Nama : Nabilah Salwa

NIM : 1103204060

UAS Machine Learning

02. PyTorch Classification Exercise

```
# Check for GPU
!nvidia-smi
```

Kode di atas menggunakan perintah !nvidia-smi untuk mengecek informasi tentang GPU yang tersedia pada sistem. Perintah ini menjalankan utilitas NVIDIA System Management Interface (nvidia-smi) dari baris perintah atau terminal, yang memberikan informasi terperinci tentang kartu grafis NVIDIA yang terpasang pada sistem.

```
# Import torch
import torch

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

Kode di atas bertujuan untuk menentukan dan menampilkan perangkat (device) yang akan digunakan selama eksekusi kode. Pertama, modul torch diimpor, yang merupakan inti dari PyTorch, sebuah kerangka kerja deep learning. Selanjutnya, dengan menggunakan torch.cuda.is_available(), kita memeriksa ketersediaan GPU pada sistem. Jika GPU tersedia, variabel device diatur sebagai "cuda", mengindikasikan penggunaan GPU untuk eksekusi. Jika tidak, variabel device diatur sebagai "cpu", menunjukkan penggunaan CPU.

1. Make a binary classification dataset with Scikit-Learn's make_moons() function.

```
from sklearn.datasets import make_moons

NUM_SAMPLES = 1000
RANDOM_SEED = 42

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

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

Kode di atas menggunakan modul `make_moons` dari scikit-learn untuk membuat dataset sintetis yang terdiri dari dua kelas yang membentuk pola seperti dua bulan sabit. Dengan mengatur parameter seperti jumlah sampel (`n_samples`), tingkat kebisingan (`noise`), dan seed acak (`random_state`), kita dapat mengontrol karakteristik dari dataset yang dihasilkan.

Pertama, variabel `NUM_SAMPLES` diatur ke nilai 1000 dan `RANDOM_SEED` ke nilai 42. Selanjutnya, fungsi `make_moons` dipanggil dengan parameter yang telah diatur sebelumnya untuk membuat dataset sintetis. Hasilnya adalah dua array NumPy, `X` dan `y`. Array `X` berisi koordinat dua dimensi untuk setiap sampel dalam dataset, sementara array `y` berisi label kelas yang menunjukkan apakah sampel tersebut termasuk dalam kelas 0 atau kelas 1.

Percobaan `X[:10]` dan `y[:10]` digunakan untuk mencetak 10 sampel pertama dari array `X` dan `y`, memberikan gambaran singkat tentang struktur dan nilai-nilai yang terdapat dalam dataset yang telah dibuat. Dengan menggunakan modul ini, kita dapat dengan mudah menghasilkan dataset sintetis untuk keperluan eksperimen atau latihan dalam pembelajaran mesin.

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

Kode di atas menggunakan modul `pandas` untuk mengubah dataset yang telah dibuat sebelumnya menjadi sebuah DataFrame. DataFrame adalah struktur data tabular yang kuat dalam Python yang memungkinkan pengolahan data dengan lebih mudah. Pertama, modul `pandas` diimpor sebagai `pd`. Selanjutnya, dataset yang terdiri dari array `X` dan `y` diubah menjadi sebuah DataFrame dengan menggunakan `pd.DataFrame(...)`. Dalam proses ini, kolom DataFrame diberi label "X0" dan "X1" untuk koordinat dua dimensi dari setiap sampel, serta label "y" untuk menandakan kelas.

Percobaan terakhir, `data_df.head()`, digunakan untuk mencetak lima baris pertama dari DataFrame yang telah dibuat. Ini memberikan gambaran singkat tentang struktur dan nilai-nilai dalam DataFrame, memungkinkan pengguna untuk dengan cepat memeriksa dan memahami format data. Dengan menggunakan DataFrame, analisis dan manipulasi data dapat dilakukan dengan mudah menggunakan berbagai fitur yang disediakan oleh modul `pandas`.

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

Kode di atas menggunakan modul `matplotlib.pyplot` untuk membuat visualisasi scatter plot dari dataset yang telah dibuat. Pertama, modul `matplotlib.pyplot` diimpor sebagai `plt`. Selanjutnya, fungsi `plt.scatter(...)` digunakan untuk membuat scatter plot. Dalam fungsi `plt.scatter(...)`, `X[:, 0]` dan `X[:, 1]` mewakili koordinat dua dimensi dari setiap sampel dalam dataset, dan `c=y` menentukan penggunaan label kelas `y` untuk memberikan warna pada setiap titik dalam plot. Dengan menggunakan `cmap=plt.cm.RdYlBu`, kita memilih peta warna yang akan digunakan, di sini menggunakan peta warna dari merah (Rd) ke kuning (Yl) hingga biru (Bu).

