Classification Exercises

The following is a template for PyTorch Classification exercises.
It's only starter code and it's your job to fill in the blanks.
Because of the flexibility of PyTorch, there may be more than one way to answer the question.
Don't worry about trying to be *right* just try writing code that suffices the question.

Resources

In [1]:
```
# Import torch
import torch

# Setup device agnostic code


# Setup random seed
RANDOM_SEED = 42
```

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

- For consistency, the dataset should have 1000 samples and a `random_state=42`.
- Turn the data into PyTorch tensors.
- Split the data into training and test sets using `train_test_split` with 80% training and 20% testing.

In [2]:
```
# Create a dataset with Scikit-Learn's make_moons()
from sklearn.datasets import make_moons
```

In [3]:
```
# Turn data into a DataFrame
import pandas as pd
```

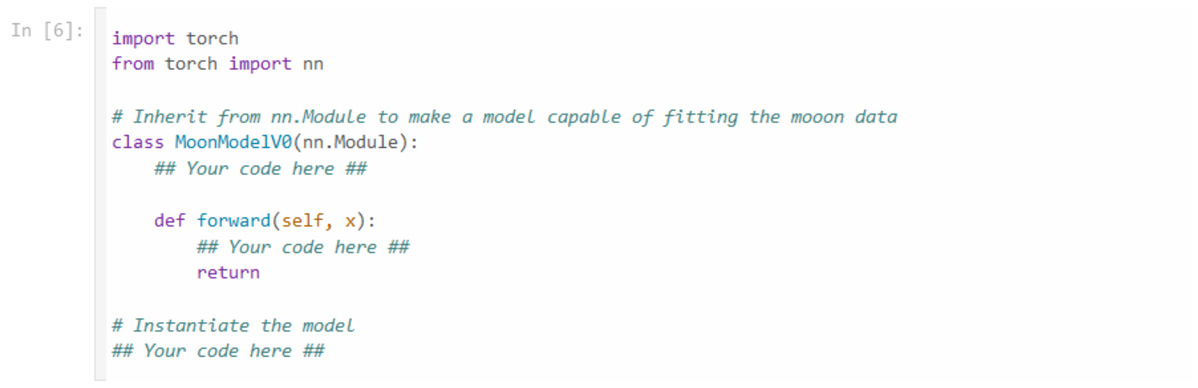
In [4]:
```
# Visualize the data on a scatter plot
import matplotlib.pyplot as plt
```

In [5]:
```
# Turn data into tensors of dtype float


# Split the data into train and test sets (80% train, 20% test)
from sklearn.model_selection import train_test_split
```

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.

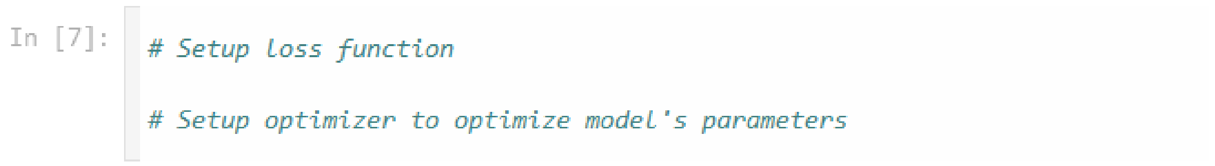
- Feel free to use any combination of PyTorch layers (linear and non-linear) you want.

import torch from torch import nn



```
In [6]:
import torch
from torch import nn

# Inherit from nn.Module to make a model capable of fitting the mooon data
class MoonModelV0(nn.Module):
    ## Your code here ##

    def forward(self, x):
        ## Your code here ##
        return

# Instantiate the model
## Your code here ##
```

