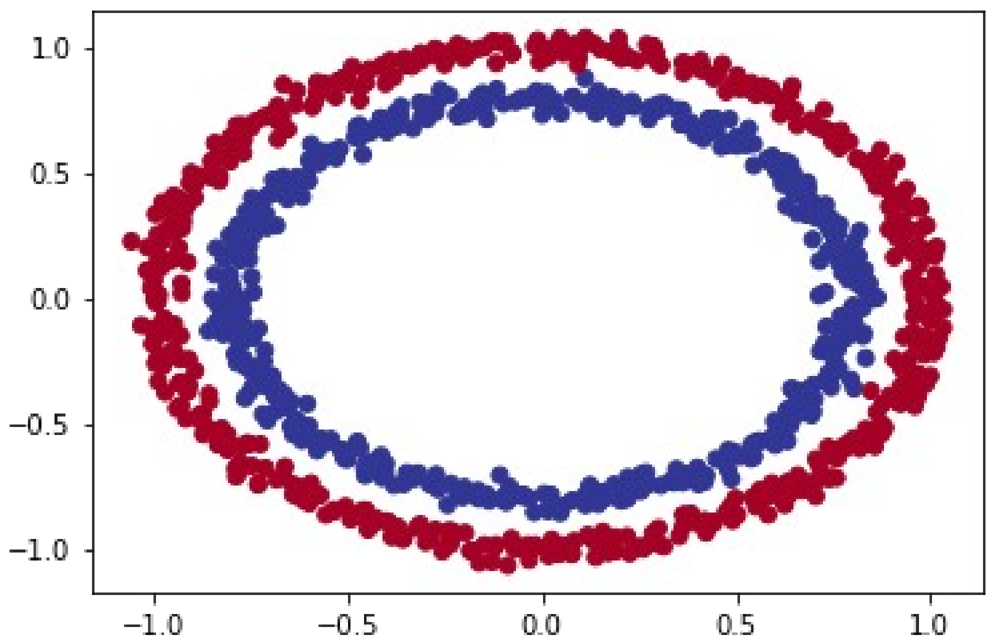
```
6 -0.013648  0.803349      1
7  0.771513  0.147760      1
8 -0.169322 -0.793456      1
9 -0.121486  1.021509      0
```

Berapa banyak nilai setiap kelas yang ada?

```
# Check different labels
circles.label.value_counts()

1    500
0    500
Name: label, dtype: int64
```

Mari kita gambarkan.

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



#Bentuk masukan dan keluaran

Apa bentuk masukan saya dan apa bentuk keluaran saya?

```
# Check the shapes of our features and labels
X.shape, y.shape
```

```
((1000, 2), (1000,))
```

Melakukan hal ini akan membantu Anda memahami bentuk masukan dan keluaran yang Anda harapkan dari model Anda.

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

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

Ubah data menjadi tensor dan buat pemisahan pelatihan dan pengujian

```
# 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.]))
```

Kita akan menggunakan test_size=0.2 (80% pelatihan, 20% pengujian) dan karena pemisahan terjadi secara acak di seluruh data, mari gunakan random_state=42 sehingga pemisahan dapat direproduksi.

```
# 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)
```

Membangun model

Mari kita mulai dengan mengimpor PyTorch dan torch.nn serta menyiapkan kode agnostik perangkat.

```
# Standard PyTorch imports
import torch
from torch import nn

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

```
'cuda'
```

Mari kita buat kelas model yang:

1. Subkelas nn.Module (hampir semua model PyTorch adalah subkelas dari nn.Module).
2. Membuat 2 lapisan nn.Linear di konstruktor yang mampu menangani bentuk masukan dan keluaran X dan y.
3. Mendefinisikan metode forward() yang berisi perhitungan forward pass model.
4. Membuat instance kelas model dan mengirimkannya ke perangkat target.

```
# 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)
)
```

nn.Sequential melakukan perhitungan forward pass dari data masukan melalui lapisan sesuai urutan kemunculannya.

```
# 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)
)
```

Sekarang kita punya modelnya, mari kita lihat apa yang terjadi ketika kita melewatkan beberapa data melaluinya.

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

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

First 10 predictions:
tensor([[-0.4279],
        [-0.3417],
        [-0.5975],
        [-0.3801],
        [-0.5078],
        [-0.4559],
        [-0.2842],
```

```
        [-0.3107],
        [-0.6010],
        [-0.3350]], device='cuda:0', grad_fn=<SliceBackward0>)

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

#Mengatur fungsi kerugian dan pengoptimal

mari buat fungsi kerugian dan pengoptimal.

```
# 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)
```

Akurasi dapat diukur dengan membagi jumlah prediksi yang benar dengan jumlah total prediksi.

Misalnya, model yang membuat 99 prediksi benar dari 100 prediksi akan memiliki akurasi 99%

```
# 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
```

#Beralih dari keluaran model mentah ke label prediksi (logit -> probabilitas prediksi -> label prediksi)

Untuk melakukannya, mari kita teruskan beberapa data ke model.

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

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

Untuk mendapatkan keluaran mentah (logit) model kita ke dalam bentuk seperti itu, kita dapat menggunakan fungsi aktivasi sigmoid

```
# Use sigmoid on model logits
y_pred_probs = torch.sigmoid(y_logits)
y_pred_probs
```

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

Untuk mengubah probabilitas prediksi kita menjadi label prediksi, kita dapat membulatkan keluaran sigmoid fungsi aktivasi.

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

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

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

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

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

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

Sekarang sepertinya prediksi model kita memiliki bentuk yang sama dengan label kebenaran kita (y_test).

```
y_test[:5]
```

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

#Membangun lingkaran pelatihan dan pengujian

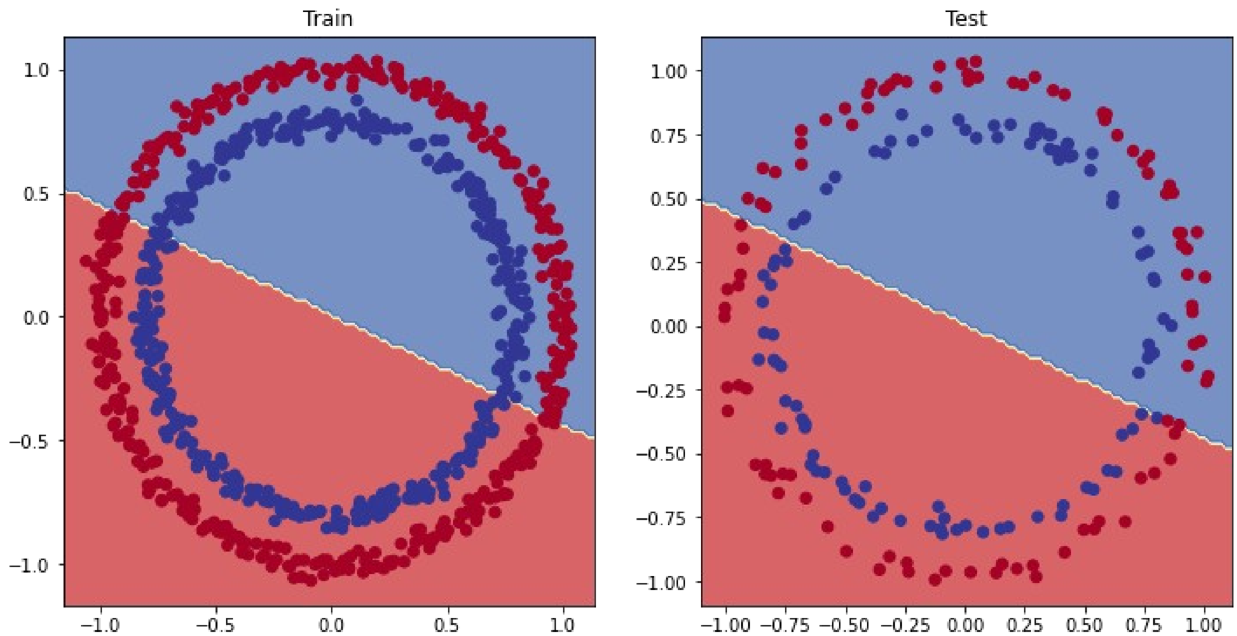
Mari kita mulai dengan melatih selama 100 epoch dan menampilkan kemajuan model setiap 10 epoch.