Hasilnya adalah scatter plot di mana setiap titik merepresentasikan satu sampel dari dataset, dengan warna yang menunjukkan kelasnya. Visualisasi ini memberikan gambaran intuitif tentang pola dan distribusi data pada dimensi dua, memudahkan pemahaman tentang karakteristik dataset sintetis yang telah dihasilkan sebelumnya.

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

# Split the data into train and test sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=RANDOM_SEED)

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

Kode di atas bertujuan untuk mengubah data dari tipe data NumPy menjadi tensor PyTorch dan kemudian membaginya menjadi set pelatihan dan pengujian. Pertama, array NumPy `X` dan `y` diubah menjadi tensor PyTorch dengan menggunakan `torch.tensor(...)` dan menetapkan tipe data (`dtype`) sebagai `torch.float`. Hal ini bertujuan untuk mempersiapkan data dalam format yang kompatibel dengan operasi PyTorch.

Selanjutnya, menggunakan modul `train_test_split` dari scikit-learn, dataset yang telah diubah menjadi tensor dibagi menjadi set pelatihan (`X_train`, `y_train`) dan set pengujian (`X_test`, `y_test`). Pengaturan `test_size=0.2` menunjukkan bahwa 20% dari data akan dialokasikan sebagai set pengujian, sementara 80% sisanya digunakan sebagai set pelatihan. Seed acak (`random_state=RANDOM_SEED`) digunakan untuk memastikan reproduktibilitas pemisahan data, sehingga setiap kali kode dijalankan, pembagian data akan tetap sama.

Percobaan terakhir, `len(X_train)`, `len(X_test)`, `len(y_train)`, dan `len(y_test)`, memberikan informasi tentang jumlah sampel dalam set pelatihan dan set pengujian. Hal ini memberikan pemahaman singkat tentang ukuran relatif dari kedua set data, yang dapat berguna untuk evaluasi dan pemantauan kinerja model pada kedua set tersebut.

2. Build a model by subclassing nn.Module that incorporates non-linear activation functions and is capable of fitting the data you created in 1.

```
import torch
from torch import nn

class MoonModelV0(nn.Module):
    def __init__(self, in_features, out_features, hidden_units):
        super().__init__()

        self.layer1 = nn.Linear(in_features=in_features,
                                out_features=hidden_units)
        self.layer2 = nn.Linear(in_features=hidden_units,
                                out_features=hidden_units)
        self.layer3 = nn.Linear(in_features=hidden_units,
                                out_features=out_features)
        self.relu = nn.ReLU()

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

model_0 = MoonModelV0(in_features=2,
                      out_features=1,
                      hidden_units=10).to(device)
model_0
```

Kode di atas mendefinisikan sebuah model neural network sederhana dengan menggunakan modul PyTorch, terutama dari modul `torch.nn`. Model ini disebut `MoonModelV0` dan memiliki arsitektur dengan tiga lapisan linear (fully connected layers) dan fungsi aktivasi ReLU di antara lapisan-lapisan tersebut.

Dalam konstruktor `__init__`, model ini menerima tiga parameter: `in_features` (jumlah fitur input), `out_features` (jumlah fitur output), dan `hidden_units` (jumlah unit di lapisan tersembunyi). Tiga lapisan linear (`nn.Linear`) dibuat dengan menggunakan parameter-parameter ini. Lapisan-lapisan ini akan bertanggung jawab untuk melakukan transformasi linier pada data input. Fungsi aktivasi ReLU (`nn.ReLU()`) ditempatkan di antara lapisan-lapisan tersebut untuk memasukkan unsur non-linearitas ke dalam model. Metode `forward` mendefinisikan aliran maju (forward pass) dari model, di mana data input `x` melewati melalui lapisan-lapisan dan fungsi aktivasi sesuai dengan arsitektur yang telah ditentukan.