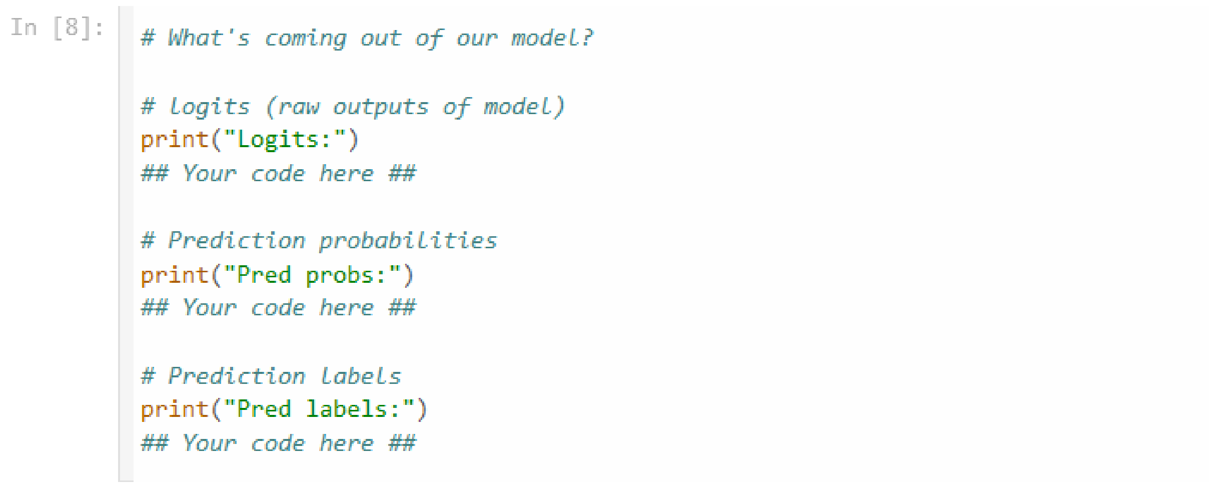
3. Setup a binary classification compatible loss function and optimizer to use when training the model built in 2.



```
In [7]:
# Setup loss function

# Setup optimizer to optimize model's parameters
```

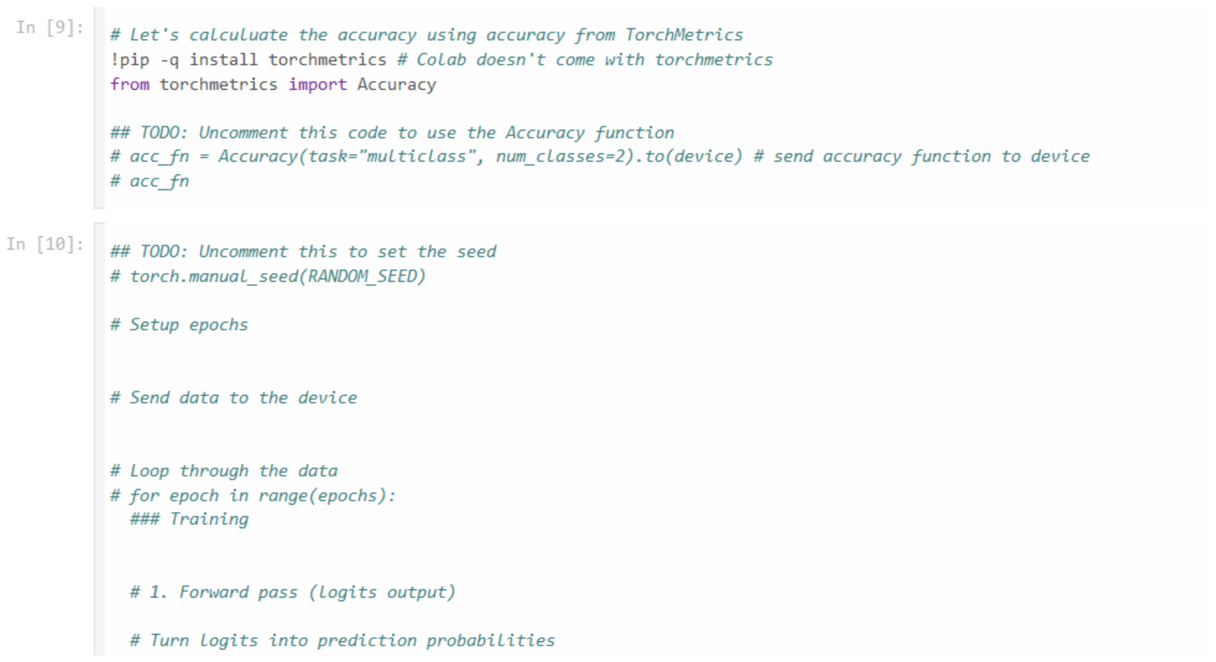
4. Create a training and testing loop to fit the model you created in 2 to the data you created in 1.

