Problem 1. Develop an MLP classifier for the MNIST digit recognition dataset

In [4]:
```
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import numpy as np
import random

# for easier reading np
np.set_printoptions(precision=3,suppress=True)
```

Data

In [5]:
```
# Prepare the data
from torchvision.datasets import MNIST

db_train = MNIST(root="./", train=True, transform=None, target_transform=None, download=True)
db_test = MNIST(root="./", train=False, transform=None, target_transform=None, download=True)
def to1hot(labels):
    """Converts an array of class labels into their 1hot encodings.
    Assumes that there are at most three classes."""
    return torch.eye(10)[labels]

for i in range(10):
    img, lbl = db_train[i]
    plt.subplot(1, 10, i+1)
    plt.imshow(img)
    plt.title(lbl)
    plt.axis(False)
```

In [6]:
```
# Reading the dataset
def data_iter(batch_size, db):
    num_examples = len(db)

    # The examples are read at random, in no particular order
    indices = list(range(num_examples))
    random.shuffle(indices)
    indices = indices[:10000]
    for i in range(0, 10000, batch_size):
        X, Y = [], []
        for j in indices[i:i + batch_size]:
            img, lbl = db[j]

            # Process image
            img = torch.from_numpy(np.array(img))   # Convert PIL to numpy and then PyTorch
            img = img.view(28*28)                   # Image is 28x28. For our MLP we want to reshape this into a
            img = img.float() / 255.                # Make the pixel values between 0 and 1 (instead of between

            lbl = torch.tensor(lbl)

            X.append(img), Y.append(lbl)
        yield torch.stack(X), torch.stack(Y)

# Check data reader
for X_batch, y_batch in data_iter(batch_size=10, db=db_train):
    print('X_batch', X_batch.shape, X_batch.dtype)
    print('y_batch', y_batch.shape, y_batch.dtype)
    break
```

```
X_batch torch.Size([10, 784]) torch.float32
y_batch torch.Size([10]) torch.int64
```

Model

In [7]:
```
# Define Model
from torch import nn
class Linear(nn.Module):
    def __init__(self, input_dim):
        super(Linear, self).__init__()
        self.layer1 = nn.Linear(input_dim, 10)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.softmax(self.layer1(x))
        return x

# Check model
model = Linear(28*28)
for X_batch, y_batch in data_iter(batch_size=16, db=db_train):
    out_batch = model(X_batch)
    print('X_batch', X_batch.shape)
    print('out_batch', out_batch.shape)
    break
```

```
X_batch torch.Size([16, 784])
out_batch torch.Size([16, 10])
```

Training

In [8]:
```
# Optimization algorithms
def sgd(model, lr):
    """Minibatch stochastic gradient descent."""
    for p in model.parameters():
        p.data -= lr * p.grad
        p.grad = None

# Loss Functions and Accuracy Metric
def mse(y_hat, y):
    """MSE Loss."""
    loss_per_sample = (to1hot(y) - y_hat).pow(2).sum(1)
    return loss_per_sample.mean()

def accuracy(y_hat, y):
    return (y_hat.argmax(dim=1) == y).float().mean()

# Check functions
yhat = torch.tensor([[0.2, 0.8, 0.1, 0, 0, 0, 0, 0, 0, 0],[0.6, 0.3, 0.1, 0, 0, 0, 0, 0, 0, 0]])
y = torch.tensor([1, 1])
loss = mse(yhat, y)
acc = accuracy(yhat, y)
print(f'loss = {loss}    acc = {acc}')
```

```
loss = 0.4749999940395355    acc = 0.5
```