Model ini diarahkan untuk mengembalikan output dari lapisan terakhir. Selanjutnya, sebuah instance dari `MoonModelV0` dibuat dengan mengatur parameter-parameter seperti jumlah fitur input, fitur output, dan jumlah unit di lapisan tersembunyi. Model ini juga dipindahkan ke perangkat yang telah ditentukan (`device`), baik itu CPU atau GPU. Akhirnya, model tersebut dicetak untuk memberikan gambaran singkat tentang arsitektur dan parameter-pesos yang terdapat dalam model. Model ini siap untuk dilatih dan diuji pada data yang telah diolah sebelumnya.

```
model_0.state_dict()
```

Kode di atas menggunakan metode .state_dict() pada model neural network (model_0) untuk mengakses dan mencetak state dictionary. State dictionary ini adalah sebuah koleksi yang berisi nilai-nilai dari semua parameter-pesos model, termasuk nilai-nilai yang telah diinisialisasi selama pembuatan model.

3. Setup a binary classification compatible loss function and optimizer to use when training the model built in 2.

```
loss_fn = nn.BCEWithLogitsLoss() # sigmoid layer built-in
# loss_fn = nn.BCELoss() # requires sigmoid layer
optimizer = torch.optim.SGD(params=model_0.parameters(), # parameters of model to optimize
                            lr=0.1) # learning rate
```

Kode di atas menetapkan fungsi kerugian (loss function) dan optimizer untuk model neural network yang telah dibuat sebelumnya (`model_0`). Pertama, `nn.BCEWithLogitsLoss()` dipilih sebagai fungsi kerugian. Fungsi ini merupakan kombinasi dari sigmoid layer dan binary cross-entropy loss, dan umumnya digunakan untuk tugas klasifikasi biner. Alternatifnya, `nn.BCELoss()` dapat digunakan, tetapi dalam kasus ini, memerlukan penambahan sigmoid layer secara terpisah sebelumnya. Selanjutnya, `torch.optim.SGD` dipilih sebagai optimizer, yang merupakan algoritma Stochastic Gradient Descent (SGD). Optimizer ini digunakan untuk mengoptimalkan parameter-pesos model (`model_0.parameters()`) dengan menggunakan learning rate sebesar 0.1.

Dengan menetapkan fungsi kerugian dan optimizer ini, model neural network siap untuk dilatih. Pada setiap iterasi pelatihan, model akan meminimalkan nilai kerugian dengan memperbarui parameter-pesosnya menggunakan metode backpropagation dan algoritma optimasi SGD. Langkah-langkah ini membentuk inti dari proses pelatihan model dalam konteks supervised learning.

4. Create a training and testing loop to fit the model you created in 2 to the data you created in 1.

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

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

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

Kode di atas digunakan untuk melakukan inferensi pada model neural network (`model_0`) menggunakan data pelatihan (`X_train`). Tiga informasi utama yang dicetak adalah logits (keluaran mentah model sebelum melewati fungsi aktivasi sigmoid), probabilitas prediksi (hasil aplikasi fungsi sigmoid pada logits), dan label prediksi biner (hasil pembulatan probabilitas prediksi).

Pertama, dicetak "Logits:", yang menunjukkan keluaran mentah atau tak teraktivasi dari model untuk sepuluh sampel pertama dalam set pelatihan. Keluaran ini belum melalui fungsi aktivasi sigmoid, sehingga nilainya dapat bervariasi dalam rentang apapun. Kemudian, dicetak "Pred probs:", yang menunjukkan probabilitas prediksi setelah menerapkan fungsi sigmoid pada logits. Fungsi sigmoid digunakan untuk mengubah logits menjadi nilai probabilitas antara 0 dan 1. Probabilitas ini mengindikasikan sejauh mana model yakin terhadap kelas positif (1) untuk setiap sampel. Terakhir, dicetak "Pred labels:", yang menunjukkan label biner hasil pembulatan probabilitas prediksi. Dengan menggunakan fungsi `torch.round`, probabilitas prediksi yang lebih besar dari 0.5 akan dibulatkan menjadi 1, sedangkan yang lebih kecil atau sama dengan 0.5 akan dibulatkan menjadi 0. Ini menghasilkan prediksi biner yang dapat digunakan untuk perbandingan dengan label sebenarnya dalam evaluasi model.

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

Dalam kode di atas, terdapat instalasi paket `torchmetrics` menggunakan perintah `!pip -q install torchmetrics`, yang bertujuan untuk menginstal paket tersebut jika belum terpasang. `torchmetrics` adalah suatu perpustakaan yang menyediakan berbagai metrik evaluasi yang umum digunakan dalam tugas pembelajaran mesin, termasuk perhitungan akurasi.