- Do a forward pass of the model to see what's coming out in the form of logits, prediction probabilities and labels.
- To measure model accuray, you can create your own accuracy function or use the accuracy function in TorchMetrics.
- Train the model for long enough for it to reach over 96% accuracy.
- The training loop should output progress every 10 epochs of the model's training and test set loss and accuracy.



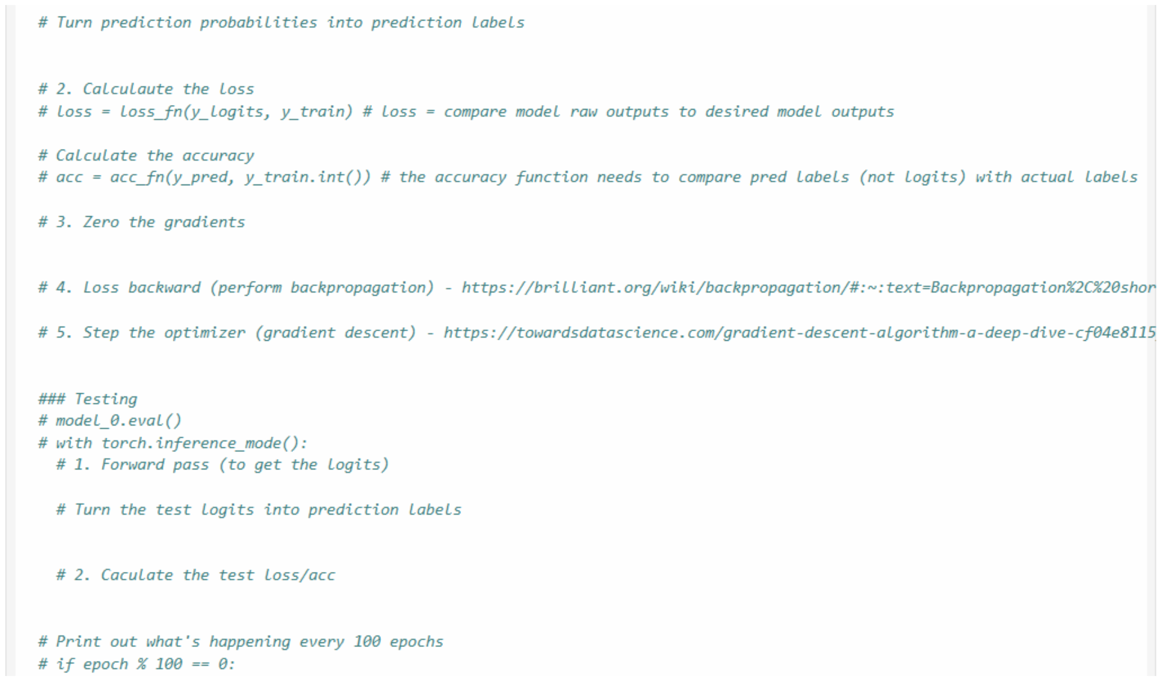
```
In [8]:
# What's coming out of our model?

# logits (raw outputs of model)
print("Logits:")
## Your code here ##

# Prediction probabilities
print("Pred probs:")
## Your code here ##

# Prediction labels
print("Pred labels:")
## Your code here ##
```

```
Logits:
Pred probs:
Pred labels:
```



