In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

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.preprocessing import LabelEncoder


from torch.utils.data import DataLoader, TensorDataset
```

In [2]:
```
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris()

# You can now access the data and target variables
X = iris.data
y = iris.target

# You can also see the feature names and target names
feature_names = iris.feature_names
target_names = iris.target_names

print("Features (X):")
print(X[:5]) # Print the first 5 rows of features

print("\nTargets (y):")
print(y[:5]) # Print the first 5 target values

print("\nFeature Names:")
print(feature_names)
print("\nTarget Names:")

print(target_names)
```

```
Features (X):
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]

Targets (y):
[0 0 0 0 0]

Feature Names:
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

Target Names:
['setosa' 'versicolor' 'virginica']
```

Data visualization and pre-processing

The PCA plot helps illustrate overall separability between classes.

In [3]:
```
# Create a DataFrame from the data
df = pd.DataFrame(X, columns=feature_names)
df['species'] = y  # Add target column

# Optional: Replace numerical labels with actual species names
df['species'] = df['species'].map({i: name for i, name in enumerate(target_names)})
```

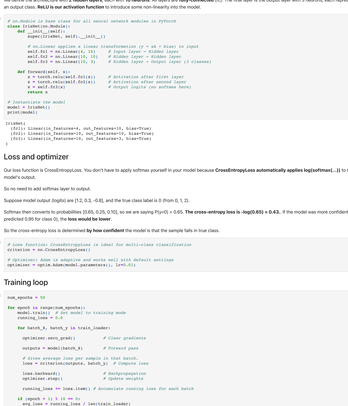
In [4]:
```
print(df.head())
print(df.describe())
print(df['species'].value_counts())
```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

  species
0  setosa
1  setosa
2  setosa
3  setosa
4  setosa
       sepal length (cm)  sepal width (cm)  petal length (cm)  \
count         150.000000        150.000000         150.000000
mean            5.843333          3.057333           3.758000
std             0.828066          0.435866           1.765298
min             4.300000          2.000000           1.000000
25%             5.100000          2.800000           1.600000
50%             5.800000          3.000000           4.350000
75%             6.400000          3.300000           5.100000
max             7.900000          4.400000           6.900000

       petal width (cm)
count        150.000000
mean           1.199333
std            0.762238
min            0.100000
25%            0.300000
50%            1.300000
75%            1.800000
max            2.500000
species
setosa      50
versicolor  50
virginica   50
Name: count, dtype: int64
```

In [5]:
```
df.hist(figsize=(15, 9))
plt.tight_layout()
plt.show()
```

In [6]:
```
sns.pairplot(df, hue='species', markers=["o", "s", "D"])
plt.show()
```

In [7]:
```
plt.figure(figsize=(8, 6))
sns.heatmap(df.iloc[:, :-1].corr(), annot=True, cmap='coolwarm')
plt.title('Feature Correlation Heatmap')
plt.show()
```



In [8]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

df_pca = pd.DataFrame(X_pca, columns=['PC1', 'PC2'])
df_pca['species'] = df['species']

sns.scatterplot(data=df_pca, x='PC1', y='PC2', hue='species')
plt.title('PCA of Iris dataset')
plt.show()
```

Model building and evaluation

StandardScaler() **standardizes the data**, which means each feature is transformed to have **mean = 0 and variance = 1**. Even when original features do **not follow Normal** distribution, standardization is helpful because **features will have one mean and be on the same scale with unit variance**. So, gradient descent will **converge faster**.

Can consider **MinMaxScaler** if data is too skewed or has many outliers.

Also, it is a good idea to **shuffle training data** to avoid overfitting and speed up convergence. Without shuffling, model might adapt to patterns in the training data and poorly generalize.

In [9]:
```
# Standardize features (mean = 0, variance = 1).
# Ensures that data is centered with same scale.
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Train-test split (80% train, 20% test)

# Note: STRATIFICATION is necessary only if output classes are imbalanced. Eg,
# if 90% of samples belong to one class, we want these proportionally represented
# in the train and test sets.
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Convert to PyTorch tensors

# PyTorch models work with tensors. Specify float32 datatype for input, and long
# datatype for output.
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)



