Multiclass Classification with PyTorch

In [57]:
```
# Import libraries
import torch
import torch.nn as nn
import torch. optim as optim
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

In [58]:
```
# Load Iris dataset, assign X and y
iris = load_iris()
X = iris.data
y = iris.target
X.shape, y.shape
```

Out[58]:
```
((150, 4), (150,))
```

In [59]:
```
# Convert Iris data into a dataframe to view data
iris_df = pd.DataFrame({
    'X1': X[:,0],
    'X2': X[:,1],
    'X3': X[:,2],
    'X4': X[:,3],
    'y': y

})
```

In [60]:
```
iris_df.head(10)
```

Out[60]:

	X1	X2	X3	X4	y
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
5	5.4	3.9	1.7	0.4	0
6	4.6	3.4	1.4	0.3	0
7	5.0	3.4	1.5	0.2	0
8	4.4	2.9	1.4	0.2	0
9	4.9	3.1	1.5	0.1	0

In [61]:

```
# How many labels for each class
iris_df.y.value_counts()
```

Out[61]:

```
y
0    50
1    50
2    50
Name: count, dtype: int64
```

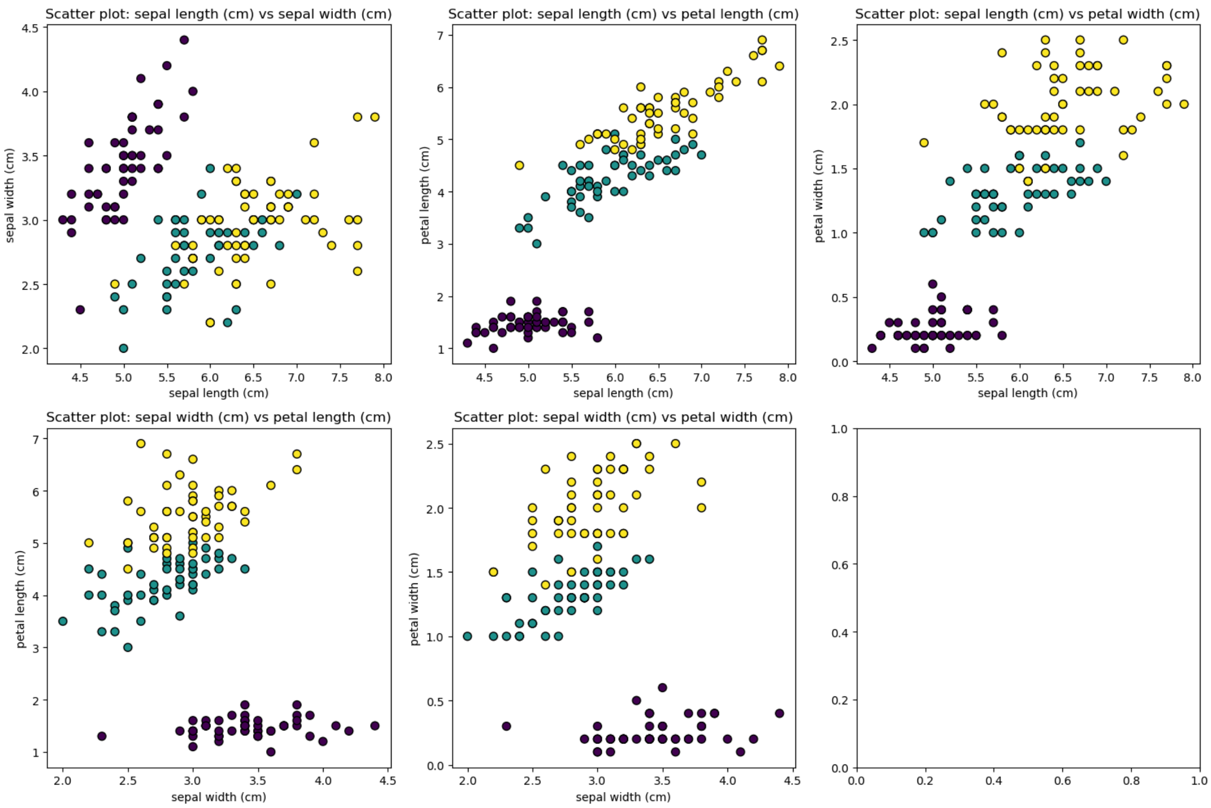
In [62]:

```
# Visualize relationship between features
fig, axes = plt.subplots(2, 3, figsize=(15,10))

feature_combinations = [(0,1), (0,2), (0,3), (1,2), (1,3)]

for ax, features in zip(axes.flatten(), feature_combinations):
    feature1, feature2 = features
    ax.scatter(X[:, feature1], X[:, feature2], c=y, cmap='viridis', edgecolo
    ax.set_xlabel(iris.feature_names[feature1])
    ax.set_ylabel(iris.feature_names[feature2])
    ax.set_title(f"Scatter plot: {iris.feature_names[feature1]} vs {iris.fea

plt.tight_layout()
plt.show()
```



In [63]:
```
# Split data
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2, ranc
```

In [64]:
```
# Normalize data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [65]:
```
# Set device to run on GPU
# device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

In [66]:
```
# Create tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float) #.to(device)
y_train_tensor = torch.tensor(y_train, dtype=torch.int64) # class labels are
X_test_tensor = torch.tensor(X_test, dtype=torch.float)
y_test_tensor = torch.tensor(y_test, dtype=torch.int64)
```

In [ ]:
```
# defining a neural network by subclassing torch.nn.Module. Every trainable
class SimpleClassifier(nn.Module): # Inherit from PyTorch
    def __init__(self, in_features, out_features):
        super().__init__() #  initializes nn.Module internals

        # Defines three fully-connected (linear) layers: the first 2 act as
        ## nn.Linear computes y=mx+b
        # In a multiclass classifier with C classes, the final layer outputs
        self.layer_1 = nn.Linear(in_features, 120) # Layer 1: projects the r
        self.layer_2 = nn.Linear(120,10) # hidden layer: compresses/reshapes
        self.layer_3 = nn.Linear(10, out_features) # out_features: maps the
        # no non-linear activation function, so purely linear representatior
```

```
    # Set forward method
    def forward(self, x):
        return self.layer_3(self.layer_2(self.layer_1(x)))
```

As written, there are no activation functions (e.g., ReLU, GELU) between linear layers. Mathematically, a stack of purely linear layers is equivalent to one linear transform (the matrices multiply). So:

You still get learnable weights and biases, and it will behave like a multiclass linear classifier (a generalized linear model).

The middle layer sizes (120 → 10) don't add expressive power without nonlinearities; they just factor the final linear map.

In [ ]:
```
# Establish parameters
in_features = X_train.shape[1] # just the columns, aka must match input tens
num_classes = len(set(y)) # set converts a list to auto remove any duplicate

# Instantiate model class
model = SimpleClassifier(in_features, num_classes) #.to(device)
```

In [ ]:
```
# Loss and Optimizer
criterion = nn.CrossEntropyLoss() # penalizes incorrect class probabilities
optimizer = optim.SGD(model.parameters(), # stochastic gradient descent opti
                      lr=0.01)
```

More on criterion (aka Cross Entropy Loss):

- It rewards the model if the correct class has a higher score than the others.
- It penalizes it if the correct class score is too low compared to the wrong ones.
- This tells the model how to move each score (increase the correct one, decrease the others) and by how much.

In [ ]:
```
epochs = 1000
for epoch in range(epochs):
    model.train() # sets training mode

    # outputs = the model's raw scores for every class, for every flower.
    outputs = model(X_train_tensor) # forward pass, each row is item and col
    loss = criterion(outputs, y_train_tensor) # computes average cross-entro

    _, predicted_labels = torch.max(outputs, 1) # looks across each row and
    correct_preds = (predicted_labels == y_train_tensor).sum().item() #item

    total_samples = len(y_train_tensor)
    acc = correct_preds / total_samples

    # correcting model and updating
    # Standard gradient update: clear old grads → backprop → take a step.
    optimizer.zero_grad() # clears the gradient calculate from previous step
    loss.backward() # which params to adjust and by how much
    optimizer.step() # actually make adjustment
```

```
    if (epoch+1) % 10 == 0:
        print(f"Epoch: {epoch+1}/{epochs}, Loss: {loss.item():.4f}, Accuracy
```

