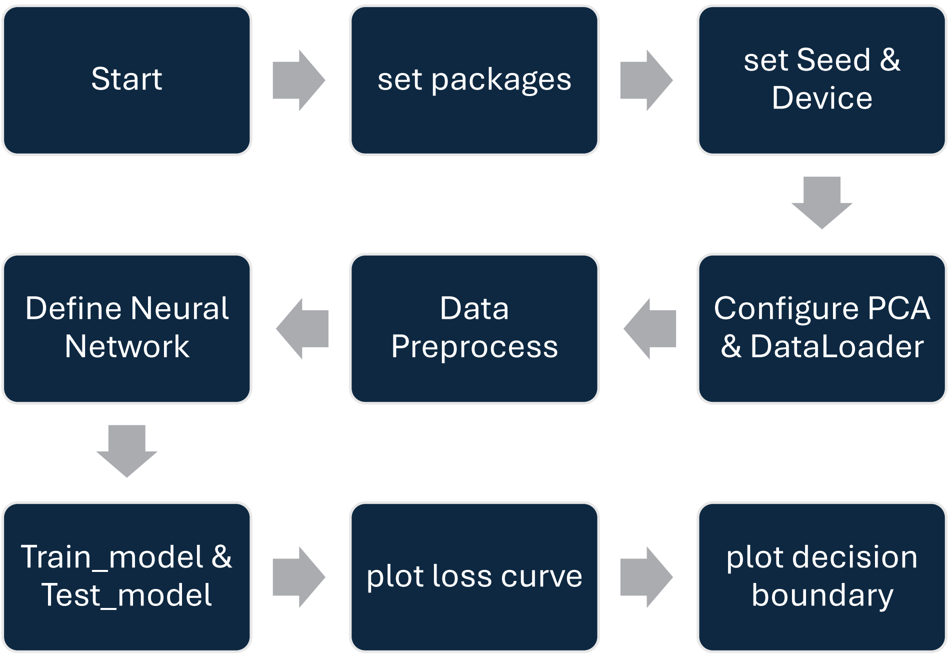
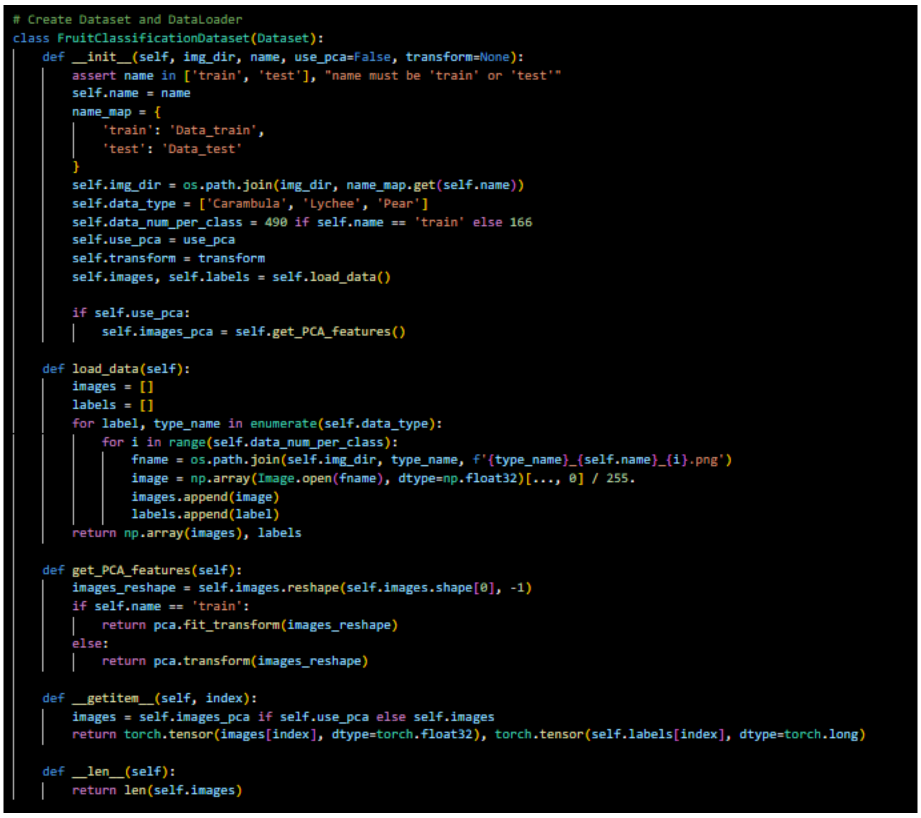
1.

Flow Chart :



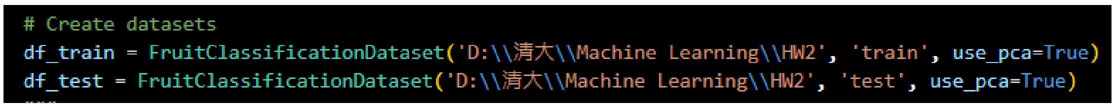
Description:



```python
# Create Dataset and DataLoader
class FruitClassificationDataset(Dataset):
    def __init__(self, img_dir, name, use_pca=False, transform=None):
        assert name in ['train', 'test'], "name must be 'train' or 'test'"
        self.name = name
        name_map = {
            'train': 'Data_train',
            'test': 'Data_test'
        }
        self.img_dir = os.path.join(img_dir, name_map.get(self.name))
        self.data_type = ['Carambula', 'Lychee', 'Pear']
        self.data_num_per_class = 490 if self.name == 'train' else 166
        self.use_pca = use_pca
        self.transform = transform
        self.images, self.labels = self.load_data()

        if self.use_pca:
            self.images_pca = self.get_PCA_features()

    def load_data(self):
        images = []
        labels = []
        for label, type_name in enumerate(self.data_type):
            for i in range(self.data_num_per_class):
                fname = os.path.join(self.img_dir, type_name, f'{type_name}_{self.name}_{i}.png')
                image = np.array(Image.open(fname), dtype=np.float32)[..., 0] / 255.
                images.append(image)
                labels.append(label)
        return np.array(images), labels

    def get_PCA_features(self):
        images_reshape = self.images.reshape(self.images.shape[0], -1)
        if self.name == 'train':
            return pca.fit_transform(images_reshape)
        else:
            return pca.transform(images_reshape)

    def __getitem__(self, index):
        images = self.images_pca if self.use_pca else self.images
        return torch.tensor(images[index], dtype=torch.float32), torch.tensor(self.labels[index], dtype=torch.long)

    def __len__(self):
        return len(self.images)
```

Defines a custom PyTorch dataset class of FruitClassificationDataset that is used to load and process image data for use in training and testing machine learning models.