```
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 labls

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

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backwards
    loss.backward()

    # 5. Optimizer step
    optimizer.step()

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

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

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

# Membuat prediksi dan mengevaluasi model

Ini berisi fungsi bermanfaat yang disebut plot_decision_boundary()
yang membuat meshgrid NumPy untuk
secara visual memplot berbagai titik di mana model kita memprediksi
kelas tertentu.

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

helper_functions.py already exists, skipping download

/home/daniel/.local/lib/python3.8/site-packages/torchvision/io/
image.py:13: UserWarning: Failed to load image Python extension:
/home/daniel/.local/lib/python3.8/site-packages/torchvision/image.so:
```

```
undefined symbol: _ZN3c106detail19maybe_wrap_dim_slowIlEET_S2_S2_b
  warn(f"Failed to load image Python extension: {e}")

# 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_0, X_train, y_train)
plt.subplot(1, 2, 2)
plt.title("Test")
plot_decision_boundary(model_0, X_test, y_test)
```



#Memperbaiki model (dari perspektif model)

Mari kita lihat apa yang terjadi jika kita menambahkan lapisan ekstra ke model kita, menyesuaikannya lebih lama (epoch=1000, bukan epoch=100) dan menambah jumlah unit tersembunyi dari 5 menjadi 10.

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

    def forward(self, x): # note: always make sure forward is spelt
correctly!
        # Creating a model like this is the same as below, though
below
```

```
        # generally benefits from speedups where possible.
        # z = self.layer_1(x)
        # z = self.layer_2(z)
        # z = self.layer_3(z)
        # return z
        return self.layer_3(self.layer_2(self.layer_1(x)))

model_1 = CircleModelV1().to(device)
model_1
```

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

Sekarang kita punya model, kita akan membuat ulang fungsi kerugian dan instance pengoptimal, menggunakan pengaturan yang sama seperti sebelumnya.

```
# loss_fn = nn.BCELoss() # Requires sigmoid on input
loss_fn = nn.BCEWithLogitsLoss() # Does not require sigmoid on input
optimizer = torch.optim.SGD(model_1.parameters(), lr=0.1)
```

Kali ini kita akan berlatih lebih lama (epochs=1000 vs epochs=100) dan melihat apakah ini meningkatkan model kita

```
torch.manual_seed(42)

epochs = 1000 # Train for longer

# 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)

for epoch in range(epochs):
    ### Training
    # 1. Forward pass
    y_logits = model_1(X_train).squeeze()
    y_pred = torch.round(torch.sigmoid(y_logits)) # logits -> 
predicition probabilities -> prediction labels

    # 2. Calculate loss/accuracy
    loss = loss_fn(y_logits, y_train)
    acc = accuracy_fn(y_true=y_train,
                      y_pred=y_pred)

    # 3. Optimizer zero grad
    optimizer.zero_grad()
```

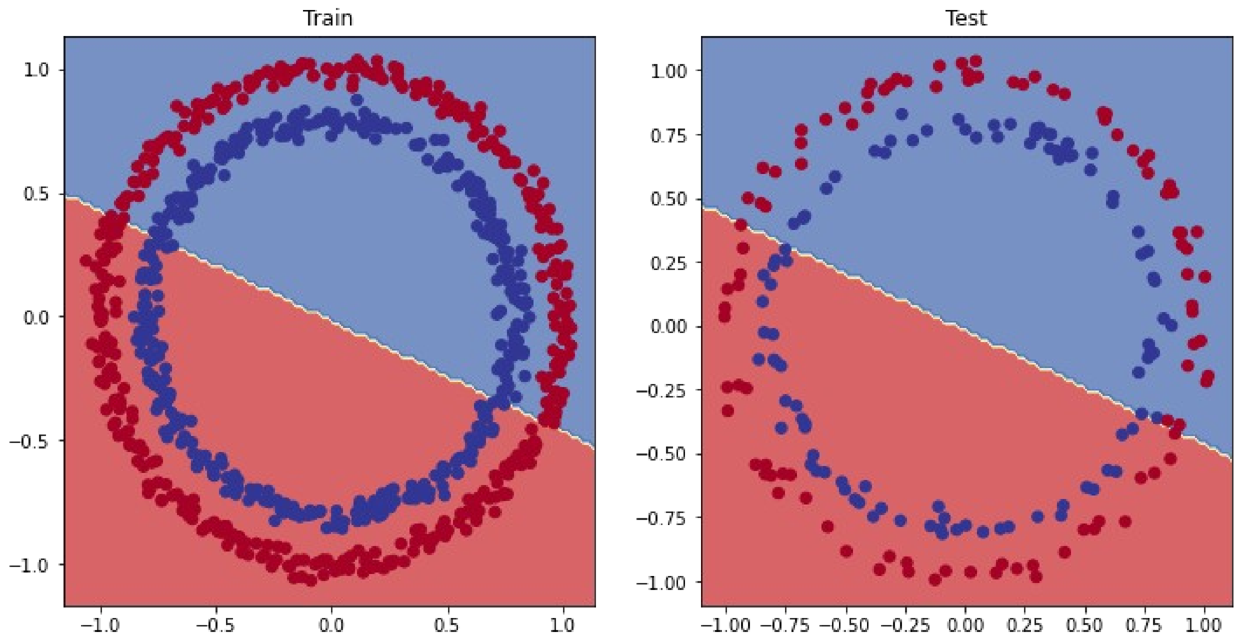
```
    # 4. Loss backwards
    loss.backward()

    # 5. Optimizer step
    optimizer.step()

    ### Testing
    model_1.eval()
    with torch.inference_mode():
        # 1. Forward pass
        test_logits = model_1(X_test).squeeze()
        test_pred = torch.round(torch.sigmoid(test_logits))
        # 2. Caculate loss/accuracy
        test_loss = loss_fn(test_logits,
                            y_test)
        test_acc = accuracy_fn(y_true=y_test,
                               y_pred=test_pred)

    # Print out what's happening every 10 epochs
    if epoch % 100 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc:.2f}
% | Test loss: {test_loss:.5f}, Test acc: {test_acc:.2f}%")
```

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

Model kami dilatih lebih lama dan dengan lapisan tambahan, tetapi sepertinya model tersebut masih tidak mempelajari pola apa pun lebih baik daripada menebak secara acak.

```
# 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)
```



#Mempersiapkan data untuk melihat apakah model kita dapat memodelkan garis lurus

Mari kita buat beberapa data linier untuk melihat apakah model kita mampu memodelkannya dan kita tidak hanya menggunakan model yang tidak dapat mempelajari apa pun.

```
# Create some data (same as notebook 01)
weight = 0.7
bias = 0.3
start = 0
end = 1
step = 0.01

# Create data
X_regression = torch.arange(start, end, step).unsqueeze(dim=1)
y_regression = weight * X_regression + bias # linear regression
formula

# Check the data
print(len(X_regression))
X_regression[:5], y_regression[:5]

100
```