In [9]:
```
# Hyperparameters
lr = 0.01
batch_size = 16
num_epochs = 10
```

In [10]:
```
# Training
model = Linear(28*28)
# model = MLP(28*28)
for epoch in range(num_epochs):
    # Train for one epoch
    losses = []
    for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_train):
        # Use model to compute predictions
        yhat = model(X_batch)
        l = mse(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`

        # Compute gradients by back propagation
        l.backward()

        # Update parameters using their gradient
        sgd(model, lr)

        losses.append(l.detach().item())

    # Measure accuracy on the test set
    acc = []
    for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
        yhat = model(X_batch)
```

```
        acc.append(accuracy(yhat, y_batch))

    print(f"Epoch {epoch+1}: Train Loss {np.mean(losses):.3f} Test Accuracy {np.mean(acc):.3f}", flush=True)
```

```
Epoch 1: Train Loss 0.695 Test Accuracy 0.752
Epoch 2: Train Loss 0.418 Test Accuracy 0.839
Epoch 3: Train Loss 0.315 Test Accuracy 0.866
Epoch 4: Train Loss 0.273 Test Accuracy 0.875
Epoch 5: Train Loss 0.246 Test Accuracy 0.881
Epoch 6: Train Loss 0.227 Test Accuracy 0.885
Epoch 7: Train Loss 0.217 Test Accuracy 0.888
Epoch 8: Train Loss 0.210 Test Accuracy 0.891
Epoch 9: Train Loss 0.197 Test Accuracy 0.894
Epoch 10: Train Loss 0.199 Test Accuracy 0.895
```

In [11]:
```
# Evaluation
with torch.no_grad():
    yhat, y = [], []
    for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
        yhat.append(model(X_batch))
        y.append(y_batch)

yhat = torch.cat(yhat, dim=0).argmax(dim=1)
y = torch.cat(y, dim=0)
cm = to1hot(y).T@to1hot(yhat)
print('CM = \n', cm.numpy())
```

```
CM =
 [[ 959.    0.    2.    3.    0.    1.    7.    1.    7.    0.]
 [   0. 1103.    1.    6.    1.    1.    4.    0.   19.    0.]
 [  14.    2.  877.   20.   16.    1.   21.   26.   44.   11.]
 [   4.    0.   16.  905.    1.   26.    6.   17.   21.   14.]
 [   2.    7.    4.    0.  897.    1.   12.    1.   10.   48.]
 [  18.   10.    5.   48.   21.  694.   19.   13.   55.    9.]
 [  23.    3.    4.    1.   12.   18.  887.    0.   10.    0.]
 [   3.   22.   31.    2.   11.    0.    2.  916.    5.   36.]
 [  11.    9.   10.   28.   13.   20.   15.   16.  834.   18.]
 [  12.    8.    7.   11.   42.   19.    1.   24.    7.  878.]]
```

(a) What is the accuracy and loss of using linear regression, MSE and SGD?

Test accuracy is 89.7%, and training loss is 19.6%.

(b) Implement the cross-entropy loss. What is the accuracy and loss of using linear regression, cross-entropy loss and SGD?

Test accuracy is 90.4%, and Train Loss is 39.4%. Compared with (a), the accuracy rises.
MSE is mainly used for regression problems because it measures the error by calculating the squared difference between the predicted value and the target value. For classification tasks, MSE will generate unnecessary error calculations between categories, resulting in the gradient direction of the model being inaccurate.
In contrast, cross entropy directly compares the predicted probability distribution with the distribution of the true category. It can better guide the model to optimize in the correct classification direction, thereby converging to a reasonable classification boundary faster and more accurate.

In [12]:
```
# Loss Functions: Cross Entropy
def cross_entropy(y_hat, y):
    """Cross-entropy Loss."""
    y_hat = y_hat.clamp(1e-7, 1 - 1e-7)
    loss_per_sample = - (to1hot(y) * torch.log(y_hat)).sum(1)

    return loss_per_sample.mean()

# Check functions
yhat = torch.tensor([[0.2, 0.8, 0.1, 0, 0, 0, 0, 0, 0, 0],[0.6, 0.3, 0.1, 0, 0, 0, 0, 0, 0, 0]])
y = torch.tensor([1, 1])
loss = cross_entropy(yhat, y)
acc = accuracy(yhat, y)
print(f'loss = {loss}    acc = {acc}')
```

```
loss = 0.7135581970214844    acc = 0.5
```