For example:

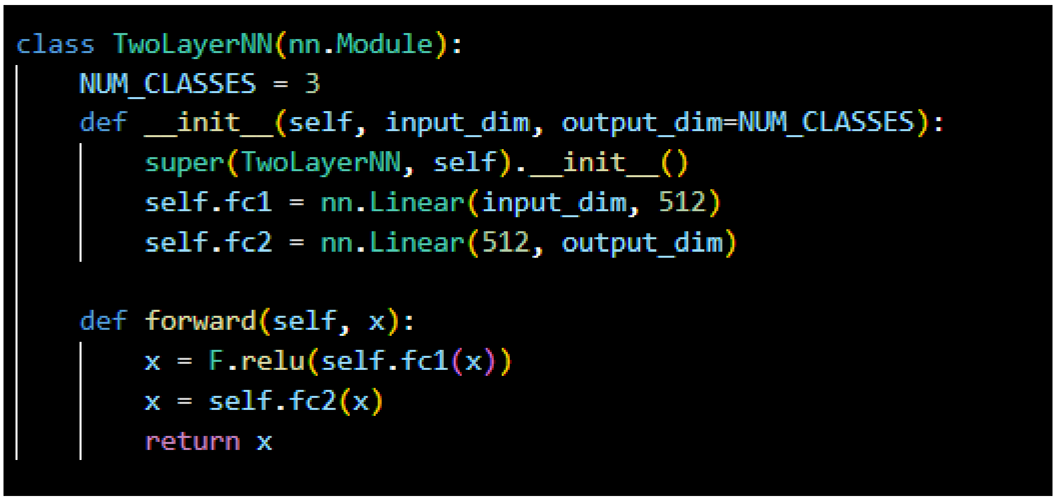


```
# Create datasets
df_train = FruitClassificationDataset('D:\\清大\\Machine Learning\\HW2', 'train', use_pca=True)
df_test = FruitClassificationDataset('D:\\清大\\Machine Learning\\HW2', 'test', use_pca=True)
```

Neural Network:

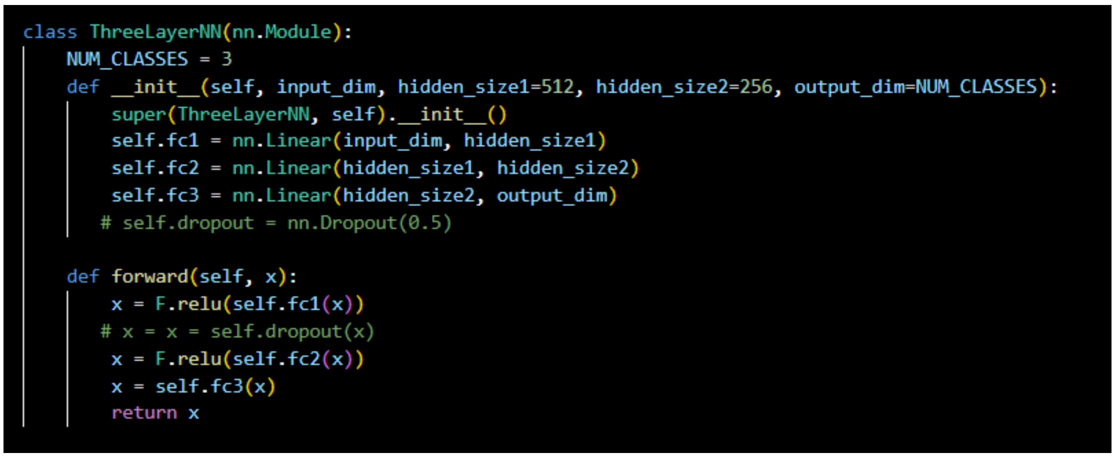
The class TwoLayerNN is a simple feedforward neural network with two fully connected (linear) layers. Here's a code to represent the structure of this network:

(ReLU can increase the sparsity of neural networks, effectively slow down the occurrence of overfitting, and can also solve problems such as gradient explosion or gradient disappearance.)



```
class TwoLayerNN(nn.Module):
    NUM_CLASSES = 3
    def __init__(self, input_dim, output_dim=NUM_CLASSES):
        super(TwoLayerNN, self).__init__()
        self.fc1 = nn.Linear(input_dim, 512)
        self.fc2 = nn.Linear(512, output_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

The input x first passes through the first fully connected layer fc1 to obtain a 512-dimensional output. Then, a ReLU activation function is applied to nonlinearly transform the output. Then through the second fully connected layer fc2, the 512-dimensional data is mapped to the output_dim dimension (that is, the number of categories for classification).However,The general structure of threeLayerNN is similar to that of twoLayerNN, but with one more layer.



```
class ThreeLayerNN(nn.Module):
    NUM_CLASSES = 3
    def __init__(self, input_dim, hidden_size1=512, hidden_size2=256, output_dim=NUM_CLASSES):
        super(ThreeLayerNN, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, output_dim)
       # self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = F.relu(self.fc1(x))
       # x = x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

train_model:

Purpose: Train the model and keep track of the training losses.

Steps:

1. **Initialization**:

   Start with an empty list train_losses to store the training loss for each epoch.

2. **Loop Through Epochs**:

   Get the total size of the dataset and set up variables for tracking loss and correct predictions.

3. **Set Model to Training Mode**:

   Use model.train() to enable training mode, which turns on features like dropout and batch normalization.

4. **Loop Through DataLoader**:

   Fetch a batch of data (images and labels) from the dataloader and move it to the device (CPU or GPU).

5. **Preprocess Data**:

   If using PCA, reshape the images accordingly.

6. **Forward Pass and Compute Loss**:

   Pass the data through the model to get predictions.

Compute the loss using the predictions and labels.

7. **Backward Pass and Optimization**:

   Clear previous gradients, perform backpropagation, and update the model's parameters.

8. **Track Loss and Accuracy**:

   Keep a running total of the loss and count how many predictions are correct.

9. **Calculate Average Loss and Accuracy**:

   Compute the average loss and accuracy for the current epoch.

10. **Print Results**:

    Display the loss and accuracy for the epoch.

11. **Return**:

    Return the trained model and the list of training losses.

Backpropagation:

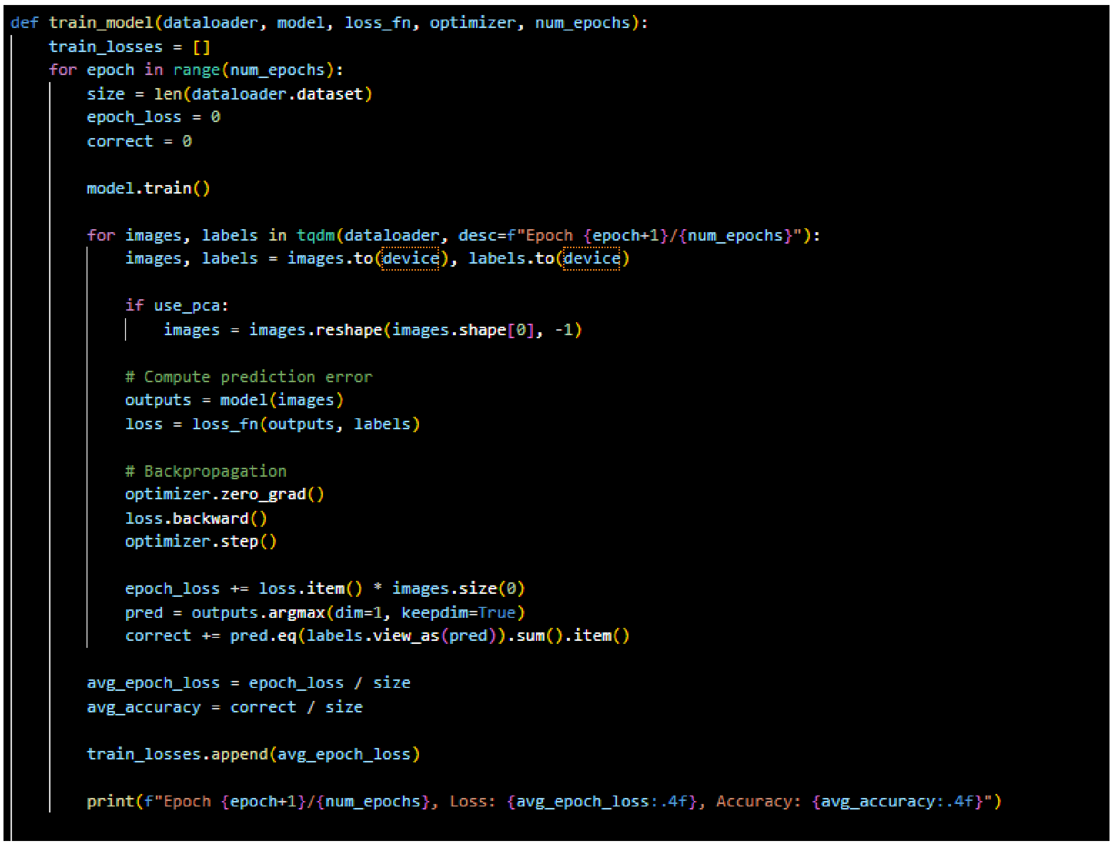
(1.)Zero Gradients: Clear previous gradients to avoid accumulation.

(2.)Compute Gradients: Calculate new gradients based on the loss.

(3.)Update Parameters: Apply the gradients to adjust the model parameters and reduce the loss.

In PyTorch, nn.CrossEntropyLoss is a commonly used loss function that internally combines softmax and negative log-likelihood loss (NLLLoss). Therefore, you can pass logits directly to nn.CrossEntropyLoss without explicitly calling softmax

The train_model code :



```python
def train_model(dataloader, model, loss_fn, optimizer, num_epochs):
    train_losses = []
    for epoch in range(num_epochs):
        size = len(dataloader.dataset)
        epoch_loss = 0
        correct = 0

        model.train()

        for images, labels in tqdm(dataloader, desc=f"Epoch {epoch+1}/{num_epochs}"):
            images, labels = images.to(device), labels.to(device)

            if use_pca:
                images = images.reshape(images.shape[0], -1)

            # Compute prediction error
            outputs = model(images)
            loss = loss_fn(outputs, labels)

            # Backpropagation
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item() * images.size(0)
            pred = outputs.argmax(dim=1, keepdim=True)
            correct += pred.eq(labels.view_as(pred)).sum().item()

        avg_epoch_loss = epoch_loss / size
        avg_accuracy = correct / size

        train_losses.append(avg_epoch_loss)

        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {avg_epoch_loss:.4f}, Accuracy: {avg_accuracy:.4f}")
```

test_model:

Purpose: Test the model and evaluate its performance.

Steps:

1. **Initialization**:

   Set the model to evaluation mode with model.eval() and initialize counters for correct predictions and total loss. Also, prepare lists for storing all predictions and labels.

2. **Turn Off Gradient Calculation**:

   Use torch.no_grad() to speed up inference by not calculating gradients.

3. **Loop Through DataLoader**:

   Fetch a batch of data (images and labels) from the dataloader and move it to the device (CPU or GPU).

4. **Preprocess Data**:

   If using PCA, reshape the images accordingly.

5. **Forward Pass and Compute Loss**:

   (1.)Pass the data through the model to get predictions.

   (2.)Compute the loss and accumulate it.

6. **Predict and Calculate Accuracy**:

   (1.)Use torch.max to get the predicted class.

   (2.)Keep track of the total number of predictions and how many were correct.

7. **Compute Probability Distributions**:

   Use F.softmax to convert model outputs into probabilities and store them.

8. **Calculate Overall Accuracy and Loss**:

   Compute the average loss and accuracy for the test set.

9. **Print Results**:

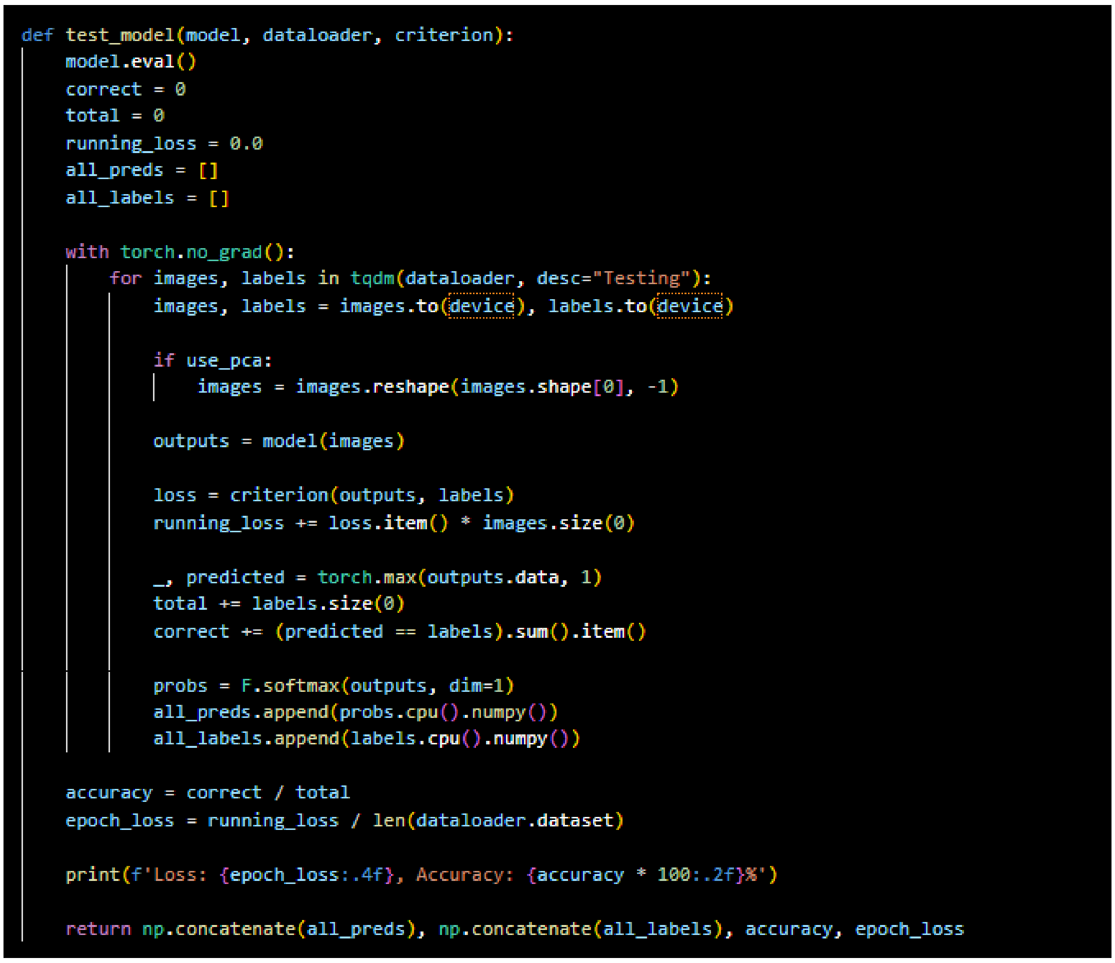
   Display the test loss and accuracy.