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

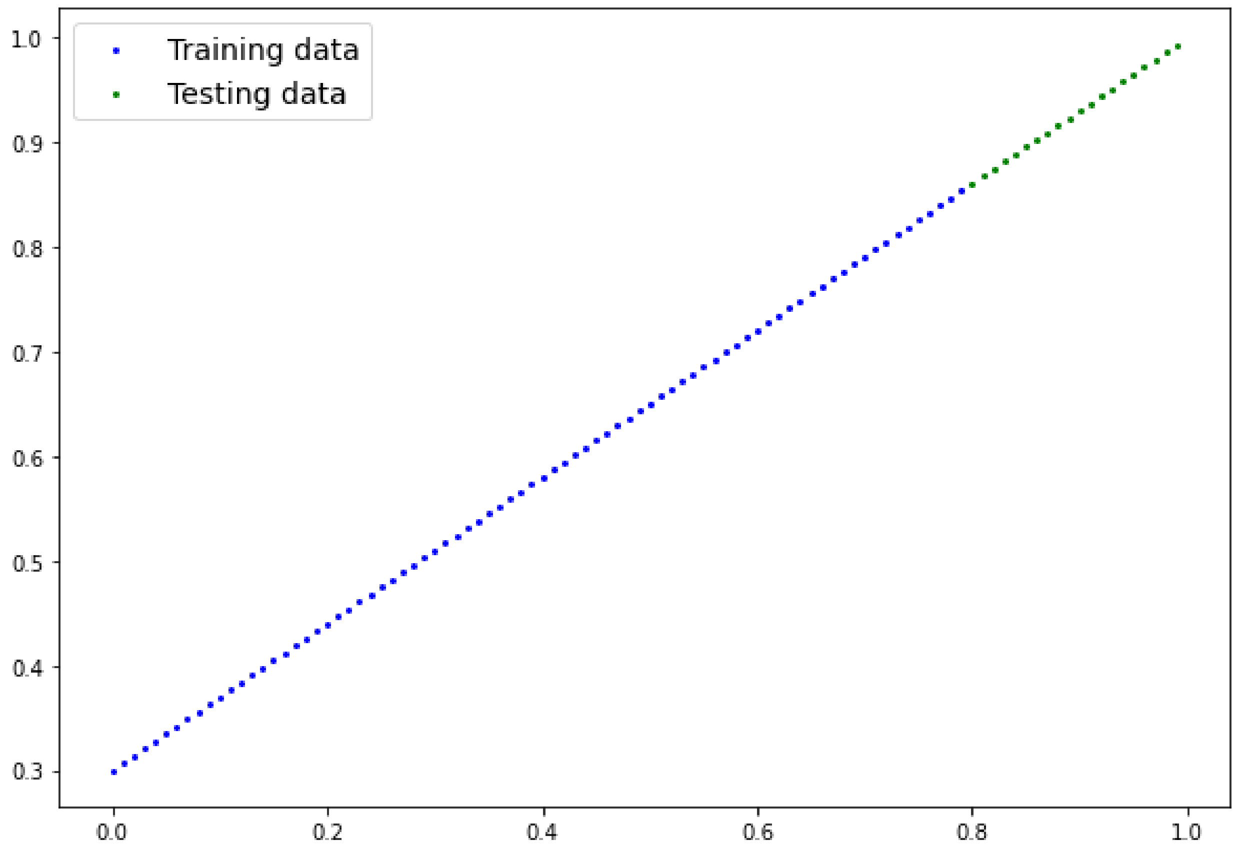
Hebat, sekarang mari kita bagi data kita menjadi set pelatihan dan pengujian.

```
# Create train and test splits
train_split = int(0.8 * len(X_regression)) # 80% of data used for
training set
X_train_regression, y_train_regression = X_regression[:train_split],
y_regression[:train_split]
X_test_regression, y_test_regression = X_regression[train_split:],
y_regression[train_split:]

# Check the lengths of each split
print(len(X_train_regression),
    len(y_train_regression),
    len(X_test_regression),
    len(y_test_regression))
```

```
80 80 20 20
```

Untuk melakukannya, kita akan menggunakan fungsi plot_predictions() yang kita buat di notebook 01.

```
plot_predictions(train_data=X_train_regression,
    train_labels=y_train_regression,
    test_data=X_test_regression,
    test_labels=y_test_regression
);
```



#Menyesuaikan model_1 agar sesuai dengan garis lurus

Sekarang kita punya beberapa data, mari buat ulang model_1 tetapi dengan fungsi kerugian yang sesuai dengan data regresi kita.

```
# Same architecture as model_1 (but using nn.Sequential)
model_2 = nn.Sequential(
    nn.Linear(in_features=1, out_features=10),
    nn.Linear(in_features=10, out_features=10),
    nn.Linear(in_features=10, out_features=1)
).to(device)

model_2
```

```
Sequential(
  (0): Linear(in_features=1, out_features=10, bias=True)
  (1): Linear(in_features=10, out_features=10, bias=True)
  (2): Linear(in_features=10, out_features=1, bias=True)
)
```

Kami akan menyiapkan fungsi kerugian menjadi nn.L1Loss() (sama dengan kesalahan absolut rata-rata) dan pengoptimal menjadi torch.optim.SGD().

```
# Loss and optimizer
loss_fn = nn.L1Loss()
optimizer = torch.optim.SGD(model_2.parameters(), lr=0.1)
```