In [13]:
```
# Training
model = Linear(28*28)

for epoch in range(num_epochs):
    # Train for one epoch
    losses = []
```

```
    for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_train):
        # Use model to compute predictions
        yhat = model(X_batch)
        l = cross_entropy(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`

        # Compute gradients by back propagation
        l.backward()

        # Update parameters using their gradient
        sgd(model, lr)

        losses.append(l.detach().item())

    # Measure accuracy on the test set
    acc = []
    for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
        yhat = model(X_batch)
        acc.append(accuracy(yhat, y_batch))

    print(f"Epoch {epoch+1}: Train Loss {np.mean(losses):.3f} Test Accuracy {np.mean(acc):.3f}", flush=True)
```

```
Epoch 1: Train Loss 1.137 Test Accuracy 0.853
Epoch 2: Train Loss 0.628 Test Accuracy 0.869
Epoch 3: Train Loss 0.532 Test Accuracy 0.883
Epoch 4: Train Loss 0.472 Test Accuracy 0.890
Epoch 5: Train Loss 0.458 Test Accuracy 0.892
Epoch 6: Train Loss 0.428 Test Accuracy 0.896
Epoch 7: Train Loss 0.411 Test Accuracy 0.898
Epoch 8: Train Loss 0.400 Test Accuracy 0.901
Epoch 9: Train Loss 0.395 Test Accuracy 0.902
Epoch 10: Train Loss 0.394 Test Accuracy 0.904
```

(c) Implement an MLP with 3 layers, with 350, 50 and 10 neurons at layers 1, 2 and 3, respectively. What is the accuracy and loss of using MLP, cross-entropy loss and SGD?

Testing Accuracy 92.5%, Training Loss is 28.8%. Compared with (b), the accuracy has increased and the loss has decreased.
Linear models can only fit the linear relationship of data. The distribution of handwritten digits in the MNIST dataset is very complex, and direct classification using linear regression will result in misclassification.
MLP uses a nonlinear activation function, ReLU in this case, to enable the model to learn the nonlinear relationship of the input data. This allows MLP to capture complex features and patterns, making it perform better than linear models on complex datasets.

In [14]:
```
# Define MLP Model
from torch import nn
class MLP(nn.Module):
    def __init__(self, input_dim):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, 350)
        self.fc2 = nn.Linear(350, 50)
        self.fc3 = nn.Linear(50, 10)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        x = self.softmax(x)
        return x

# Check model
model = MLP(28*28)
for X_batch, y_batch in data_iter(batch_size=16, db=db_train):
    out_batch = model(X_batch)
    print('X_batch', X_batch.shape)
    print('out_batch', out_batch.shape)
    break
```

```
X_batch torch.Size([16, 784])
out_batch torch.Size([16, 10])
```

In [15]:
```
# MLP Model
model = MLP(28*28)

for epoch in range(num_epochs):
```

```
    # Train for one epoch
    losses = []
    for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_train):
        # Use model to compute predictions
        yhat = model(X_batch)
        # l = mse(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`
        l = cross_entropy(yhat, y_batch)

        # Compute gradients by back propagation
        l.backward()

        # Update parameters using their gradient
        sgd(model, lr)

        losses.append(l.detach().item())

    # Measure accuracy on the test set
    acc = []
    for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
        yhat = model(X_batch)
        acc.append(accuracy(yhat, y_batch))

    print(f"Epoch {epoch+1}: Train Loss {np.mean(losses):.3f} Test Accuracy {np.mean(acc):.3f}", flush=True)
```

```
Epoch 1: Train Loss 1.988 Test Accuracy 0.722
Epoch 2: Train Loss 0.865 Test Accuracy 0.845
Epoch 3: Train Loss 0.515 Test Accuracy 0.882
Epoch 4: Train Loss 0.441 Test Accuracy 0.893
Epoch 5: Train Loss 0.384 Test Accuracy 0.900
Epoch 6: Train Loss 0.339 Test Accuracy 0.907
Epoch 7: Train Loss 0.331 Test Accuracy 0.914
Epoch 8: Train Loss 0.304 Test Accuracy 0.920
Epoch 9: Train Loss 0.294 Test Accuracy 0.922
Epoch 10: Train Loss 0.288 Test Accuracy 0.925
```

(d) Implement stochastic gradient descent with momentum. Use a momentum coefficient of 0.9. What is the accuracy and loss of using MLP, cross-entropy loss and SGD with momentum?

