Assignment 1: Classification of Iris Using MLP

```
# [THIS IS READ-ONLY]
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
```

```
# [THIS IS READ-ONLY]
#
# load the iris dataset as a dataframe.
#

df = pd.read_csv('my_data/iris.csv')
```

Unit 1: constructing PyTorch datasets

In this unit, you are to implement `make_dataset`.

Hint:

- Learn how to use TensorDataset.
- You are to extract the features and target from the dataframe, and construct a TensorDataset object.
- Refer to https://pytorch.org/docs/stable/data.html

Note:

- Make sure the features are using `dtype=torch.float32`.
- Make sure the targets are using `dtype=torch.int64`.

```
# [THIS IS READ-ONLY]
import torch
```

```
from torch.utils.data import (
    DataLoader,
    TensorDataset,
    Dataset
)
```

```
# [YOUR WORK HERE]
# @workUnit

def make_dataset(df: pd.DataFrame) -> Dataset:
    ...
```

```
# [THIS IS READ-ONLY]
# @check
# @title: inspecting training dataset

train_dataset = make_dataset(df)
(x, y) = train_dataset[0]
print(f"First input {x}.  Its dtype must be {x.dtype}.")
print(f"First output {y}.  Its dtype must be {y.dtype}.")
```

Unit 2: constructing PyTorch dataloader

In this unit, you are to implement `make_dataloader` which converts `Dataset` to `DataLoader` with specified `batch_size` and `shuffle` flag.

Refer to https://pytorch.org/tutorials/beginner/basics/data_tutorial.html#preparing-your-data-for-training-with-dataloaders

```
# [YOUR WORK HERE]
# @workUnit

def make_dataloader(dataset: Dataset, batch_size:int, shuffle:bool) -> DataLoader:
    ...
```

```
# [THIS IS READ-ONLY]
# @check
# @title: inspect training dataloader
train_dataloader = make_dataloader(train_dataset, shuffle=False, batch_size=5)
first_batch = next(iter(train_dataloader))
```

```
first_batch
```

Unit 3: Linear classifier

In this unit, you are to implement a neural network module that performs simple linear classification. Namely,

$$y_{\text{pred}} = xW + b$$

Hint:

- Use the built-in `nn.Linear(...)` as a layer in your module.
- You must name the attribute in the `LinearClassifier` as `linear` for you to pass the checkpoint.

```
# [THIS IS READ-ONLY]
from torch import nn
from torchinfo import summary
```

```
# [YOUR WORK HERE]
# @workUnit
# initialize the `linear` attribute
# implement the forward(...) method

class LinearClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = ...

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

```
# [THIS IS READ-ONLY]
# @check
# @title: architecture of linear classifier

m = LinearClassifier()
summary(m, input_size=(32,4))
```

Unit 4: Training loop

In this unit, you are given a function that performs the training loop.

You will use the provided training loop to train the linear classifier and inspect the accuracy.

```
# [THIS IS READ-ONLY]
from torch.optim import (Optimizer, Adam)
from torch.nn.functional import cross_entropy
from torchmetrics import Accuracy
```

```
# [THIS IS READ-ONLY]
def train(model: nn.Module, optimizer: Optimizer, dataloader: DataLoader, epochs: int):
    history = []
    accuracy = Accuracy(task='multiclass', num_classes=3)
    for epoch in range(epochs):
        for (x, target) in dataloader:
            pred = model(x)
            loss = cross_entropy(pred, target)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        metrics = {
            'epoch': epoch,
            'loss': loss.item(),
            'acc': accuracy(pred, target).item()
        }
        if epoch % (epochs // 10) == 0:
            print("{epoch}: loss={loss:.4f}, acc={acc:.2f}".format(**metrics))
        history.append(metrics)
    return pd.DataFrame(history)
```

```
# [YOUR WORK HERE]
# @workUnit

dataloader = make_dataloader(train_dataset, shuffle=False, batch_size=32)
linearclassifier = LinearClassifier()
optimizer = Adam(...)

history_linear = train(linearclassifier, optimizer, dataloader, 100)
```

```
# [THIS IS READ-ONLY]
# @check
# @title: ensure linear classifier performance

print("linear classifier acc > 50%?", history_linear.acc.iloc[-1] > 0.5)
print("linear classifier acc < 90%?", history_linear.acc.iloc[-1] < 0.9)
```