Selanjutnya, objek `Accuracy` dari `torchmetrics` dibuat dengan parameter `task="multiclass"` dan `num_classes=2`. `task="multiclass"` menunjukkan bahwa metrik akurasi ini digunakan untuk tugas klasifikasi multikelas, sedangkan `num_classes=2` menunjukkan jumlah kelas pada masalah ini. Objek ini juga dikirimkan ke perangkat yang digunakan, baik itu CPU atau GPU, dengan menggunakan metode `.to(device)`.

Dengan memiliki objek `acc_fn`, kita dapat menggunakannya untuk mengukur akurasi model pada suatu prediksi terhadap data tertentu. Akurasi ini umumnya digunakan sebagai metrik evaluasi untuk mengukur seberapa baik model klasifikasi dapat mengklasifikasikan data dengan benar.

```
torch.manual_seed(RANDOM_SEED)

epochs=1000

# Send data to the device
X_train, y_train = X_train.to(device), y_train.to(device)
X_test, y_test = X_test.to(device), y_test.to(device)

# Loop through the data
for epoch in range(epochs):
  ### Training
  model_0.train()

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

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

  # 3. Zero the gradients
  optimizer.zero_grad()

  # 4. Loss backward (perform backpropagation)
  loss.backward()

  # 5. Step the optimizer (gradient descent)
  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 the loss/acc
    test_loss = loss_fn(test_logits, y_test)
    test_acc = acc_fn(test_pred, y_test.int())

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

Kode di atas merupakan implementasi loop pelatihan (training loop) untuk model neural network (`model_0`) dengan menggunakan tugas klasifikasi biner. Loop ini melibatkan beberapa tahapan utama yang diperlukan dalam proses pelatihan dan evaluasi model.

Pertama, seed acak diatur menggunakan `torch.manual_seed(RANDOM_SEED)` untuk memastikan reproduktibilitas hasil. Selanjutnya, jumlah `epochs` ditentukan sebagai 1000, yang merupakan jumlah iterasi melalui seluruh set data pelatihan. Data pelatihan dan pengujian dipindahkan ke perangkat yang telah ditentukan (`device`), baik itu CPU atau GPU, dengan menggunakan `to(device)`. Selanjutnya, loop dimulai dengan iterasi melalui

setiap epoch. Pada setiap iterasi, model berada dalam mode pelatihan (`model_0.train()`) untuk mengaktifkan dropout atau layer-layer yang bersifat training-specific.

Proses pelatihan dimulai dengan forward pass, di mana data pelatihan (`X_train`) melewati model untuk menghasilkan logits (keluaran mentah model sebelum aktivasi sigmoid). Logits ini kemudian diubah menjadi probabilitas menggunakan fungsi sigmoid, dan prediksi akhir dibulatkan menjadi label biner. Selanjutnya, dilakukan perhitungan loss menggunakan fungsi kerugian yang telah ditetapkan (`loss_fn`), dan akurasi dihitung menggunakan metrik akurasi (`acc_fn`). Gradients diatur ulang menjadi nol (`optimizer.zero_grad()`), dan dilakukan backpropagation (`loss.backward()`) untuk menghitung dan menyesuaikan gradients model. Terakhir, model diupdate menggunakan optimizer (`optimizer.step()`). Selanjutnya, model diubah ke mode evaluasi (`model_0.eval()`) untuk mematikan dropout atau layer-layer yang bersifat training-specific. Dalam konteks evaluasi, model diuji pada set pengujian (`X_test`), dan loss serta akurasi dihitung untuk evaluasi kinerja model pada data yang tidak terlihat sebelumnya.

Pesan dicetak setiap 100 epoch untuk memantau perkembangan pelatihan dan evaluasi. Hasil akhir dari proses ini adalah model yang telah dilatih dan dinilai pada dataset yang telah diolah sebelumnya.

5. Make predictions with your trained model and plot them using the plot_decision_boundary() function created in this notebook.

```
# Plot the model predictions

import numpy as np

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

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

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

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

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

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

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

Kode di atas mendefinisikan fungsi `plot_decision_boundary` yang digunakan untuk memvisualisasikan batas keputusan (decision boundary) dari model pada data dua dimensi.