Testing Accuracy 96.7%, Training Loss is 9.2%. Compared with (c), the accuracy has increased and the loss has decreased.
Since the MNIST classification task involves a large number of parameters, the optimization process may fall into a local minimum.
SGD with Momentum can use the accumulated information of previous gradients to help the model break through the local minimum area and help find a better solution.
The existence of momentum can provide additional inertia to the model, allowing it to get rid of some low-value stable points and further improve the training effect.

In [27]:
```
# Optimization algorithms
def sgd_with_momentum(model, lr, velocity=None, momentum=0.9):
    """Minibatch stochastic gradient descent with momentum."""
    if velocity is None:
        # Initialize velocity to zero for each parameter
        velocity = {p: torch.zeros_like(p.data) for p in model.parameters()}

    for p in model.parameters():
        # Update velocity
        velocity[p] = momentum * velocity[p] + lr * p.grad
        # Update parameters
        p.data -= velocity[p]
        # Clear the gradients
        p.grad = None

    return velocity  # Return the updated velocity for next iteration
```

In [25]:
```
# MLP Model
model = MLP(28*28)

velocity = None

for epoch in range(num_epochs):
    # Train for one epoch
    losses = []
    for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_train):
        # Use model to compute predictions
        yhat = model(X_batch)
```

```
        # l = mse(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`
        l = cross_entropy(yhat, y_batch)

        # Compute gradients by back propagation
        l.backward()

        # Update parameters using their gradient
        # sgd(model, lr)
        velocity = sgd_with_momentum(model, lr, velocity)

        losses.append(l.detach().item())

    # Measure accuracy on the test set
    acc = []
    for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
        yhat = model(X_batch)
        acc.append(accuracy(yhat, y_batch))

    print(f"Epoch {epoch+1}: Train Loss {np.mean(losses):.3f} Test Accuracy {np.mean(acc):.3f}", flush=True)
```

```
Epoch 1: Train Loss 0.748 Test Accuracy 0.872
Epoch 2: Train Loss 0.291 Test Accuracy 0.922
Epoch 3: Train Loss 0.215 Test Accuracy 0.945
Epoch 4: Train Loss 0.166 Test Accuracy 0.956
Epoch 5: Train Loss 0.148 Test Accuracy 0.945
Epoch 6: Train Loss 0.115 Test Accuracy 0.965
Epoch 7: Train Loss 0.105 Test Accuracy 0.965
Epoch 8: Train Loss 0.104 Test Accuracy 0.961
Epoch 9: Train Loss 0.097 Test Accuracy 0.968
Epoch 10: Train Loss 0.092 Test Accuracy 0.967
```

(e) Training for 50 epochs

Testing Accuracy 97.8%, Training Loss is 1.3%. Compared with (d), the accuracy has increased and the loss has decreased.
After increasing the number of epochs, the model will continue to converge near a lower loss value and find a better parameter configuration than 10 times.
Thus, a lower training loss and higher accuracy are obtained. Therefore, better training results can be obtained by increasing the number of epochs. However, when the epoch reaches 47, the model has basically reached the optimal value, so it can be expected that blindly increasing the epoch may bring unnecessary computing time. The best way to deal with it is to add judgment conditions and stop training when the performance of the validation set no longer improves.