```
Epoch: 10/1000, Loss: 0.8988, Accuracy: 0.7583
Epoch: 20/1000, Loss: 0.7366, Accuracy: 0.7667
Epoch: 30/1000, Loss: 0.6328, Accuracy: 0.7833
Epoch: 40/1000, Loss: 0.5631, Accuracy: 0.7917
Epoch: 50/1000, Loss: 0.5141, Accuracy: 0.8000
Epoch: 60/1000, Loss: 0.4781, Accuracy: 0.8333
Epoch: 70/1000, Loss: 0.4503, Accuracy: 0.8333
Epoch: 80/1000, Loss: 0.4280, Accuracy: 0.8333
Epoch: 90/1000, Loss: 0.4095, Accuracy: 0.8417
Epoch: 100/1000, Loss: 0.3936, Accuracy: 0.8583
Epoch: 110/1000, Loss: 0.3797, Accuracy: 0.8667
Epoch: 120/1000, Loss: 0.3672, Accuracy: 0.8833
Epoch: 130/1000, Loss: 0.3558, Accuracy: 0.9000
Epoch: 140/1000, Loss: 0.3453, Accuracy: 0.9000
Epoch: 150/1000, Loss: 0.3355, Accuracy: 0.9000
Epoch: 160/1000, Loss: 0.3263, Accuracy: 0.8917
Epoch: 170/1000, Loss: 0.3176, Accuracy: 0.9000
Epoch: 180/1000, Loss: 0.3093, Accuracy: 0.9000
Epoch: 190/1000, Loss: 0.3014, Accuracy: 0.9000
Epoch: 200/1000, Loss: 0.2938, Accuracy: 0.9083
Epoch: 210/1000, Loss: 0.2865, Accuracy: 0.9083
Epoch: 220/1000, Loss: 0.2794, Accuracy: 0.9083
Epoch: 230/1000, Loss: 0.2726, Accuracy: 0.9083
Epoch: 240/1000, Loss: 0.2660, Accuracy: 0.9000
Epoch: 250/1000, Loss: 0.2596, Accuracy: 0.9000
Epoch: 260/1000, Loss: 0.2534, Accuracy: 0.9167
Epoch: 270/1000, Loss: 0.2474, Accuracy: 0.9167
Epoch: 280/1000, Loss: 0.2415, Accuracy: 0.9250
Epoch: 290/1000, Loss: 0.2359, Accuracy: 0.9250
Epoch: 300/1000, Loss: 0.2304, Accuracy: 0.9250
Epoch: 310/1000, Loss: 0.2251, Accuracy: 0.9250
Epoch: 320/1000, Loss: 0.2199, Accuracy: 0.9333
Epoch: 330/1000, Loss: 0.2150, Accuracy: 0.9417
Epoch: 340/1000, Loss: 0.2101, Accuracy: 0.9417
Epoch: 350/1000, Loss: 0.2054, Accuracy: 0.9417
Epoch: 360/1000, Loss: 0.2009, Accuracy: 0.9417
Epoch: 370/1000, Loss: 0.1965, Accuracy: 0.9417
Epoch: 380/1000, Loss: 0.1923, Accuracy: 0.9417
Epoch: 390/1000, Loss: 0.1882, Accuracy: 0.9417
Epoch: 400/1000, Loss: 0.1842, Accuracy: 0.9417
Epoch: 410/1000, Loss: 0.1804, Accuracy: 0.9500
Epoch: 420/1000, Loss: 0.1767, Accuracy: 0.9583
Epoch: 430/1000, Loss: 0.1732, Accuracy: 0.9583
Epoch: 440/1000, Loss: 0.1697, Accuracy: 0.9583
Epoch: 450/1000, Loss: 0.1664, Accuracy: 0.9667
Epoch: 460/1000, Loss: 0.1632, Accuracy: 0.9667
Epoch: 470/1000, Loss: 0.1601, Accuracy: 0.9667
Epoch: 480/1000, Loss: 0.1571, Accuracy: 0.9667
Epoch: 490/1000, Loss: 0.1542, Accuracy: 0.9667
Epoch: 500/1000, Loss: 0.1514, Accuracy: 0.9667
Epoch: 510/1000, Loss: 0.1487, Accuracy: 0.9667
Epoch: 520/1000, Loss: 0.1462, Accuracy: 0.9667
Epoch: 530/1000, Loss: 0.1437, Accuracy: 0.9667
Epoch: 540/1000, Loss: 0.1413, Accuracy: 0.9667
Epoch: 550/1000, Loss: 0.1389, Accuracy: 0.9667
Epoch: 560/1000, Loss: 0.1367, Accuracy: 0.9667
```