Fungsi ini memanfaatkan modul `matplotlib.pyplot` dan `numpy` untuk menghasilkan plot yang menggambarkan area keputusan yang dihasilkan oleh model pada rentang nilai input tertentu.

Pertama, fungsi menerima tiga parameter: `model` (model neural network), `X` (data input), dan `y` (label target). Semua parameter ini diubah ke perangkat CPU untuk memastikan konsistensi dengan modul `numpy` dan `matplotlib.pyplot`. Selanjutnya, rentang nilai input (`xx` dan `yy`) dihasilkan menggunakan fungsi `np.meshgrid` pada data `X`. Sebuah grid yang padat dari nilai-nilai dalam rentang data input diperoleh untuk digunakan dalam plot. Fungsi kemudian membuat features (`X_to_pred_on`) dengan menggabungkan nilai-nilai `xx` dan `yy` secara terpilih. Selanjutnya, model dievaluasi pada features yang baru dibuat untuk menghasilkan logits. Terakhir, fungsi menyesuaikan prediksi sesuai dengan tipe tugas klasifikasi, apakah itu multi-class atau binary. Untuk multi-class, fungsi menggunakan `torch.softmax` dan `argmax` untuk mendapatkan label prediksi, sedangkan untuk binary, menggunakan `torch.round` dan `sigmoid`. Hasil prediksi diubah ke dalam bentuk yang sesuai dengan grid (`y_pred`) dan diplot sebagai kontur keputusan menggunakan `plt.contourf`. Data asli (`X`) juga diplot sebagai titik-titik pada plot menggunakan `plt.scatter`.

Dengan menggunakan fungsi ini, kita dapat memvisualisasikan dengan jelas bagaimana model mengambil keputusan pada rentang nilai input tertentu, membantu dalam pemahaman dan interpretasi kinerja model pada data dua dimensi.

```
# 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)
```

Kode di atas menggunakan modul `matplotlib.pyplot` untuk membuat dua subplot pada satu gambar, masing-masing menunjukkan batas keputusan (decision boundary) dari model pada set data pelatihan (`X_train`, `y_train`) dan set data pengujian (`X_test`, `y_test`).

Pertama, gambar yang akan dihasilkan diberi ukuran (figsize) 12x6 dengan menggunakan `plt.figure(figsize=(12, 6))`. Subplot pertama dibuat dengan judul "Train" menggunakan `plt.subplot(1, 2, 1)`. Fungsi `plot_decision_boundary` kemudian dipanggil untuk menggambarkan batas keputusan pada data pelatihan dengan menggunakan model `model_0`. Setelah itu, subplot kedua dibuat dengan judul "Test" menggunakan `plt.subplot(1, 2, 2)`, dan kembali fungsi `plot_decision_boundary` dipanggil untuk menggambarkan batas keputusan pada data pengujian.

Hasilnya adalah sebuah gambar yang memuat dua subplot, masing-masing menunjukkan batas keputusan dari model pada dua set data yang berbeda. Dengan visualisasi ini, kita dapat memperoleh wawasan yang lebih baik tentang sejauh mana model dapat memisahkan kelas pada kedua set data tersebut. Hal ini dapat membantu dalam mengevaluasi kinerja model dan memahami cara model membuat keputusan pada kelas-kelas yang berbeda.

6. Replicate the Tanh (hyperbolic tangent) activation function in pure PyTorch.

```
tensor_A = torch.arange(-100, 100, 1)
plt.plot(tensor_A)
```

Kode di atas menggunakan modul `matplotlib.pyplot` untuk membuat sebuah plot garis yang menampilkan nilai-nilai dari sebuah tensor yang dibuat menggunakan PyTorch. Tensor `tensor_A` dibuat dengan menggunakan fungsi `torch.arange(-100, 100, 1)`, yang menghasilkan sebuah tensor berisi nilai-nilai dari -100 hingga 99 dengan selang satu.

Selanjutnya, fungsi `plt.plot(tensor_A)` digunakan untuk membuat plot garis dengan nilai-nilai dari tensor sebagai sumbu Y. Sumbu X akan mengikuti indeks dari elemen-elemen dalam tensor, yang secara default dimulai dari 0. Sehingga, plot ini akan menunjukkan bagaimana nilai-nilai dalam tensor `tensor_A` berubah secara sekuensial.