In [26]:
```
# MLP Model
model = MLP(28*28)

velocity = None
num_epochs = 50

for epoch in range(num_epochs):
    # Train for one epoch
    losses = []
    for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_train):
        # Use model to compute predictions
        yhat = model(X_batch)
        # l = mse(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`
        l = cross_entropy(yhat, y_batch)

        # Compute gradients by back propagation
        l.backward()

        # Update parameters using their gradient
        # sgd(model, lr)
        velocity = sgd_with_momentum(model, lr, velocity)

        losses.append(l.detach().item())

    # Measure accuracy on the test set
    acc = []
    for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
        yhat = model(X_batch)
        acc.append(accuracy(yhat, y_batch))

    print(f"Epoch {epoch+1}: Train Loss {np.mean(losses):.3f} Test Accuracy {np.mean(acc):.3f}", flush=True)
```

```
Epoch 1: Train Loss 0.727 Test Accuracy 0.898
Epoch 2: Train Loss 0.282 Test Accuracy 0.938
Epoch 3: Train Loss 0.214 Test Accuracy 0.948
Epoch 4: Train Loss 0.162 Test Accuracy 0.948
Epoch 5: Train Loss 0.148 Test Accuracy 0.959
Epoch 6: Train Loss 0.127 Test Accuracy 0.962
Epoch 7: Train Loss 0.110 Test Accuracy 0.965
Epoch 8: Train Loss 0.106 Test Accuracy 0.967
Epoch 9: Train Loss 0.098 Test Accuracy 0.964
Epoch 10: Train Loss 0.085 Test Accuracy 0.970
Epoch 11: Train Loss 0.089 Test Accuracy 0.970
Epoch 12: Train Loss 0.074 Test Accuracy 0.972
Epoch 13: Train Loss 0.075 Test Accuracy 0.973
Epoch 14: Train Loss 0.064 Test Accuracy 0.974
Epoch 15: Train Loss 0.068 Test Accuracy 0.971
Epoch 16: Train Loss 0.065 Test Accuracy 0.974
Epoch 17: Train Loss 0.058 Test Accuracy 0.973
Epoch 18: Train Loss 0.060 Test Accuracy 0.977
Epoch 19: Train Loss 0.053 Test Accuracy 0.974
Epoch 20: Train Loss 0.052 Test Accuracy 0.976
Epoch 21: Train Loss 0.052 Test Accuracy 0.977
Epoch 22: Train Loss 0.044 Test Accuracy 0.975
Epoch 23: Train Loss 0.042 Test Accuracy 0.975
Epoch 24: Train Loss 0.048 Test Accuracy 0.976
Epoch 25: Train Loss 0.043 Test Accuracy 0.978
Epoch 26: Train Loss 0.041 Test Accuracy 0.980
Epoch 27: Train Loss 0.039 Test Accuracy 0.979
Epoch 28: Train Loss 0.031 Test Accuracy 0.975
Epoch 29: Train Loss 0.037 Test Accuracy 0.977
Epoch 30: Train Loss 0.029 Test Accuracy 0.978
Epoch 31: Train Loss 0.029 Test Accuracy 0.978
Epoch 32: Train Loss 0.035 Test Accuracy 0.978
Epoch 33: Train Loss 0.027 Test Accuracy 0.979
Epoch 34: Train Loss 0.024 Test Accuracy 0.978
Epoch 35: Train Loss 0.029 Test Accuracy 0.980
Epoch 36: Train Loss 0.023 Test Accuracy 0.981
Epoch 37: Train Loss 0.028 Test Accuracy 0.980
Epoch 38: Train Loss 0.024 Test Accuracy 0.977
Epoch 39: Train Loss 0.026 Test Accuracy 0.978
Epoch 40: Train Loss 0.026 Test Accuracy 0.981
Epoch 41: Train Loss 0.025 Test Accuracy 0.976
Epoch 42: Train Loss 0.021 Test Accuracy 0.980
Epoch 43: Train Loss 0.024 Test Accuracy 0.979
Epoch 44: Train Loss 0.020 Test Accuracy 0.980
Epoch 45: Train Loss 0.022 Test Accuracy 0.981
Epoch 46: Train Loss 0.021 Test Accuracy 0.981
Epoch 47: Train Loss 0.019 Test Accuracy 0.979
Epoch 48: Train Loss 0.012 Test Accuracy 0.978
Epoch 49: Train Loss 0.012 Test Accuracy 0.981
Epoch 50: Train Loss 0.013 Test Accuracy 0.978
```

In [ ]:

Problem 2. Develop a deep multi-layer perceptron model for the iris dataset.