# Create datasets and dataloaders

# Dataset wrapper allows pairing inputs and targets.
# PyTorch knows how to fetch a sample as (X_train[i], y_train[i])
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

# DataLoader: splits dataset into mini-batches (here, 5 samples per batch)
# Optionally shuffles the data (important during training!)

train_loader = DataLoader(train_dataset, batch_size=5, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=5)

# Don't shuffle test_loader, so we can consistently evaluate model.
```

Model architecture

We define the architecture with **2 hidden layers**, each with **10 neurons**. All layers are **fully-connected** (fc). The final layer is the output layer with 3 neurons, each representing an output class. **ReLU is our activation function** to introduce some non-linearity into the model.

In [10]:
```
# nn.Module is base class for all neural network modules in PyTorch
class IrisNet(nn.Module):
    def __init__(self):
        super(IrisNet, self).__init__()

        # An Linear applies a linear transformation (y = wA + b)as to input
        self.fc1 = nn.Linear(4, 10)   # Input layer - Hidden layer
        self.fc2 = nn.Linear(10, 10)  # Hidden layer - Hidden layer
        self.fc3 = nn.Linear(10, 3)   # Hidden layer - Output layer (3 classes)

    def forward(self, x):
        x = torch.relu(self.fc1(x))   # Activation after first layer
        x = torch.relu(self.fc2(x))   # Activation after second layer
        x = self.fc3(x)               # Output logits (no softmax here)
        return x

# Instantiate the model
model = IrisNet()
print(model)

[class]
  (fc1): Linear(in_features=4, out_features=10, bias=True)
  (fc2): Linear(in_features=10, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=3, bias=True)
)
```

Loss and optimizer

Our loss function is CrossEntropyLoss. You don't have to apply softmax yourself in your model because **CrossEntropyLoss automatically applies log(softmax(...))** to the model's output.

So no need to add softmax layer to output.

Suppose model output (logits) are [1.2, 0.3, -0.8], and the true class label is 0 (from 0, 1, 2).

Softmax then converts to probabilities [0.65, 0.25, 0.10], so we are saying P(y=0) = 0.65. **The cross-entropy loss is -log(0.65) = 0.43.** If the model was more confident (e.g., predicted 0.95 for class 0), the **loss would be lower**.

So the cross-entropy loss is determined **by how confident** the model is that the sample falls in true class.

In [11]:
```
# Loss function: CrossEntropyLoss is ideal for multi-class classification
criterion = nn.CrossEntropyLoss()

# Optimizer: Adam is adaptive and works well with default settings
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Training loop

In [12]:
```
num_epochs = 50

for epoch in range(num_epochs):
    model.train()  # Set model to training mode
    running_loss = 0.0

    for batch_X, batch_y in train_loader:

        optimizer.zero_grad()           # Clear gradients

        outputs = model(batch_X)        # Forward pass

        # Gives average loss per sample in that batch.
        loss = criterion(outputs, batch_y)  # Compute loss

        loss.backward()                 # Backpropagation
        optimizer.step()                # Update weights

        running_loss += loss.item() # Accumulate running loss for each batch

    if (epoch + 1) % 10 == 0:
        avg_loss = running_loss / len(train_loader)
        print(f'Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss:.4f}')
```

```
Epoch [10/50], Loss: 0.9768
Epoch [20/50], Loss: 0.0565
Epoch [30/50], Loss: 0.0547
Epoch [40/50], Loss: 0.0567
Epoch [50/50], Loss: 0.0687
```

Evaluation on test set

In [13]:
```
model.eval()  # Set model to evaluation mode
correct = 0
total = 0

with torch.no_grad():  # Disable gradient computation for efficiency
    for batch_X, batch_y in test_loader:
        outputs = model(batch_X)               # Forward pass

        _, predicted = torch.max(outputs, 1) # Get predicted class index

        total += batch_y.size(0)               # Update total sample count

        correct += (predicted == batch_y).sum().item()  # Count correct predictions

accuracy = 100 * correct / total
print(f'\nTest Accuracy: {accuracy:.2f}%')
```

```

Test Accuracy: 96.67%
```

Possible next steps

- Try dropout or batch normalization

In [14]:
```

```