Hasilnya adalah plot garis yang menampilkan perubahan nilai-nilai dalam tensor dari -100 hingga 99. Dengan menggunakan visualisasi ini, kita dapat dengan cepat memahami pola atau tren dalam distribusi nilai-nilai tensor tersebut.

```
plt.plot(torch.tanh(tensor_A))
```

Kode di atas menggunakan modul matplotlib.pyplot untuk membuat plot garis yang menampilkan nilai-nilai dari fungsi tangen hiperbolik (tanh) yang diterapkan pada sebuah tensor PyTorch. Tensor tensor_A dibuat sebelumnya dengan menggunakan fungsi torch.arange(-100, 100, 1), yang menghasilkan sebuah tensor dengan nilai-nilai dari -100 hingga 99 dengan selang satu.

```
def tanh(x):
  return (torch.exp(x) - torch.exp(-x)) / (torch.exp(x) + torch.exp(-x))

plt.plot(tanh(tensor_A))
```

Kode di atas mendefinisikan fungsi `tanh(x)` yang mewakili implementasi manual dari fungsi tangen hiperbolik (tanh). Fungsi `tanh` ini diterapkan pada sebuah tensor PyTorch `tensor_A`, yang sebelumnya telah dibuat dengan menggunakan fungsi `torch.arange(-100, 100, 1)`, menghasilkan tensor dengan nilai-nilai dari -100 hingga 99 dengan selang satu.

Dalam implementasi fungsi `tanh(x)`, rumus matematika tangen hiperbolik diterapkan menggunakan operasi PyTorch, yaitu `(torch.exp(x) - torch.exp(-x)) / (torch.exp(x) +

torch.exp(-x))`. Ini mencakup perhitungan eksponensial menggunakan fungsi `torch.exp` dan operasi penjumlahan serta pengurangan. Setelah itu, hasil dari fungsi tangen hiperbolik tersebut diplot menggunakan `plt.plot(tanh(tensor_A))`, menghasilkan plot garis yang menampilkan transformasi non-linear dari distribusi nilai-nilai dalam tensor `tensor_A` sesuai dengan fungsi tangen hiperbolik.

Hasil plot ini memberikan representasi visual tentang bagaimana implementasi manual dari fungsi tangen hiperbolik dapat mengubah distribusi nilai-nilai dalam tensor, dan sejauh mana hal ini konsisten dengan hasil yang dihasilkan oleh fungsi tangen hiperbolik bawaan PyTorch (`torch.tanh`).

7. Create a multi-class dataset using the spirals data creation function from CS231n (see below for the code).

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

Kode di atas digunakan untuk membuat dataset berbentuk spiral dengan tiga kelas menggunakan metode yang diambil dari course CS231n (Convolutional Neural Networks for Visual Recognition). Dataset spiral ini dibuat dengan menggunakan parameter seperti jumlah poin per kelas (`N`), dimensi (`D`), jumlah kelas (`K`), dan seed acak (`RANDOM_SEED`).

Pertama, sebuah matriks kosong `X` dengan ukuran `(N * K, D)` dibuat untuk menyimpan data, dan sebuah array `y` dengan ukuran `(N * K,)` dibuat untuk menyimpan label kelas. Selanjutnya, dilakukan iterasi sebanyak jumlah kelas (`K`), di mana setiap kelas akan diberi label (`j`), dan `N` titik data akan dihasilkan dengan koordinat kartesian berdasarkan parameter radius (`r`) dan theta (`t`). Koordinat tersebut kemudian diubah menjadi polar menggunakan rumus `np.c_[r*np.sin(t), r*np.cos(t)]` dan disimpan dalam matriks `X` serta label kelas disimpan dalam array `y`.

Hasilnya adalah dataset spiral yang terdiri dari tiga kelas dengan poin-poin data yang membentuk pola spiral pada bidang dua dimensi. Visualisasi dataset ini dilakukan menggunakan `plt.scatter` untuk memplot titik-titik data dengan warna yang merepresentasikan label kelasnya, dan titik-titik dalam setiap kelas membentuk pola spiral yang khas.