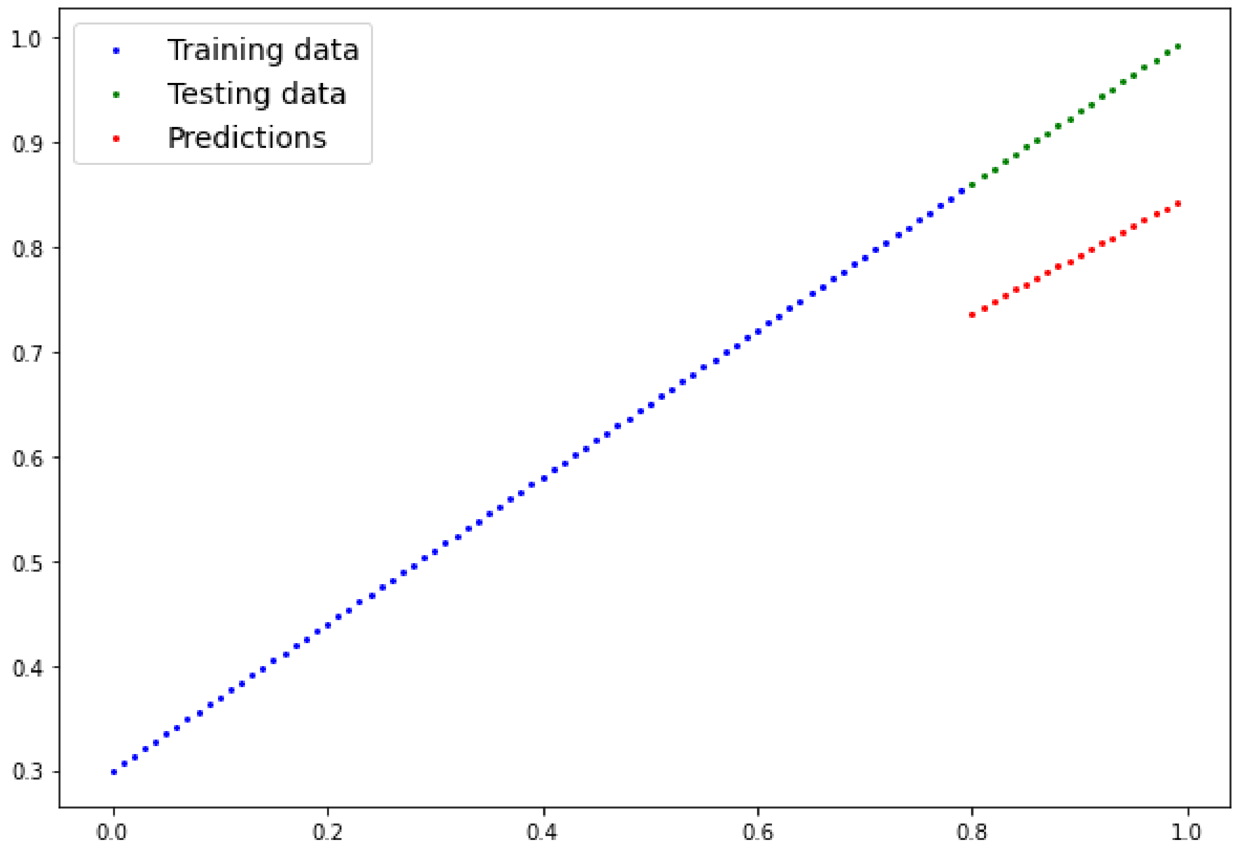
Sekarang mari kita latih model menggunakan langkah-langkah loop pelatihan reguler untuk epochs=1000 (seperti model_1).

```
# Train the model
torch.manual_seed(42)

# Set the number of epochs
epochs = 1000

# Put data to target device
X_train_regression, y_train_regression =
X_train_regression.to(device), y_train_regression.to(device)
X_test_regression, y_test_regression = X_test_regression.to(device),
y_test_regression.to(device)

for epoch in range(epochs):
    ### Training
    # 1. Forward pass
    y_pred = model_2(X_train_regression)

    # 2. Calculate loss (no accuracy since it's a regression problem,
not classification)
    loss = loss_fn(y_pred, y_train_regression)

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backwards
    loss.backward()

    # 5. Optimizer step
    optimizer.step()

    ### Testing
    model_2.eval()
    with torch.inference_mode():
      # 1. Forward pass
      test_pred = model_2(X_test_regression)
      # 2. Calculate the loss
      test_loss = loss_fn(test_pred, y_test_regression)

    # Print out what's happening
    if epoch % 100 == 0:
        print(f"Epoch: {epoch} | Train loss: {loss:.5f}, Test loss:
{test_loss:.5f}")
```

```
Epoch: 0 | Train loss: 0.75986, Test loss: 0.54143
Epoch: 100 | Train loss: 0.09309, Test loss: 0.02901
Epoch: 200 | Train loss: 0.07376, Test loss: 0.02850
Epoch: 300 | Train loss: 0.06745, Test loss: 0.00615
Epoch: 400 | Train loss: 0.06107, Test loss: 0.02004
Epoch: 500 | Train loss: 0.05698, Test loss: 0.01061
Epoch: 600 | Train loss: 0.04857, Test loss: 0.01326
Epoch: 700 | Train loss: 0.06109, Test loss: 0.02127
Epoch: 800 | Train loss: 0.05599, Test loss: 0.01426
Epoch: 900 | Train loss: 0.05571, Test loss: 0.00603
```

kami akan mengirimkan semua data kami ke CPU menggunakan .cpu() saat kami meneruskannya ke plot_predictions().

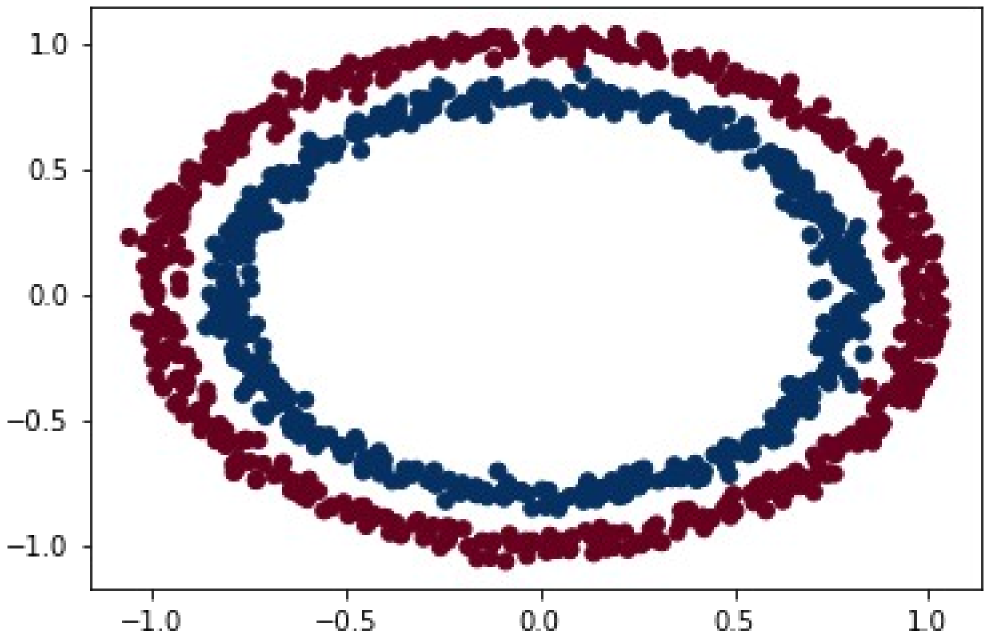
```
# Turn on evaluation mode
model_2.eval()

# Make predictions (inference)
with torch.inference_mode():
    y_preds = model_2(X_test_regression)

# Plot data and predictions with data on the CPU (matplotlib can't
handle data on the GPU)
# (try removing .cpu() from one of the below and see what happens)
plot_predictions(train_data=X_train_regression.cpu(),
                 train_labels=y_train_regression.cpu(),
                 test_data=X_test_regression.cpu(),
                 test_labels=y_test_regression.cpu(),
                 predictions=y_preds.cpu());
```



#Membuat ulang data non-linier (lingkaran merah dan biru)

Pertama, mari kita buat ulang datanya untuk memulai dari awal. Kami akan menggunakan pengaturan yang sama seperti sebelumnya.

```
# Make and plot data
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles

n_samples = 1000

X, y = make_circles(n_samples=1000,
    noise=0.03,
    random_state=42,
)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdBu);
```




```
# Convert to tensors and split into train and test sets
import torch
from sklearn.model_selection import train_test_split

# Turn data into tensors
X = torch.from_numpy(X).type(torch.float)
y = torch.from_numpy(y).type(torch.float)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42
)

X_train[:5], y_train[:5]
```

```
(tensor([[ 0.6579, -0.4651],
         [ 0.6319, -0.7347],
         [-1.0086, -0.1240],
         [-0.9666, -0.2256],
         [-0.1666,  0.7994]]),
 tensor([1., 0., 0., 0., 1.]))
```

#Membangun model dengan non-linearitas

mari kita letakkan di jaringan saraf kita di antara lapisan tersembunyi di forward pass dan lihat apa yang terjadi.

```python
# Build model with non-linear activation function
from torch import nn
class CircleModelV2(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(in_features=2, out_features=10)
        self.layer_2 = nn.Linear(in_features=10, out_features=10)
        self.layer_3 = nn.Linear(in_features=10, out_features=1)
        self.relu = nn.ReLU() # <- add in ReLU activation function
        # Can also put sigmoid in the model
        # This would mean you don't need to use it on the predictions
        # self.sigmoid = nn.Sigmoid()

    def forward(self, x):
      # Intersperse the ReLU activation function between layers
       return
self.layer_3(self.relu(self.layer_2(self.relu(self.layer_1(x)))))

model_3 = CircleModelV2().to(device)
print(model_3)
```

```
CircleModelV2(
  (layer_1): Linear(in_features=2, out_features=10, bias=True)
  (layer_2): Linear(in_features=10, out_features=10, bias=True)
  (layer_3): Linear(in_features=10, out_features=1, bias=True)
  (relu): ReLU()
)
```

```python
# Setup loss and optimizer
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.SGD(model_3.parameters(), lr=0.1)
```