10. **Return**:

    Return the concatenated predictions, labels, accuracy, and average loss.

During the testing stage, softmax is often used to obtain the predicted probability of each category, which helps understand the model's prediction confidence and conduct performance evaluation.

<u>The test_model code:</u>

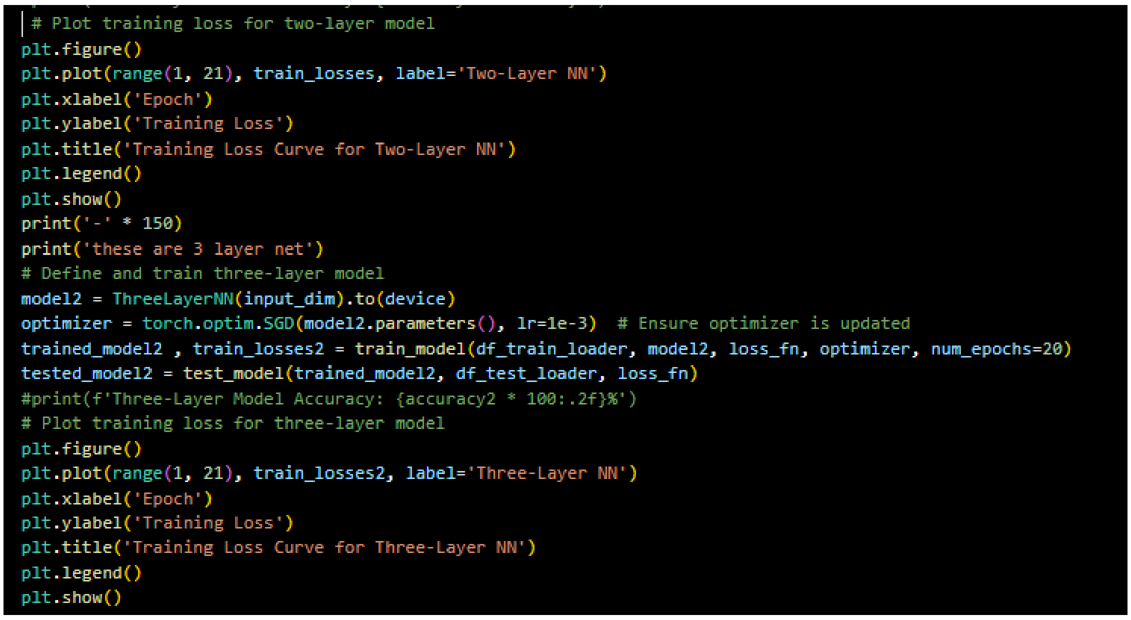


```python
def test_model(model, dataloader, criterion):
    model.eval()
    correct = 0
    total = 0
    running_loss = 0.0
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for images, labels in tqdm(dataloader, desc="Testing"):
            images, labels = images.to(device), labels.to(device)

            if use_pca:
                images = images.reshape(images.shape[0], -1)

            outputs = model(images)

            loss = criterion(outputs, labels)
            running_loss += loss.item() * images.size(0)

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            probs = F.softmax(outputs, dim=1)
            all_preds.append(probs.cpu().numpy())
            all_labels.append(labels.cpu().numpy())

    accuracy = correct / total
    epoch_loss = running_loss / len(dataloader.dataset)

    print(f'Loss: {epoch_loss:.4f}, Accuracy: {accuracy * 100:.2f}%')

    return np.concatenate(all_preds), np.concatenate(all_labels), accuracy, epoch_loss
```

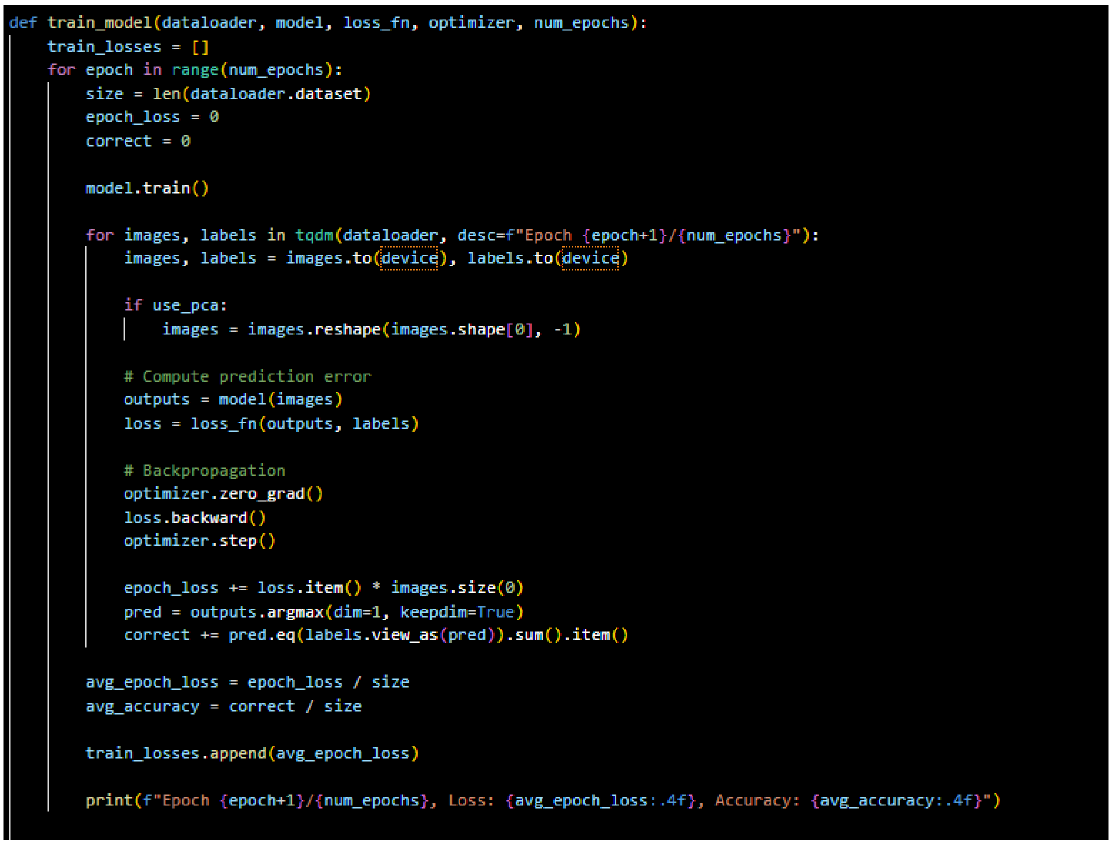
<u>Why not use softmax for neural networks?</u>

During the training process of neural networks, softmax is usually not used directly in the network output stage but is left to be processed inside the loss function. This is because directly using separate processing of softmax and cross-entropy loss may lead to numerical instability.