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

# Create train and test splits
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=RANDOM_SEED)
len(X_train), len(X_test), len(y_train), len(y_test)
```

Kode di atas bertujuan untuk mengubah data dari format NumPy menjadi format PyTorch tensors. Pertama, data fitur `X` dan label kelas `y` yang sebelumnya disimpan dalam format NumPy array diubah menjadi tensors PyTorch menggunakan `torch.from_numpy()`. Selain itu, tipe data tensors juga diatur agar sesuai, yaitu fitur sebagai float32 dan label sebagai LongTensor.

Selanjutnya, data dibagi menjadi set pelatihan (`X_train`, `y_train`) dan set pengujian (`X_test`, `y_test`) menggunakan fungsi `train_test_split` dari scikit-learn. Pembagian ini dilakukan dengan mengambil 20% dari data sebagai data pengujian (`test_size=0.2`) dan menggunakan seed acak yang telah ditentukan sebelumnya (`random_state=RANDOM_SEED`).

Hasilnya adalah empat tensors PyTorch yang siap digunakan untuk melatih dan menguji model pada dataset spiral yang telah dibuat sebelumnya. Tensors ini akan digunakan sebagai input dan target output pada model machine learning yang akan dikembangkan.

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

Kode di atas bertujuan untuk menghitung akurasi model pada dataset spiral yang telah dibuat. Pertama, sebuah pustaka tambahan, yaitu `torchmetrics`, diinstal menggunakan `!pip -q install torchmetrics` karena platform Colab tidak menyertakan `torchmetrics` secara default. `torchmetrics` menyediakan berbagai metrik evaluasi, dan dalam hal ini, kita menggunakan metrik akurasi (`Accuracy`).

Setelah instalasi, objek `Accuracy` dibuat dengan parameter `task="multiclass"` dan `num_classes=3` untuk menyesuaikan tugas klasifikasi multikelas dengan tiga kelas pada dataset spiral. Objek ini kemudian dipindahkan ke perangkat yang sesuai dengan menggunakan metode `.to(device)`.

Hasilnya adalah objek `Accuracy` yang siap digunakan untuk mengukur akurasi dari model yang akan dilatih pada dataset spiral tersebut. Metrik ini dapat membantu dalam mengevaluasi sejauh mana model dapat mengklasifikasikan dengan benar pada set data yang tidak terlihat selama pelatihan.

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

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

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

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

Kode di atas mendefinisikan arsitektur model neural network untuk tugas klasifikasi pada dataset spiral. Model ini disebut `SpiralModel` dan dibangun menggunakan modul `nn.Module` dari PyTorch.

Arsitektur model terdiri dari tiga lapisan linear (`nn.Linear`) yang mengimplementasikan transformasi linier pada data input. Lapisan pertama (`linear1`) memiliki 2 neuron input (sesuai dengan dimensi fitur dataset spiral), lapisan tengah (`linear2`) memiliki 10 neuron, dan lapisan keluaran (`linear3`) memiliki 3 neuron output, sesuai dengan jumlah kelas pada dataset spiral. Fungsi aktivasi ReLU (`nn.ReLU()`) diterapkan setelah setiap lapisan linear untuk memperkenalkan non-linearitas ke dalam model. Metode `forward` mendefinisikan aliran maju (forward pass) dari model, di mana data input `x` melewati setiap lapisan linear dan fungsi aktivasi ReLU untuk menghasilkan prediksi output model. Model tersebut kemudian dipindahkan ke perangkat yang sesuai dengan menggunakan `.to(device)`. Hasilnya adalah objek `SpiralModel` yang siap digunakan untuk dilatih pada dataset spiral dengan tiga kelas.

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

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

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

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

Kode di atas bertujuan untuk menyiapkan data pada perangkat yang sesuai dengan perangkat yang akan digunakan untuk pelatihan dan pengujian model. Pertama, data pelatihan (`X_train`, `y_train`) dan data pengujian (`X_test`, `y_test`) dipindahkan ke perangkat (`device`) yang telah ditentukan sebelumnya menggunakan metode `.to(device)`.

Selanjutnya, tipe data (dtype) dari tensor-tensor yang dihasilkan, yaitu `X_train`, `X_test`, `y_train`, dan `y_test`, dicetak menggunakan `print(X_train.dtype, X_test.dtype, y_train.dtype, y_test.dtype)` untuk memastikan bahwa tipe data telah disesuaikan dengan perangkat yang digunakan. Terakhir, hasil prediksi dari model yang belum dilatih dicetak untuk memberikan pemahaman awal tentang bagaimana model menghasilkan keluaran sebelum mengalami proses pelatihan. Logit, probabilitas prediksi, dan label prediksi dari sepuluh data pertama dalam set pelatihan dicetak menggunakan `print("Logits:", model_1(X_train)[:10])`, `print("Pred probs:", torch.softmax(model_1(X_train)[:10], dim=1))`, dan `print("Pred labels:", torch.softmax(model_1(X_train)[:10], dim=1).argmax(dim=1))` masing-masing. Ini memberikan gambaran awal tentang keluaran model sebelum melibatkan proses pembelajaran.

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

Kode di atas bertujuan untuk menyiapkan fungsi kerugian (loss function) dan optimizer untuk digunakan dalam pelatihan model. Pertama, fungsi kerugian yang dipilih adalah `nn.CrossEntropyLoss()`, yang sesuai untuk tugas klasifikasi multikelas seperti pada dataset spiral yang memiliki tiga kelas. Fungsi ini menggabungkan lapisan softmax dan fungsi entropi silang (cross-entropy) dalam satu langkah. Selanjutnya, optimizer yang digunakan adalah `torch.optim.Adam` dengan parameter `lr=0.02` yang menentukan laju pembelajaran (learning rate). Optimizer ini akan digunakan untuk mengoptimalkan parameter-model selama pelatihan dengan mengurangi nilai fungsi kerugian.

Hasilnya adalah dua objek, yaitu `loss_fn` yang berisi fungsi kerugian dan `optimizer` yang berisi algoritma optimasi Adam dengan laju pembelajaran yang telah ditentukan. Objek ini akan digunakan dalam langkah-langkah pelatihan model selanjutnya.