#Melatih model dengan non-linearitas

Anda sudah mengetahui latihan, model, fungsi kerugian, pengoptimal yang siap digunakan, mari buat loop pelatihan dan pengujian.

```python
# Fit the model
torch.manual_seed(42)
epochs = 1000

# Put all data on 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)

for epoch in range(epochs):
    # 1. Forward pass
    y_logits = model_3(X_train).squeeze()
    y_pred = torch.round(torch.sigmoid(y_logits)) # logits ->
prediction probabilities -> prediction labels
```

```
    # 2. Calculate loss and accuracy
    loss = loss_fn(y_logits, y_train) # BCEWithLogitsLoss calculates
loss using logits
    acc = accuracy_fn(y_true=y_train,
                      y_pred=y_pred)

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backward
    loss.backward()

    # 5. Optimizer step
    optimizer.step()

    ### Testing
    model_3.eval()
    with torch.inference_mode():
      # 1. Forward pass
      test_logits = model_3(X_test).squeeze()
      test_pred = torch.round(torch.sigmoid(test_logits)) # logits ->
prediction probabilities -> prediction labels
      # 2. Calcuate loss and 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
    if epoch % 100 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc:.2f}
% | Test Loss: {test_loss:.5f}, Test Accuracy: {test_acc:.2f}%")
```

```
Epoch: 0 | Loss: 0.69295, Accuracy: 50.00% | Test Loss: 0.69319, Test
Accuracy: 50.00%
Epoch: 100 | Loss: 0.69115, Accuracy: 52.88% | Test Loss: 0.69102,
Test Accuracy: 52.50%
Epoch: 200 | Loss: 0.68977, Accuracy: 53.37% | Test Loss: 0.68940,
Test Accuracy: 55.00%
Epoch: 300 | Loss: 0.68795, Accuracy: 53.00% | Test Loss: 0.68723,
Test Accuracy: 56.00%
Epoch: 400 | Loss: 0.68517, Accuracy: 52.75% | Test Loss: 0.68411,
Test Accuracy: 56.50%
Epoch: 500 | Loss: 0.68102, Accuracy: 52.75% | Test Loss: 0.67941,
Test Accuracy: 56.50%
Epoch: 600 | Loss: 0.67515, Accuracy: 54.50% | Test Loss: 0.67285,
Test Accuracy: 56.00%
Epoch: 700 | Loss: 0.66659, Accuracy: 58.38% | Test Loss: 0.66322,
Test Accuracy: 59.00%
Epoch: 800 | Loss: 0.65160, Accuracy: 64.00% | Test Loss: 0.64757,
```

```
Test Accuracy: 67.50%
Epoch: 900 | Loss: 0.62362, Accuracy: 74.00% | Test Loss: 0.62145,
Test Accuracy: 79.00%
```

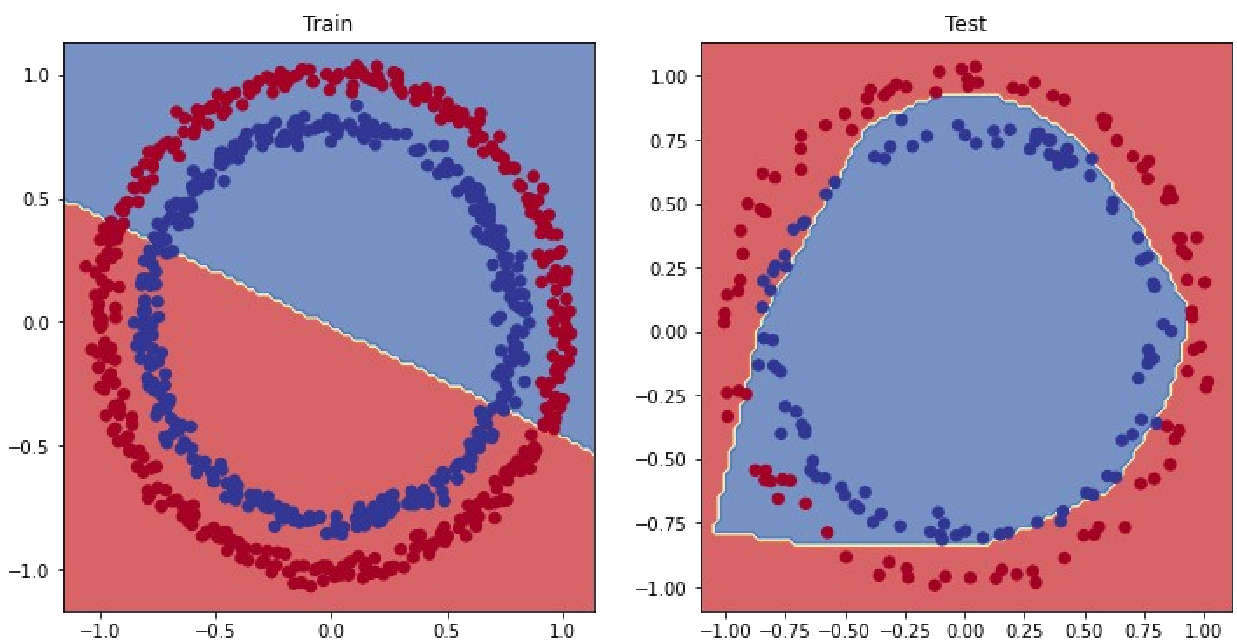
#Mengevaluasi model yang dilatih dengan fungsi aktivasi non-linier

mari kita lihat tampilan prediksi model kita sekarang karena model tersebut telah dilatih dengan fungsi aktivasi non-linier.

```
# Make predictions
model_3.eval()
with torch.inference_mode():
    y_preds = torch.round(torch.sigmoid(model_3(X_test))).squeeze()
y_preds[:10], y[:10] # want preds in same format as truth labels
```

```
(tensor([1., 0., 1., 0., 0., 1., 0., 0., 1., 0.], device='cuda:0'),
 tensor([1., 1., 1., 1., 0., 1., 1., 1., 1., 0.]))
```

```
# 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) # model_1 = no non-linearity
plt.subplot(1, 2, 2)
plt.title("Test")
plot_decision_boundary(model_3, X_test, y_test) # model_3 = has non-linearity
```