```
Epoch: 570/1000, Loss: 0.1345, Accuracy: 0.9667
Epoch: 580/1000, Loss: 0.1324, Accuracy: 0.9667
Epoch: 590/1000, Loss: 0.1304, Accuracy: 0.9667
Epoch: 600/1000, Loss: 0.1285, Accuracy: 0.9667
Epoch: 610/1000, Loss: 0.1266, Accuracy: 0.9667
Epoch: 620/1000, Loss: 0.1248, Accuracy: 0.9667
Epoch: 630/1000, Loss: 0.1230, Accuracy: 0.9667
Epoch: 640/1000, Loss: 0.1213, Accuracy: 0.9667
Epoch: 650/1000, Loss: 0.1197, Accuracy: 0.9667
Epoch: 660/1000, Loss: 0.1181, Accuracy: 0.9667
Epoch: 670/1000, Loss: 0.1166, Accuracy: 0.9667
Epoch: 680/1000, Loss: 0.1151, Accuracy: 0.9667
Epoch: 690/1000, Loss: 0.1137, Accuracy: 0.9667
Epoch: 700/1000, Loss: 0.1123, Accuracy: 0.9667
Epoch: 710/1000, Loss: 0.1109, Accuracy: 0.9667
Epoch: 720/1000, Loss: 0.1096, Accuracy: 0.9667
Epoch: 730/1000, Loss: 0.1084, Accuracy: 0.9667
Epoch: 740/1000, Loss: 0.1072, Accuracy: 0.9667
Epoch: 750/1000, Loss: 0.1060, Accuracy: 0.9583
Epoch: 760/1000, Loss: 0.1048, Accuracy: 0.9583
Epoch: 770/1000, Loss: 0.1037, Accuracy: 0.9583
Epoch: 780/1000, Loss: 0.1026, Accuracy: 0.9583
Epoch: 790/1000, Loss: 0.1016, Accuracy: 0.9583
Epoch: 800/1000, Loss: 0.1006, Accuracy: 0.9583
Epoch: 810/1000, Loss: 0.0996, Accuracy: 0.9583
Epoch: 820/1000, Loss: 0.0986, Accuracy: 0.9583
Epoch: 830/1000, Loss: 0.0977, Accuracy: 0.9583
Epoch: 840/1000, Loss: 0.0968, Accuracy: 0.9583
Epoch: 850/1000, Loss: 0.0959, Accuracy: 0.9583
Epoch: 860/1000, Loss: 0.0951, Accuracy: 0.9583
Epoch: 870/1000, Loss: 0.0943, Accuracy: 0.9583
Epoch: 880/1000, Loss: 0.0935, Accuracy: 0.9583
Epoch: 890/1000, Loss: 0.0927, Accuracy: 0.9583
Epoch: 900/1000, Loss: 0.0919, Accuracy: 0.9583
Epoch: 910/1000, Loss: 0.0912, Accuracy: 0.9583
Epoch: 920/1000, Loss: 0.0905, Accuracy: 0.9583
Epoch: 930/1000, Loss: 0.0898, Accuracy: 0.9583
Epoch: 940/1000, Loss: 0.0891, Accuracy: 0.9583
Epoch: 950/1000, Loss: 0.0884, Accuracy: 0.9583
Epoch: 960/1000, Loss: 0.0878, Accuracy: 0.9583
Epoch: 970/1000, Loss: 0.0871, Accuracy: 0.9583
Epoch: 980/1000, Loss: 0.0865, Accuracy: 0.9583
Epoch: 990/1000, Loss: 0.0859, Accuracy: 0.9583
Epoch: 1000/1000, Loss: 0.0853, Accuracy: 0.9583
```

In [ ]:

```
# test data set
model.eval() # eval mode, turns off grad
with torch.inference_mode(): # better than zero grad
    outputs = model(X_test_tensor) # forward pass, spits out scores for each

    _, predicted = torch.max(outputs, 1) # choosest highest flower score = p

    accuracy = accuracy_score(y_test, predicted.numpy()) # cannot use scikit

    predicted_tensor = predicted.clone().detach() # make a clean, standalone
    loss = criterion(outputs, y_test_tensor) # calculate loss: compare predi
```

```python
print(f"Loss: {loss.item():.4f}, Accuracy: {accuracy:.4f}")
```

```
Loss: 0.0676, Accuracy: 0.9667
```