Plot training loss curves:



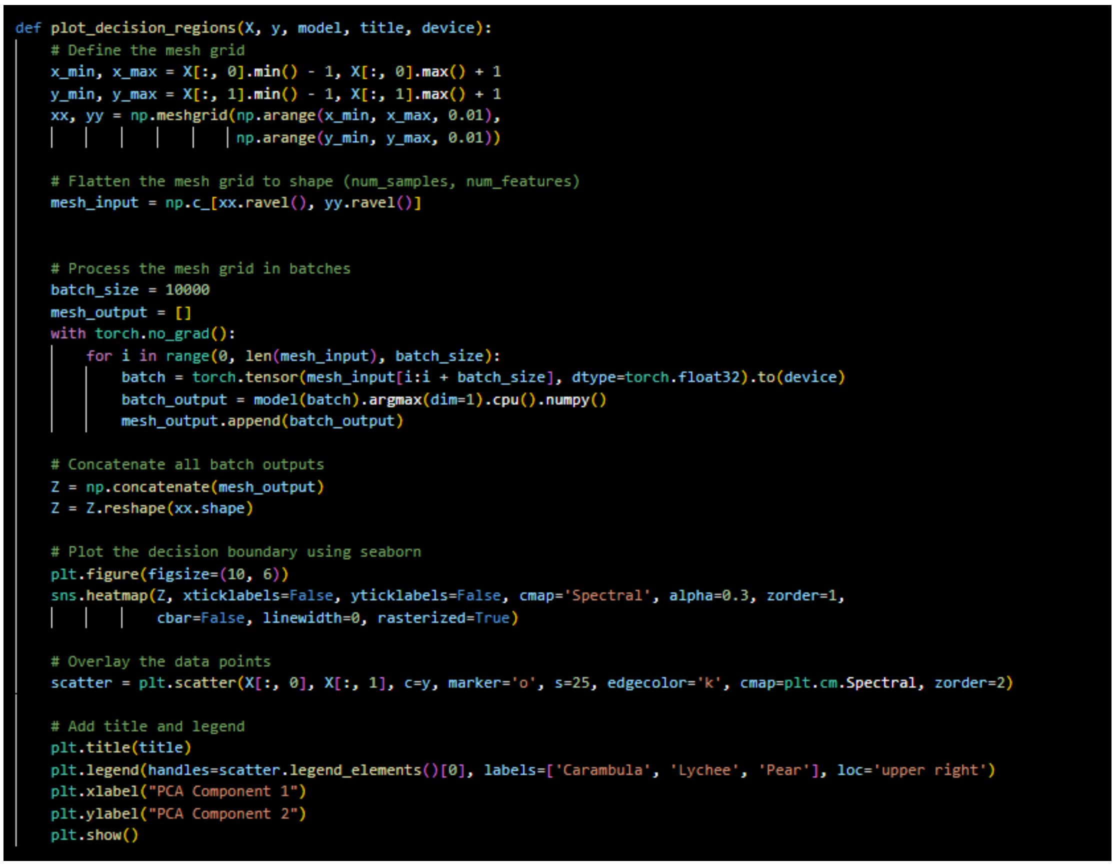
```
# Plot training loss for two-layer model
plt.figure()
plt.plot(range(1, 21), train_losses, label='Two-Layer NN')
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Training Loss Curve for Two-Layer NN')
plt.legend()
plt.show()
print('-' * 150)
print('these are 3 layer net')
# Define and train three-layer model
model2 = ThreeLayerNN(input_dim).to(device)
optimizer = torch.optim.SGD(model2.parameters(), lr=1e-3)  # Ensure optimizer is updated
trained_model2 , train_losses2 = train_model(df_train_loader, model2, loss_fn, optimizer, num_epochs=20)
tested_model2 = test_model(trained_model2, df_test_loader, loss_fn)
#print(f'Three-Layer Model Accuracy: {accuracy2 * 100:.2f}%')
# Plot training loss for three-layer model
plt.figure()
plt.plot(range(1, 21), train_losses2, label='Three-Layer NN')
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Training Loss Curve for Three-Layer NN')
plt.legend()
plt.show()
```



```
def train_model(dataloader, model, loss_fn, optimizer, num_epochs):
    train_losses = []
    for epoch in range(num_epochs):
        size = len(dataloader.dataset)
        epoch_loss = 0
        correct = 0

        model.train()

        for images, labels in tqdm(dataloader, desc=f"Epoch {epoch+1}/{num_epochs}"):
            images, labels = images.to(device), labels.to(device)

            if use_pca:
                images = images.reshape(images.shape[0], -1)

            # Compute prediction error
            outputs = model(images)
            loss = loss_fn(outputs, labels)

            # Backpropagation
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item() * images.size(0)
            pred = outputs.argmax(dim=1, keepdim=True)
            correct += pred.eq(labels.view_as(pred)).sum().item()

        avg_epoch_loss = epoch_loss / size
        avg_accuracy = correct / size

        train_losses.append(avg_epoch_loss)

        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {avg_epoch_loss:.4f}, Accuracy: {avg_accuracy:.4f}")
```

Record the loss of each loop according to train_loss in train_model, and finally draw the graph.

plot decision boundary:



```python
def plot_decision_regions(X, y, model, title, device):
    # Define the mesh grid
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                         np.arange(y_min, y_max, 0.01))

    # Flatten the mesh grid to shape (num_samples, num_features)
    mesh_input = np.c_[xx.ravel(), yy.ravel()]


    # Process the mesh grid in batches
    batch_size = 10000
    mesh_output = []
    with torch.no_grad():
        for i in range(0, len(mesh_input), batch_size):
            batch = torch.tensor(mesh_input[i:i + batch_size], dtype=torch.float32).to(device)
            batch_output = model(batch).argmax(dim=1).cpu().numpy()
            mesh_output.append(batch_output)

    # Concatenate all batch outputs
    Z = np.concatenate(mesh_output)
    Z = Z.reshape(xx.shape)

    # Plot the decision boundary using seaborn
    plt.figure(figsize=(10, 6))
    sns.heatmap(Z, xticklabels=False, yticklabels=False, cmap='Spectral', alpha=0.3, zorder=1,
                cbar=False, linewidth=0, rasterized=True)

    # Overlay the data points
    scatter = plt.scatter(X[:, 0], X[:, 1], c=y, marker='o', s=25, edgecolor='k', cmap=plt.cm.Spectral, zorder=2)

    # Add title and legend
    plt.title(title)
    plt.legend(handles=scatter.legend_elements()[0], labels=['Carambula', 'Lychee', 'Pear'], loc='upper right')
    plt.xlabel("PCA Component 1")
    plt.ylabel("PCA Component 2")
    plt.show()
```

This code defines a function called plot_decision_regions for plotting decision boundaries. The function accepts parameters X, y, model, title, and device.