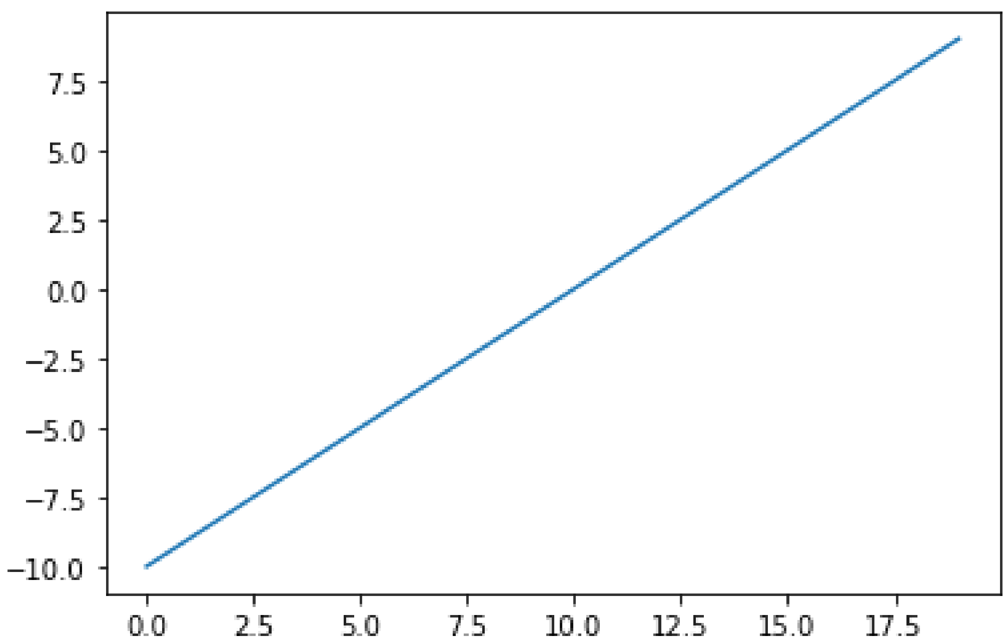
#Mereplikasi fungsi aktivasi non-linier

Mari kita mulai dengan membuat sejumlah kecil data.

```
# Create a toy tensor (similar to the data going into our model(s))
A = torch.arange(-10, 10, 1, dtype=torch.float32)
A

tensor([-10.,  -9.,  -8.,  -7.,  -6.,  -5.,  -4.,  -3.,  -2.,  -1.,
0.,   1.,
          2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.])
```

sekarang mari kita plot.

```
# Visualize the toy tensor
plt.plot(A);
```

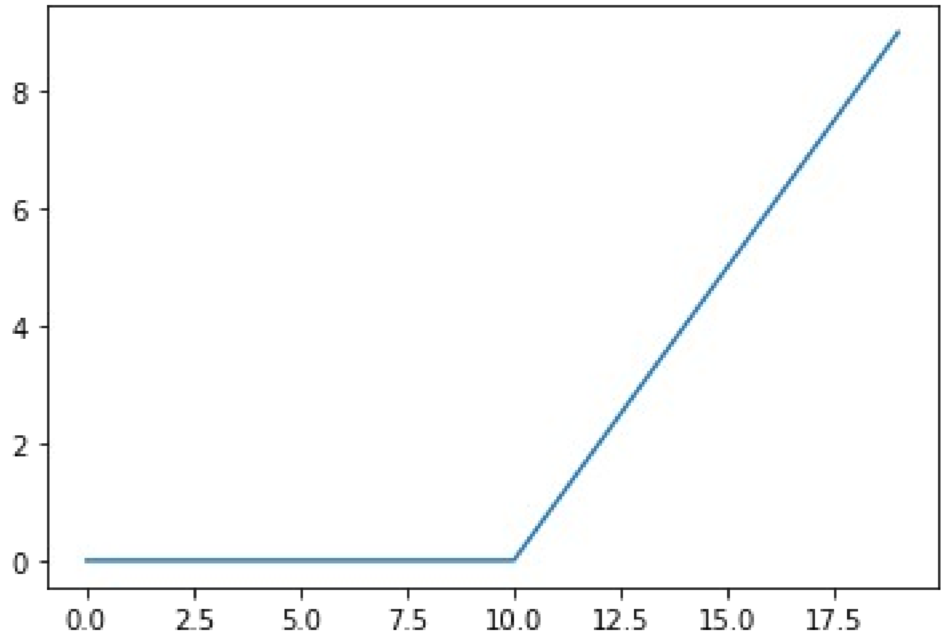


Sekarang mari kita lihat bagaimana fungsi aktivasi ReLU mempengaruhinya.

```
# Create ReLU function by hand
def relu(x):
  return torch.maximum(torch.tensor(0), x) # inputs must be tensors

# Pass toy tensor through ReLU function
relu(A)

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

Mari kita gambarkan.

```
# Plot ReLU activated toy tensor
plt.plot(relu(A));
```

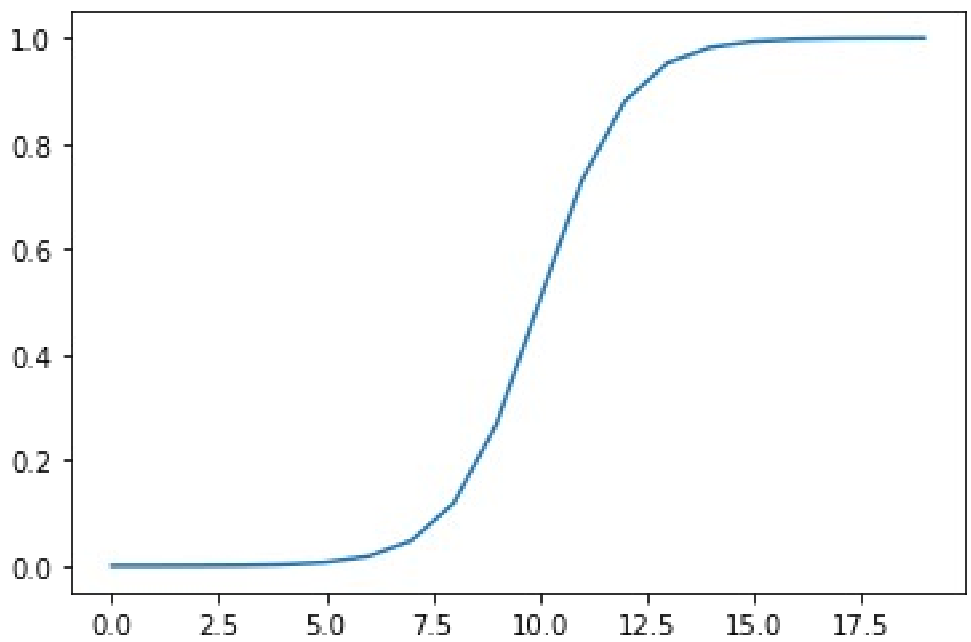


Mari buat fungsi untuk mereplikasi fungsi sigmoid dengan PyTorch.

```
# Create a custom sigmoid function
def sigmoid(x):
  return 1 / (1 + torch.exp(-x))

# Test custom sigmoid on toy tensor
sigmoid(A)

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

mari kita lihat seperti apa visualisasinya.

```
# Plot sigmoid activated toy tensor
plt.plot(sigmoid(A));
```