```
# Build a training loop for the model
epochs = 1000

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

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

  # 3. optimizer zero grad
  optimizer.zero_grad()

  # 4. loss backwards
  loss.backward()

  # 5. optimizer step step step
  optimizer.step()

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

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

Kode di atas membangun suatu loop pelatihan (training loop) untuk model klasifikasi pada dataset spiral. Loop tersebut memiliki jumlah epoch sebanyak 1000, yang mengindikasikan bahwa model akan melalui seluruh dataset pelatihan sebanyak 1000 kali.

Dalam setiap epoch, langkah-langkah pelatihan dilakukan. Pertama, model ditempatkan dalam mode pelatihan dengan menggunakan metode `model_1.train()`. Selanjutnya, dilakukan forward pass untuk menghasilkan logit (nilai sebelum fungsi softmax) dan prediksi dari model. Fungsi softmax diaplikasikan pada logit untuk mendapatkan distribusi probabilitas, dan kemudian argmax diambil untuk mendapatkan label prediksi. Selanjutnya, dilakukan perhitungan fungsi kerugian (`loss_fn`) dan akurasi (`acc_fn`) menggunakan prediksi dan label sebenarnya dari data pelatihan (`y_train`). Setelah itu, dilakukan langkah-langkah optimasi: gradient optimizer direset ke nol (`optimizer.zero_grad()`), gradient dihitung menggunakan backpropagation (`loss.backward()`), dan optimizer melakukan langkah optimasi (`optimizer.step()`).

```
# 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)
```

Kode di atas bertujuan untuk menghasilkan visualisasi batas keputusan (decision boundaries) pada dataset spiral menggunakan model yang telah dilatih. Dua subplot diperlihatkan dalam suatu gambar dengan ukuran (12, 6).

Pertama, subplot pertama (`plt.subplot(1, 2, 1)`) menggambarkan batas keputusan pada dataset pelatihan. Fungsi `plot_decision_boundary` dipanggil dengan model yang telah dilatih (`model_1`), dan data pelatihan (`X_train`, `y_train`) sebagai parameter. Visualisasi ini memberikan gambaran tentang bagaimana model mengklasifikasikan data pada saat pelatihan. Selanjutnya, subplot kedua (`plt.subplot(1, 2, 2)`) menyajikan batas keputusan pada dataset pengujian. Kembali, fungsi `plot_decision_boundary` dipanggil dengan model yang sama (`model_1`), tetapi dengan data pengujian (`X_test`, `y_test`) sebagai parameter. Ini memberikan pemahaman tentang sejauh mana model dapat generalisasi dan menghasilkan keputusan pada data yang belum pernah dilihat selama pelatihan.

Keseluruhan visualisasi ini membantu dalam mengevaluasi performa model pada kedua dataset, dan dapat memberikan wawasan tentang seberapa baik model dapat mengklasifikasikan data pada domain yang tidak terlihat selama proses pelatihan.