First, the grid ranges x_min, x_max, y_min, and y_max is defined, which are based on the minimum and maximum values of the data X plus or minus 1. Mesh points xx and yy were generated using the np.meshgrid function and flattened into the appropriately shaped mesh_input.

Next, set the batch size batch_size to 10000 and initialize mesh_output as an empty list. After disabling gradient calculation, the grid points are processed in batches, and each batch of data is converted into torch.tensor and passed to the model for prediction. The results of each prediction are stored in batch_output and appended to the mesh_output list.

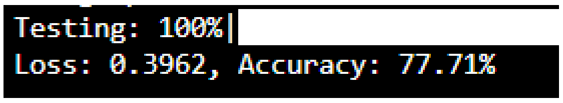
Merge the output of all batches into z and reshape z into a grid shape.

Use seaborn's heatmap function to draw the decision boundary, set the colormap to 'Spectral', and adjust the layer order and other parameters. Then, use matplotlib's scatter function to overlay the data points on the decision boundary and add a title, legend, and coordinate labels.Finally, call plt.show() to display the graph.

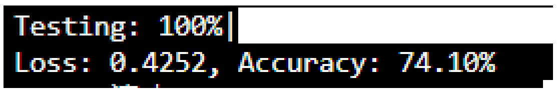
2. test accuracy

two-layer neural network: 77.71%

three-layer neural network: 74.10%

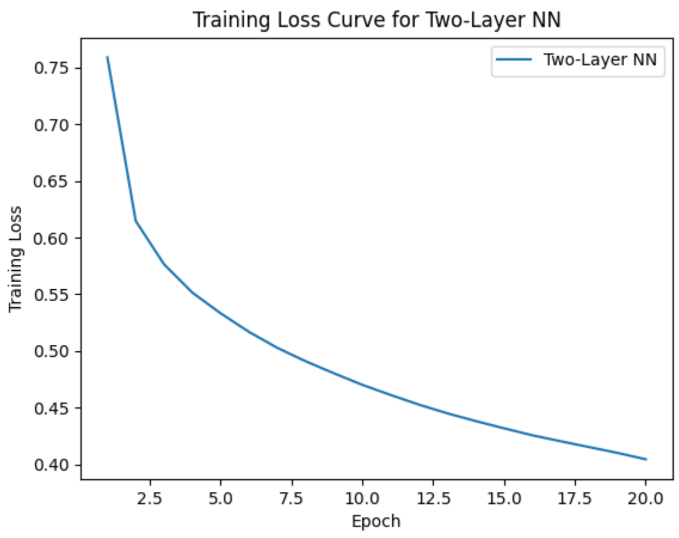


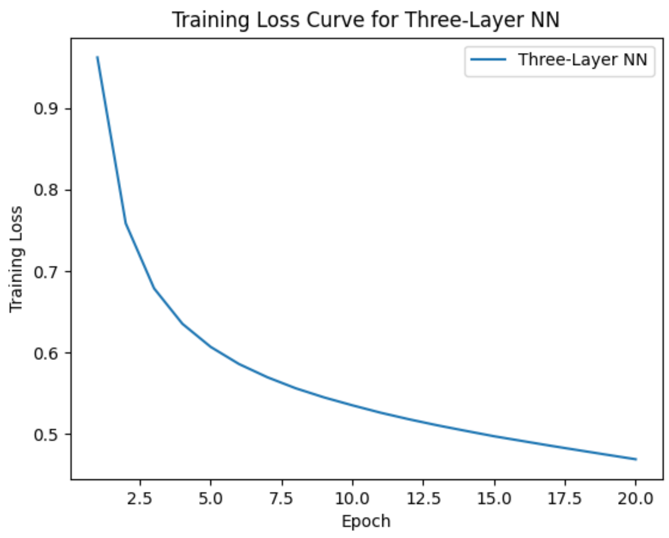
```
Testing: 100%|
Loss: 0.3962, Accuracy: 77.71%
```



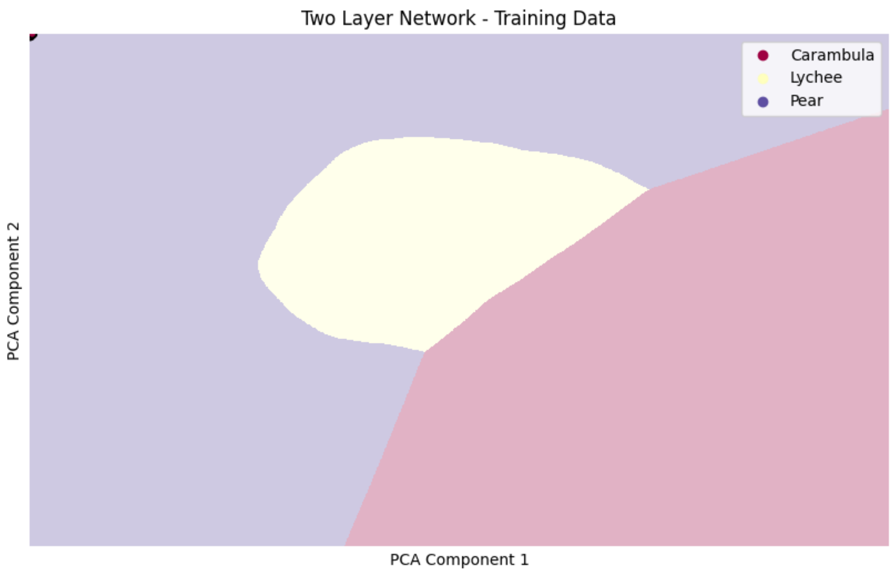
```
Testing: 100%|
Loss: 0.4252, Accuracy: 74.10%
```