```
# [THIS IS READ-ONLY]
#
# Plotting the loss function and accuracy
#
import matplotlib.pyplot as plt
fig, axes = plt.subplots(ncols=2, figsize=(10,4))
history_linear.loss.plot.line(ax=axes[0])
history_linear.acc.plot.line(ax=axes[1]);
```

Unit 5: MLP with hidden layer

In this section, you are to implement a multi-layer perceptron (MLP) with a single hidden layer of 100 neurons.

Note: You must name the attributes as follows.

- `linear1`: the hidden layer with 100 neurons.
- `act1`: the ReLU activation function.
- `output`: the output layer that outputs the logits over the 3 categories.

Refer to: http://db.science.uoit.ca/csci4050u/2_fitting_2d/03_mlp.html

```
# [YOUR WORK HERE]
# @workUnit

class MLPClassifier(nn.Module):
    ...
```

```
# [THIS IS READ-ONLY]
# @check
# @title: architecture of MLP classifier

m = MLPClassifier()
summary(m, input_size=(32,4))
```

```
# [THIS IS READ-ONLY]
#
# training the MLP model
#
mlp = MLPClassifier()
optimizer = Adam(mlp.parameters())
dataloader = make_dataloader(train_dataset, shuffle=False, batch_size=32)

history_mlp = train(mlp, optimizer, dataloader, 100)
```

```
# [THIS IS READ-ONLY]
# @check
# @title: ensure MLP performance

history_mlp.acc.iloc[-1] > 0.9
```

```
# [THIS IS READ-ONLY]
#
# Plotting the loss function and accuracy
#
import matplotlib.pyplot as plt
fig, axes = plt.subplots(ncols=2, figsize=(10,4))
history_mlp.loss.plot.line(ax=axes[0])
history_mlp.acc.plot.line(ax=axes[1]);
```

Unit 6: Explaining MLP action

In this unit, we will explore ways to uncover how deep neural networks organize data by generating 2D hidden features and visualize the generated features as a scatter plot.

You must create a `MLP2DClassifier` neural network consisting of the following layers:

- `linear1` is a linear layer with 100 neurons.
- `act1` is the ReLU activation function for the `linear1` layer.
- `linear2` is a linear layer that maps the 100 dimensional hidden feature to 2 dimensional feature.
- `output` is a linear layer that maps the 2D feature to 3D logits.

It is the output of `linear2` layer provides insight into how `x2 = act1(linear1(x))` works.

Your implementation of `MLP2DClassifier` will have an additional method `hiddenFeature(x)` that will return the output of `linear2`.

```
# [YOUR WORK HERE]
# @workUnit

class MLP2DClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = ...
        self.act1 = ...
        self.linear2 = ...
        self.output = ...
    def forward(self, x):
        ...
    def hiddenFeature(self, x):
        ...
```

```
# [THIS IS READ-ONLY]
# @check
# @title: MLP2DClassifier architecture

m = MLP2DClassifier()
summary(m, input_size=(32, 4))
```

```
# [THIS IS READ-ONLY]
mlp2 = MLP2DClassifier()
optimizer = Adam(mlp2.parameters())
dataloader = make_dataloader(train_dataset, shuffle=False, batch_size=32)

history_mlp2 = train(mlp2, optimizer, dataloader, 100)
```

```
# [THIS IS READ-ONLY]
# @check
# @title: ensure MLP2 performance

history_mlp2.acc.iloc[-1] > 0.9
```

```
# [THIS IS READ-ONLY]
#
# compute the hidden features for the first 100 training samples
#
```

```
(x, target) = train_dataset[0:100]
with torch.no_grad():
    x2 = mlp2.hiddenFeature(x)
```

```
# [THIS IS READ-ONLY]
# @check
# @title: get the hidden layout output

x2.shape
```

```
# [THIS IS READ-ONLY]
#
# Plot the three species using their hidden features
#

I0 = target == 0
I1 = target == 1
I2 = target == 2

plt.figure(figsize=(5,5))
plt.plot(x2[I0, 0], x2[I0, 1], '*', color='red');
plt.plot(x2[I1, 0], x2[I1, 1], '+', color='green');
plt.plot(x2[I2, 0], x2[I2, 1], '^', color='blue');
```