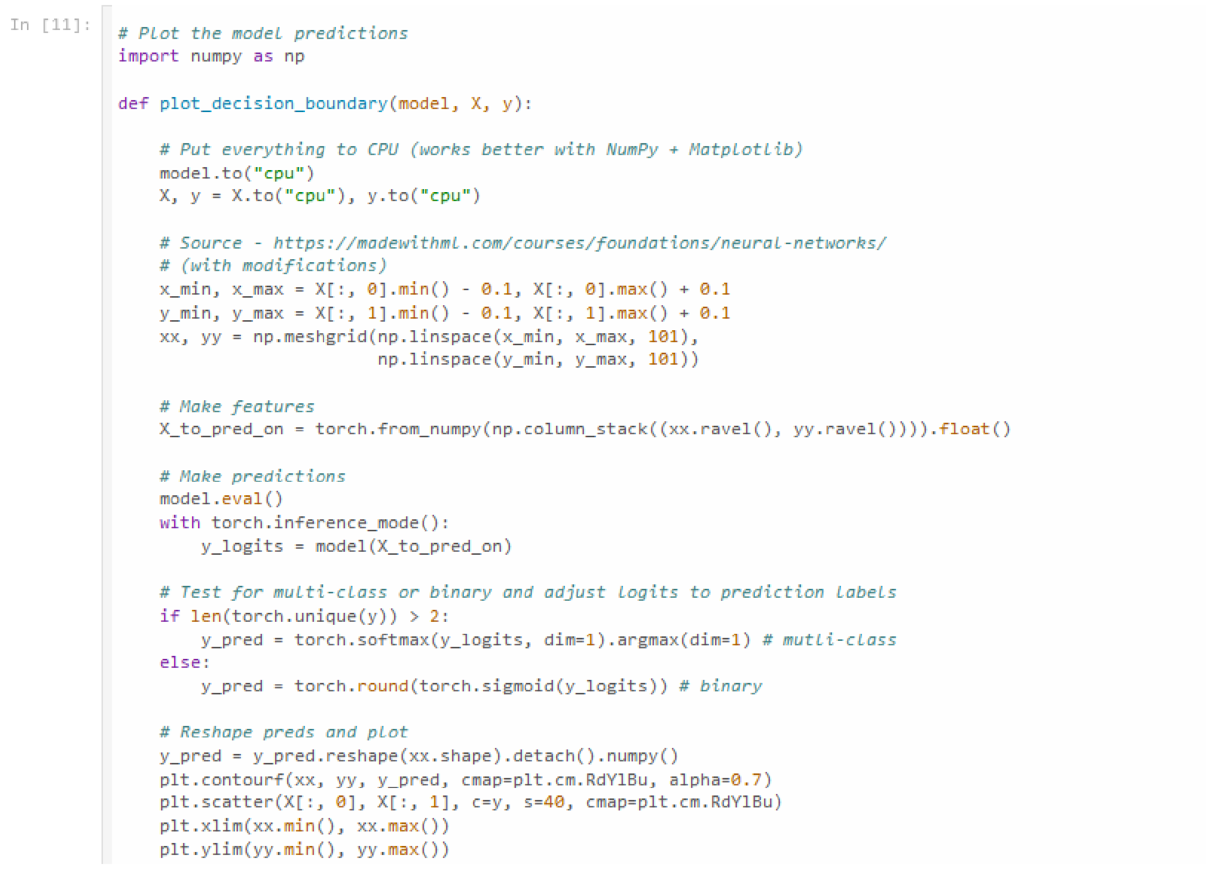
```
In [9]:  # Let's calculuate the accuracy using accuracy from TorchMetrics
         !pip -q install torchmetrics # Colab doesn't come with torchmetrics
         from torchmetrics import Accuracy

         ## TODO: Uncomment this code to use the Accuracy function
         # acc_fn = Accuracy(task="multiclass", num_classes=2).to(device) # send accuracy function to device
         # acc_fn
```

```
In [10]: ## TODO: Uncomment this to set the seed
         # torch.manual_seed(RANDOM_SEED)

         # Setup epochs


         # Send data to the device


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


           # 1. Forward pass (logits output)

           # Turn logits into prediction probabilities
```

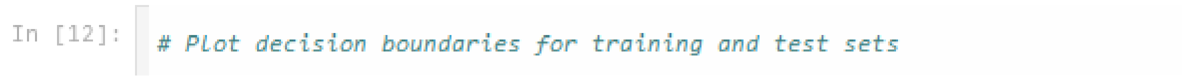


```
# Turn prediction probabilities into prediction labels


# 2. Calculaute the loss
# loss = loss_fn(y_logits, y_train) # loss = compare model raw outputs to desired model outputs

# Calculate the accuracy
# 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


# 4. Loss backward (perform backpropagation) - https://brilliant.org/wiki/backpropagation/#:~:text=Backpropagation%2C%20shor

# 5. Step the optimizer (gradient descent) - https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115


### Testing
# model_0.eval()
# with torch.inference_mode():
  # 1. Forward pass (to get the logits)

  # Turn the test logits into prediction labels


  # 2. Caculate the test loss/acc


# Print out what's happening every 100 epochs
# if epoch % 100 == 0:
```