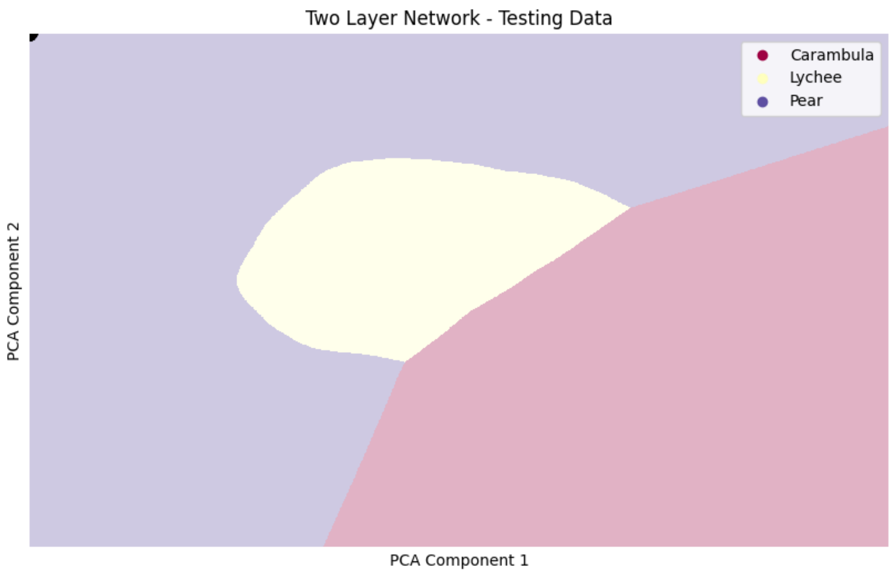
3. Plot training loss curves





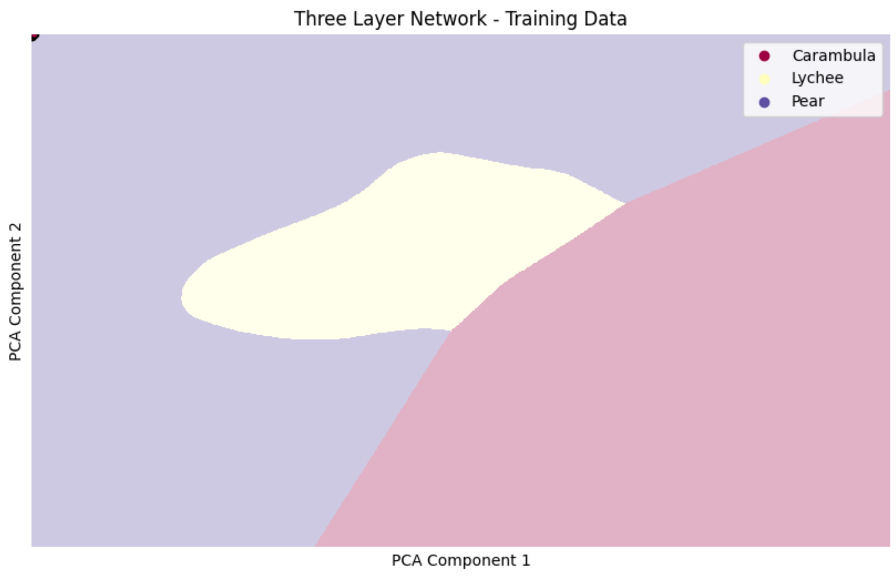
4. Plot decision regions for training / testing data, analyzing the result.

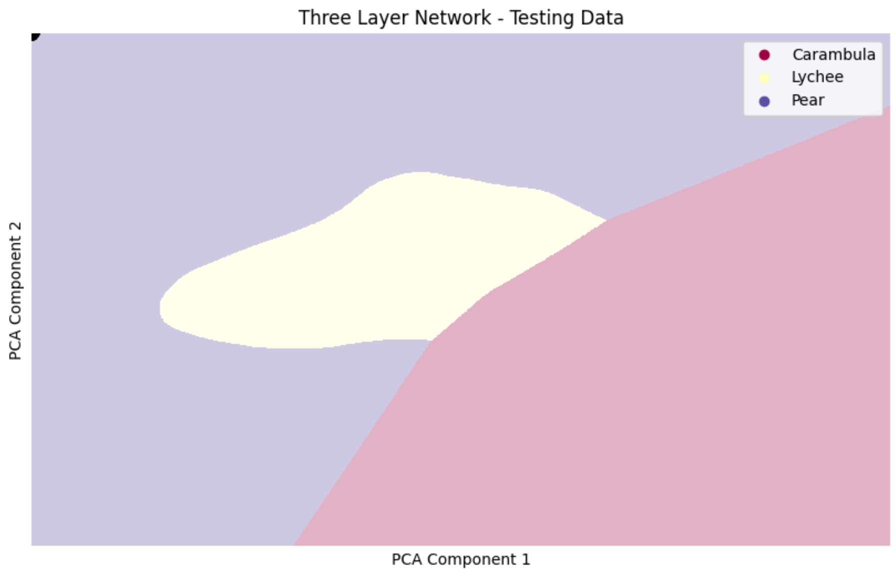




The picture above shows the decision boundary of the training set and training with Two-layer neural network. It can be seen that different categories occupy different areas, and the decision boundary is clear and smooth, indicating that the model can well distinguish the three categories in the training data.

The figure below is the decision boundary of the test set. The decision area of the test data is very similar to the training data, which means that the model's performance on the test data is consistent with the training data. This indicates that the model generalizes well to new data without significant overfitting or underfitting issues. Therefore, these two layers of neural networks can distinguish three kinds of fruits very well, and the decision-making area is reasonable. This type of model is suitable for such classification problems and has good performance.





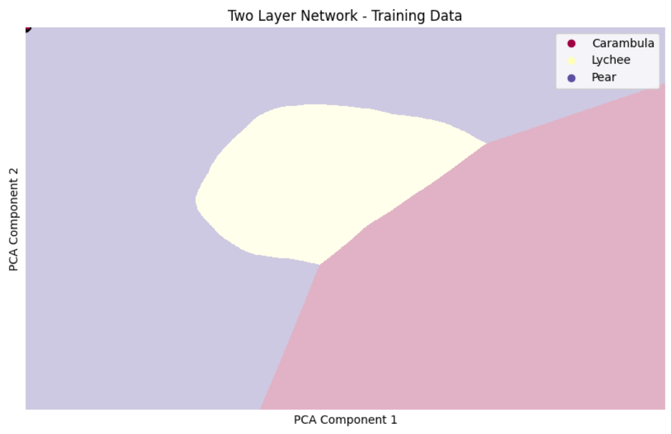
The three-layer neural network is good in Generalization and Boundary Stability.

The following compares two-layer neural network and three-layer neural network.

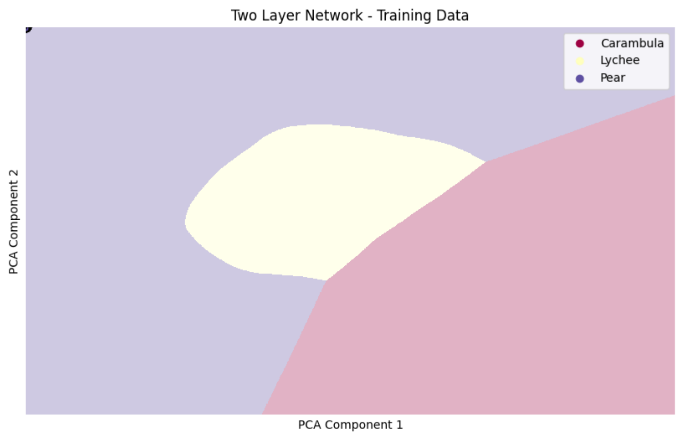
	Two-Layer neural network	Three-Layer neural network
Decision Boundary Complexity	Decision boundaries are smoother and simpler.	The decision boundaries are more complex, which means the network can learn more details and potentially separate different categories more accurately.
overfitting	Because its decision boundary is simpler, the two-layer network may be less prone to overfitting and may perform better on small and simple data sets.	The complexity of the decision boundary increases, which means that it may perform better on more complex data sets. If the amount of training data is insufficient, there may be a greater risk of overfitting.
Generalization	Due to its simpler decision boundary, this kind of network may generalize better when processing new data, but in some complex data patterns, the accuracy may not be as good as the three-layer network.	The decision boundary is more complex, which means it can better fit the training data, but be aware that overfitting problems may occur on the test data. If the model can successfully avoid overfitting, its generalization ability will be stronger.