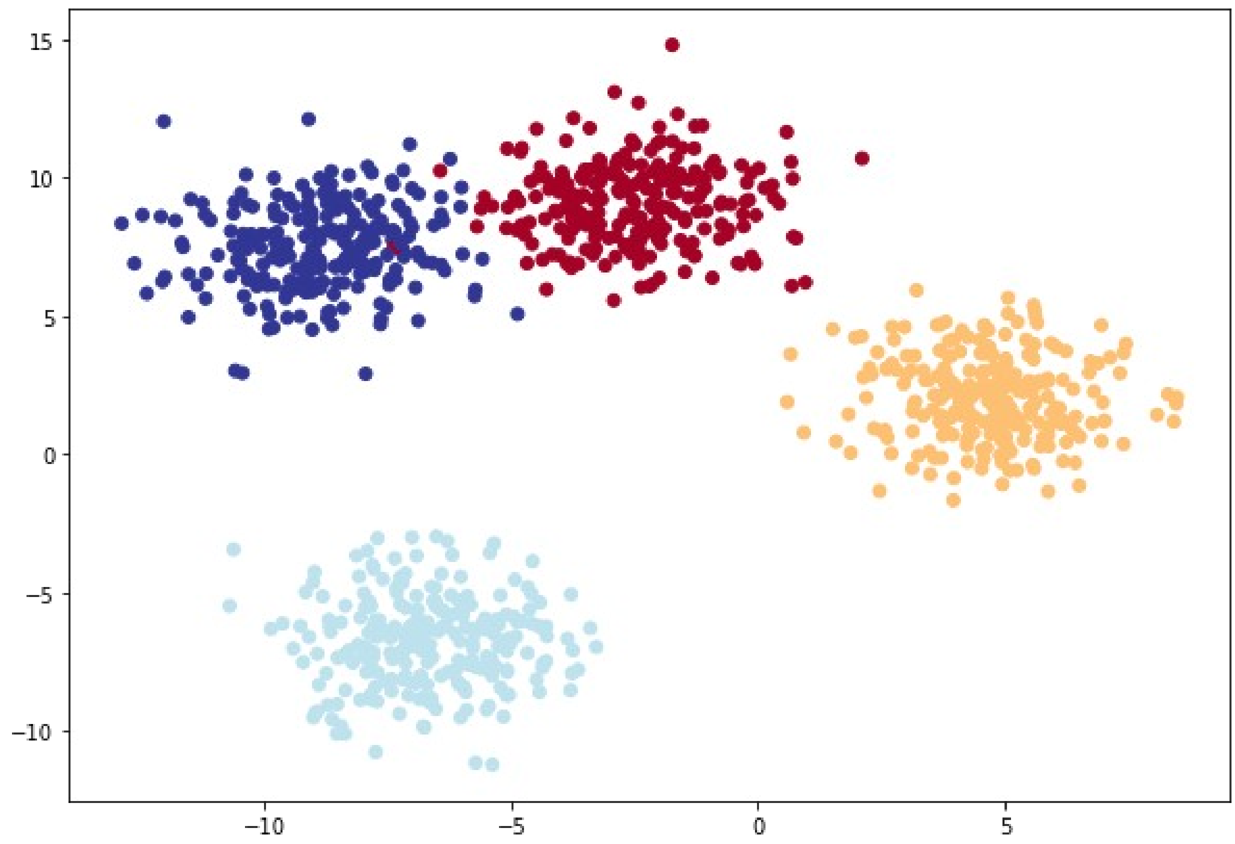
#Membuat data klasifikasi mutli-kelas

Untuk melakukannya, kita dapat memanfaatkan metode make_blobs() Scikit-Learn.

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

# Set the hyperparameters for data creation
NUM_CLASSES = 4
NUM_FEATURES = 2
RANDOM_SEED = 42

# 1. Create multi-class data
X_blob, y_blob = make_blobs(n_samples=1000,
    n_features=NUM_FEATURES, # X features
    centers=NUM_CLASSES, # y labels
    cluster_std=1.5, # give the clusters a little shake up (try
changing this to 1.0, the default)
    random_state=RANDOM_SEED
)

# 2. Turn data into tensors
X_blob = torch.from_numpy(X_blob).type(torch.float)
y_blob = torch.from_numpy(y_blob).type(torch.LongTensor)
print(X_blob[:5], y_blob[:5])

# 3. Split into train and test sets
```

```
X_blob_train, X_blob_test, y_blob_train, y_blob_test =
train_test_split(X_blob,
    y_blob,
    test_size=0.2,
    random_state=RANDOM_SEED
)

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

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



Membangun model klasifikasi kelas jamak di PyTorch

Untuk melakukannya, mari buat subkelas nn.Module yang menggunakan tiga hyperparameter: • input_features - jumlah fitur X yang masuk ke dalam model. • output_features - jumlah ideal fitur keluaran yang kita inginkan (ini adalah setara dengan NUM_CLASSES atau jumlah kelas dalam masalah klasifikasi kelas jamak Anda). • Hidden_units - jumlah neuron tersembunyi yang ingin kita gunakan pada setiap lapisan tersembunyi.

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

```
'cuda'
```

```
from torch import nn

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

        Args:
            input_features (int): Number of input features to the
model.
            out_features (int): Number of output features of the model
              (how many classes there are).
            hidden_units (int): Number of hidden units between layers,
default 8.
        """
        super().__init__()
        self.linear_layer_stack = nn.Sequential(
            nn.Linear(in_features=input_features,
out_features=hidden_units),
            # nn.ReLU(), # <- does our dataset require non-linear
layers? (try uncommenting and see if the results change)
            nn.Linear(in_features=hidden_units,
out_features=hidden_units),
            # nn.ReLU(), # <- does our dataset require non-linear
layers? (try uncommenting and see if the results change)
            nn.Linear(in_features=hidden_units,
out_features=output_features), # how many classes are there?
        )

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

```
# Create an instance of BlobModel and send it to the target device
model_4 = BlobModel(input_features=NUM_FEATURES,
                    output_features=NUM_CLASSES,
                    hidden_units=8).to(device)
model_4

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

#Membuat fungsi kerugian dan pengoptimal untuk model PyTorch multi-kelas

Dan kami akan tetap menggunakan SGD dengan kecepatan pemelajaran 0,1 untuk mengoptimalkan parameter model_4 kami.

```
# Create loss and optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_4.parameters(),
                            lr=0.1) # exercise: try changing the
learning rate here and seeing what happens to the model's performance
```

#Mendapatkan probabilitas prediksi untuk model PyTorch kelas jamak

kita lakukan satu forward pass dengan model kita untuk melihat apakah model tersebut berfungsi.

```
# Perform a single forward pass on the data (we'll need to put it to
the target device for it to work)
model_4(X_blob_train.to(device))[:5]

tensor([[-1.2711, -0.6494, -1.4740, -0.7044],
        [ 0.2210, -1.5439,  0.0420,  1.1531],
        [ 2.8698,  0.9143,  3.3169,  1.4027],
        [ 1.9576,  0.3125,  2.2244,  1.1324],
        [ 0.5458, -1.2381,  0.4441,  1.1804]], device='cuda:0',
       grad_fn=<SliceBackward0>)
```

Mari kita periksa bentuknya untuk mengonfirmasi.

```
# How many elements in a single prediction sample?
model_4(X_blob_train.to(device))[0].shape, NUM_CLASSES

(torch.Size([4]), 4)
```

Fungsi softmax menghitung probabilitas setiap kelas prediksi menjadi kelas prediksi sebenarnya dibandingkan dengan semua kemungkinan kelas lainnya.

Jika ini tidak masuk akal, mari kita lihat di kode.

```
# Make prediction logits with model
y_logits = model_4(X_blob_test.to(device))

# Perform softmax calculation on logits across dimension 1 to get
prediction probabilities
y_pred_probs = torch.softmax(y_logits, dim=1)
print(y_logits[:5])
print(y_pred_probs[:5])

tensor([[-1.2549, -0.8112, -1.4795, -0.5696],
        [ 1.7168, -1.2270,  1.7367,  2.1010],
        [ 2.2400,  0.7714,  2.6020,  1.0107],
        [-0.7993, -0.3723, -0.9138, -0.5388],
        [-0.4332, -1.6117, -0.6891,  0.6852]], device='cuda:0',
       grad_fn=<SliceBackward0>)
tensor([[0.1872, 0.2918, 0.1495, 0.3715],
        [0.2824, 0.0149, 0.2881, 0.4147],
        [0.3380, 0.0778, 0.4854, 0.0989],
        [0.2118, 0.3246, 0.1889, 0.2748],
        [0.1945, 0.0598, 0.1506, 0.5951]], device='cuda:0',
       grad_fn=<SliceBackward0>)
```

Setelah meneruskan logit melalui fungsi softmax, setiap sampel kini berjumlah 1 (atau sangat mendekati).