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



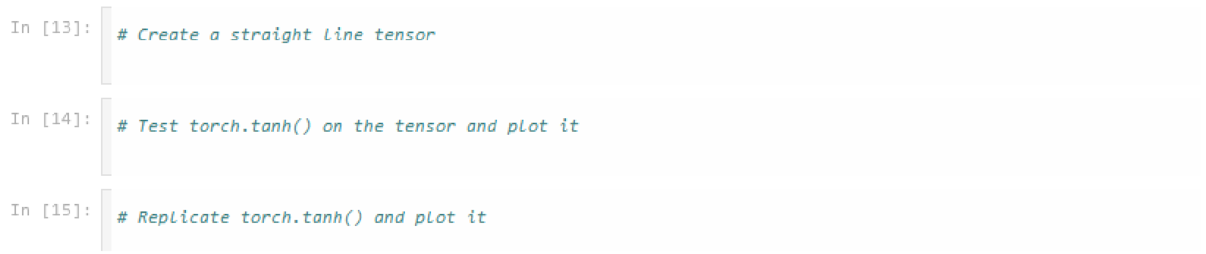
```
In [11]: # Plot the model predictions
         import numpy as np

         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())
```



```
In [12]: # Plot decision boundaries for training and test sets
```

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

- Feel free to reference the ML cheatsheet website for the formula.

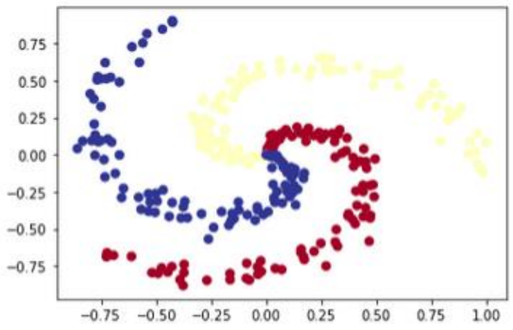


```
In [13]: # Create a straight line tensor

In [14]: # Test torch.tanh() on the tensor and plot it

In [15]: # Replicate torch.tanh() and plot it
```

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

- Split the data into training and test sets (80% train, 20% test) as well as turn it into PyTorch tensors.
- Construct a model capable of fitting the data (you may need a combination of linear and non-linear layers).
- Build a loss function and optimizer capable of handling multi-class data (optional extension: use the Adam optimizer instead of SGD, you may have to experiment with different values of the learning rate to get it working).
- Make a training and testing loop for the multi-class data and train a model on it to reach over 95% testing accuracy (you can use any accuracy measuring function here that you like) - 1000 epochs should be plenty.
- Plot the decision boundaries on the spirals dataset from your model predictions, the `plot_decision_boundary()` function should work for this dataset too.

```
In [16]:
# Code for creating a spiral dataset from CS231n
import numpy as np
import matplotlib.pyplot as plt
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()
```



```
In [17]:
# Turn data into tensors
import torch
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
```

```
In [18]:
# Let's calculuate the accuracy for when we fit our model
!pip -q install torchmetrics # colab doesn't come with torchmetrics
from torchmetrics import Accuracy

## TODO: uncomment the two lines below to send the accuracy function to the device
# acc_fn = Accuracy(task="multiclass", num_classes=4).to(device)
# acc_fn
```

```
In [19]:
# Prepare device agnostic code
# device = "cuda" if torch.cuda.is_available() else "cpu"

# Create model by subclassing nn.Module



# Instantiate model and send it to device
```

```
In [20]:
# Setup data to be device agnostic


# Print out first 10 untrained model outputs (forward pass)
print("Logits:")
## Your code here ##

print("Pred probs:")
## Your code here ##

print("Pred labels:")
print("Pred labels:")
## Your code here ##
```

```
Logits:
Pred probs:
Pred labels:
```

In [21]:
```
# Setup loss function and optimizer
# loss_fn =
# optimizer =
```

In [22]:
```
# Build a training loop for the model

# Loop over data


  ## Training

  # 1. Forward pass


  # 2. Calculate the loss


  # 3. Optimizer zero grad


  # 4. Loss backward


  # 5. Optimizer step


  ## Testing


    # 1. Forward pass

    # 2. Caculate loss and acc

  # Print out what's happening every 100 epochs
```

In [23]:
```
# Plot decision boundaries for training and test sets
```