4. compare original and dropout technique

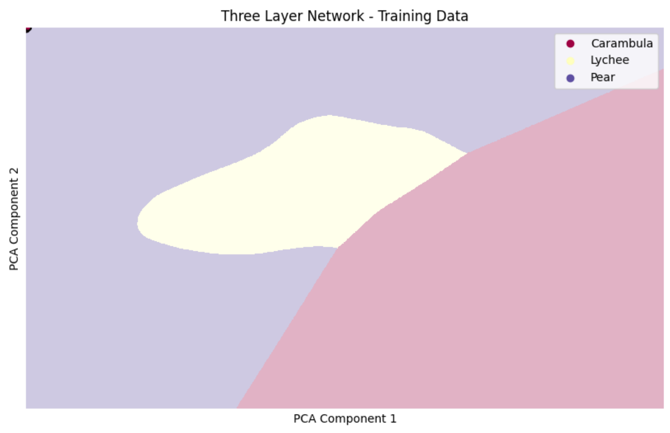
Original 2NN training data



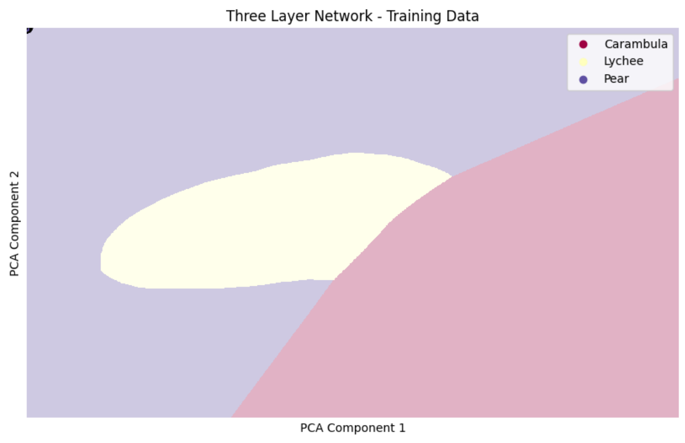
Dropout technique 2NN training data:



Original 3NN training data:

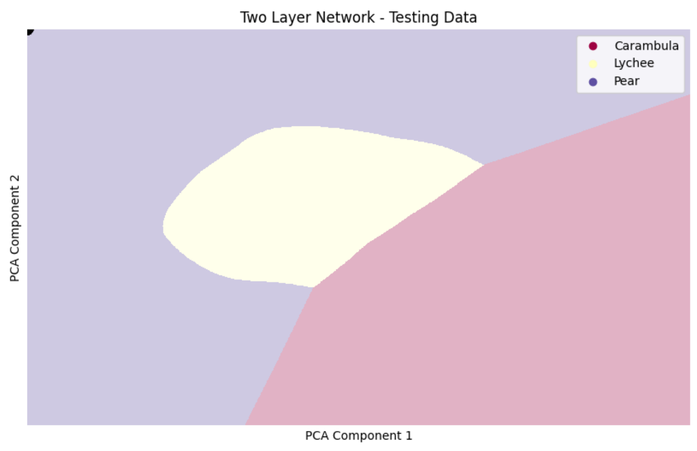


Dropout technique 3NN training data:

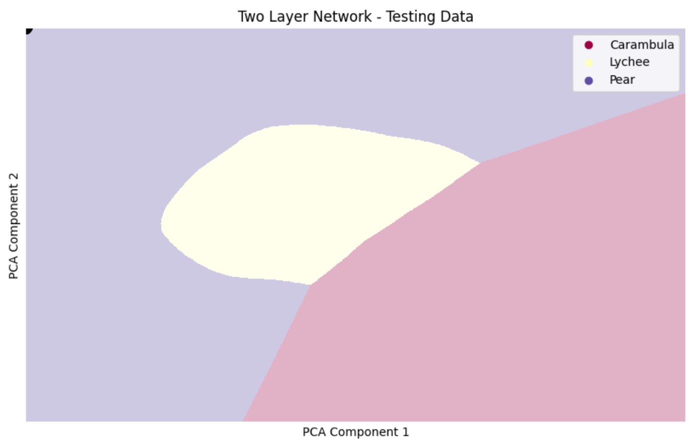


There is a Decision Region using Dropout technology. The part of the yellow area boundary is slightly smoother and the area is slightly larger than the original curve. This is because dropout technology omits some data from one layer and then passes it to the next layer of neural network. This may avoid overfitting.

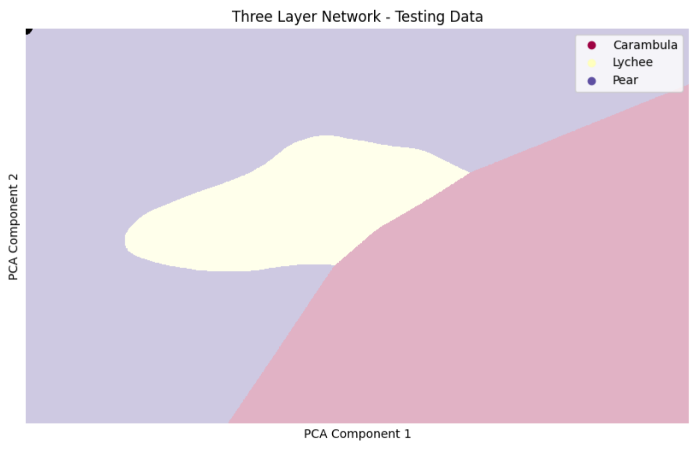
Original 2NN testing data:



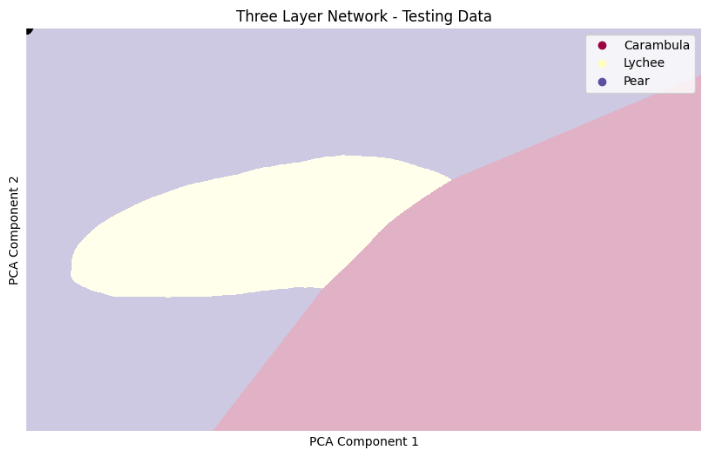
Dropout technique 2NN testing data:



Original 3NN testing data:



Dropout technique 3NN testing data:



From the original diagram, we can know that from 2 layer neural network to 3 layer neural network, the curves and areas will become more complex and larger, but if the coverage is too large, it will easily cause overfitting and the accuracy will decrease. You can know from the original results The accuracy rate of adding layers drops a little, and using dropout technology makes the curve become larger and slightly flatter as you can see in the figure. Therefore, there are models using dropout technology, and their accuracy rate is slightly higher. The following is a comparison.

	Original(%)	Dropout(%)
2 layer neural network	77.71	77.79
3 layer neural network	74.10	75.90