Mari kita periksa.

```
# Sum the first sample output of the softmax activation function
torch.sum(y_pred_probs[0])

tensor(1., device='cuda:0', grad_fn=<SumBackward0>)
```

Kita dapat memeriksa indeks mana yang memiliki nilai tertinggi menggunakan torch.argmax().

```
# Which class does the model think is *most* likely at the index 0
sample?
print(y_pred_probs[0])
print(torch.argmax(y_pred_probs[0]))

tensor([0.1872, 0.2918, 0.1495, 0.3715], device='cuda:0',
       grad_fn=<SelectBackward0>)
tensor(3, device='cuda:0')
```

#Membuat loop pelatihan dan pengujian untuk model PyTorch multi-kelas

Mari kita latih model untuk epochs=100 dan evaluasi setiap 10 epoch.

```
# Fit the model
torch.manual_seed(42)

# Set number of epochs
epochs = 100

# Put data to target device
X_blob_train, y_blob_train = X_blob_train.to(device),
y_blob_train.to(device)
X_blob_test, y_blob_test = X_blob_test.to(device),
y_blob_test.to(device)

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

    # 1. Forward pass
    y_logits = model_4(X_blob_train) # model outputs raw logits
    y_pred = torch.softmax(y_logits, dim=1).argmax(dim=1) # go from
logits -> prediction probabilities -> prediction labels
    # print(y_logits)
    # 2. Calculate loss and accuracy
    loss = loss_fn(y_logits, y_blob_train)
    acc = accuracy_fn(y_true=y_blob_train,
                      y_pred=y_pred)

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backwards
    loss.backward()

    # 5. Optimizer step
    optimizer.step()

    ### Testing
    model_4.eval()
    with torch.inference_mode():
      # 1. Forward pass
      test_logits = model_4(X_blob_test)
      test_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
      # 2. Calculate test loss and accuracy
      test_loss = loss_fn(test_logits, y_blob_test)
      test_acc = accuracy_fn(y_true=y_blob_test,
                             y_pred=test_pred)

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

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

#Membuat dan mengevaluasi prediksi dengan model kelas jamak PyTorch

mari kita membuat beberapa prediksi dan memvisualisasikannya.

```
# Make predictions
model_4.eval()
with torch.inference_mode():
    y_logits = model_4(X_blob_test)

# View the first 10 predictions
y_logits[:10]
```

```
tensor([[  4.3377,  10.3539, -14.8948,  -9.7642],
        [  5.0142, -12.0371,   3.3860,  10.6699],
        [ -5.5885, -13.3448,  20.9894,  12.7711],
        [  1.8400,   7.5599,  -8.6016,  -6.9942],
        [  8.0726,   3.2906, -14.5998,  -3.6186],
        [  5.5844, -14.9521,   5.0168,  13.2890],
        [ -5.9739, -10.1913,  18.8655,   9.9179],
        [  7.0755,  -0.7601,  -9.5531,   0.1736],
        [ -5.5918, -18.5990,  25.5309,  17.5799],
        [  7.3142,   0.7197, -11.2017,  -1.2011]], device='cuda:0')
```

Mari kita ubah logit prediksi model kita menjadi probabilitas prediksi (menggunakan torch.softmax()) lalu ke label prediksi (dengan mengambil argmax() dari setiap sampel).

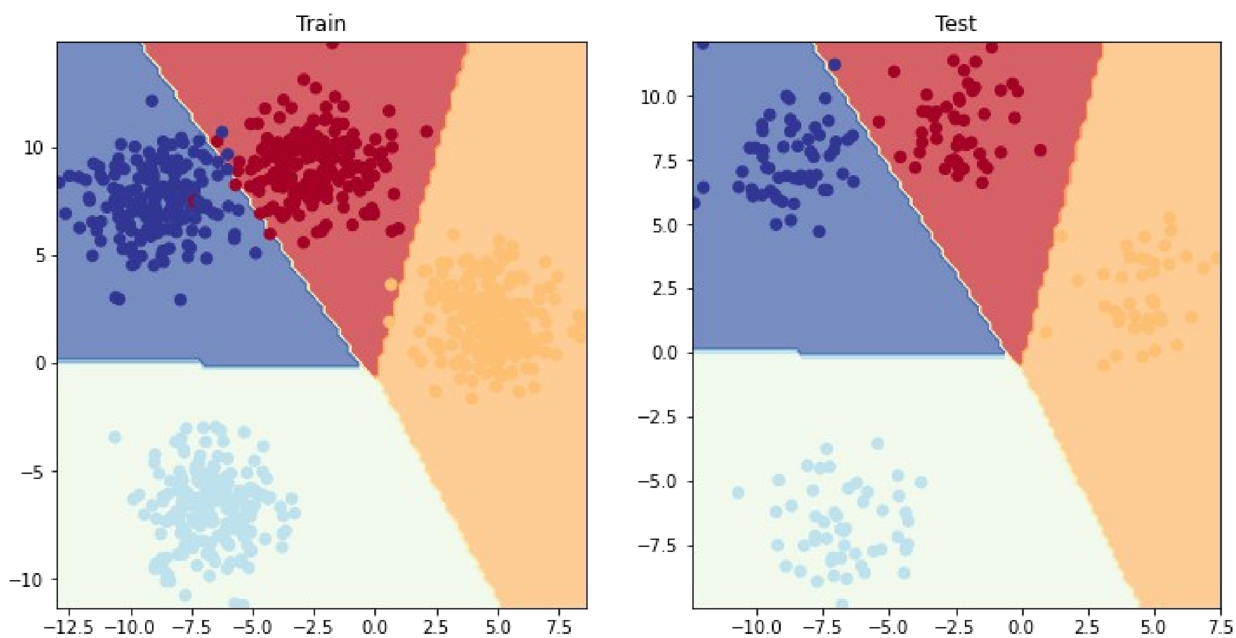
```
# Turn predicted logits in prediction probabilities
y_pred_probs = torch.softmax(y_logits, dim=1)
```

```python
# Turn prediction probabilities into prediction labels
y_preds = y_pred_probs.argmax(dim=1)

# Compare first 10 model preds and test labels
print(f"Predictions: {y_preds[:10]}\nLabels: {y_blob_test[:10]}")
print(f"Test accuracy: {accuracy_fn(y_true=y_blob_test,
y_pred=y_preds)}%")
```

```
Predictions: tensor([1, 3, 2, 1, 0, 3, 2, 0, 2, 0], device='cuda:0')
Labels: tensor([1, 3, 2, 1, 0, 3, 2, 0, 2, 0], device='cuda:0')
Test accuracy: 99.5%
```

Mari kita visualisasikannya dengan plot_decision_boundary(), ingat karena data kita ada di GPU, kita harus memindahkannya ke CPU untuk digunakan dengan matplotlib (plot_decision_boundary() melakukan ini secara otomatis untuk kita).

```python
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Train")
plot_decision_boundary(model_4, X_blob_train, y_blob_train)
plt.subplot(1, 2, 2)
plt.title("Test")
plot_decision_boundary(model_4, X_blob_test, y_blob_test)
```



#Lebih banyak metrik evaluasi klasifikasi

Mari kita coba metrik torchmetrics.Accuracy.

```python
try:
    from torchmetrics import Accuracy
except:
    !pip install torchmetrics==0.9.3 # this is the version we're using in this notebook (later versions exist here: https://torchmetrics.readthedocs.io/en/stable/generated/CHANGELOG.html#changelog)
    from torchmetrics import Accuracy

# Setup metric and make sure it's on the target device
torchmetrics_accuracy = Accuracy(task='multiclass', num_classes=4).to(device)

# Calculate accuracy
torchmetrics_accuracy(y_preds, y_blob_test)
```

```
tensor(0.9950, device='cuda:0')
```