In [2]:
```
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import numpy as np
import random

# for easier reading np
np.set_printoptions(precision=3,suppress=True)
```

Data

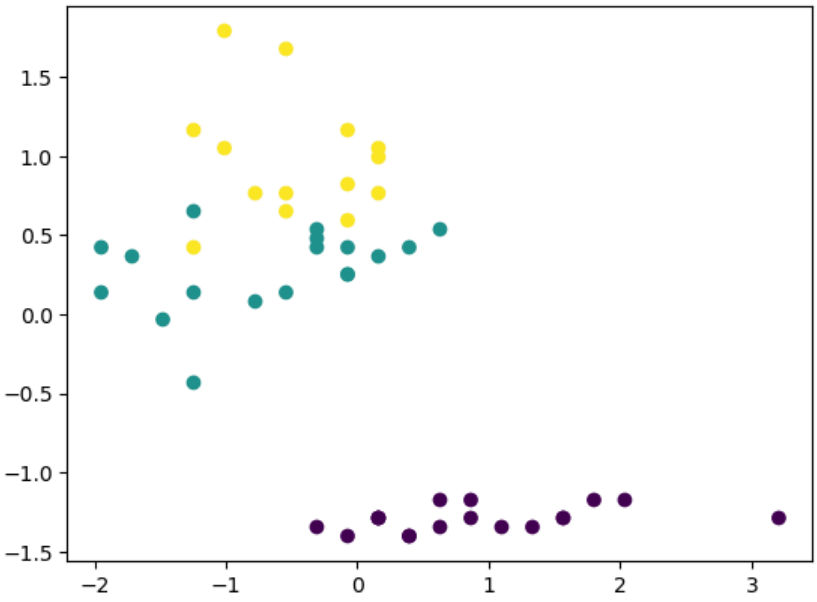
In [3]:
```
# Prepare the data
def to1hot(labels):
    """Converts an array of class labels into their 1hot encodings.
    Assumes that there are at most 3 classes."""
    return torch.eye(3)[labels]

class Dataset:
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

    def __len__(self):
        return len(self.X)

from sklearn import datasets
iris = datasets.load_iris()
X, y = iris.data, iris.target

# Split into train/test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, stratify=y)

# Normalize features
mu = X_train.mean(0, keepdims=True)
std = X_train.std(0, keepdims=True)
X_train = (X_train - mu) / std
X_test = (X_test - mu) / std

db_train = Dataset(X_train, y_train)
db_test = Dataset(X_test, y_test)
print(len(db_train), db_train[0])
print(len(db_test), db_test[0])
```

```
120 (array([0.211, 0.863, 0.423, 0.533]), 1)
30 (array([-0.741,  2.503, -1.286, -1.453]), 0)
```

In [4]:
```
# Reading the dataset
def data_iter(batch_size, db):
    num_examples = len(db)

    # The examples are read at random, in no particular order
    indices = list(range(num_examples))
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        X, Y = [], []
        for j in indices[i:i + batch_size]:
            x, lbl = db[j]

            # Process image
            x = torch.from_numpy(x).float()
            lbl = torch.tensor(lbl).long()

            X.append(x), Y.append(lbl)
        yield torch.stack(X), torch.stack(Y)

# Check data reader
for X_batch, y_batch in data_iter(batch_size=50, db=db_train):
```

```
    print('X_batch', X_batch.shape)
    print('y_batch', y_batch.shape)
    plt.scatter(X_batch[:, 1].numpy(), X_batch[:, 2].numpy(), c=y_batch.numpy())
    break
```

```
X_batch torch.Size([50, 4])
y_batch torch.Size([50])
```



Model

In [5]:

```
# Define Model
from torch import nn

class DNN(nn.Module):
    def __init__(self, input_dim, hidden_dim=30, num_classes=3, num_layers=2):
        super(DNN, self).__init__()

        # Define layers
        layers = [nn.Linear(input_dim, hidden_dim)]
        for i in range(1, num_layers-1):
            layers.append(nn.Linear(hidden_dim, hidden_dim))
        layers.append(nn.Linear(hidden_dim, num_classes))
        self.layers = nn.ModuleList(layers)
        self.relu = nn.ReLU()

        # Initialize weights
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, mean=0, std=0.01)

    def forward(self, x):
        for i, layer in enumerate(self.layers):
            if i < len(self.layers) - 1:
                x = self.relu(layer(x))
            else:
                # Do not apply relu to the last layer
                x = layer(x)
        return x

# Check model
model = DNN(input_dim=4, num_classes=3, hidden_dim=30, num_layers=2)
for X_batch, y_batch in data_iter(batch_size=8, db=db_train):
    print('X_batch', X_batch.shape)
    out_batch = model(X_batch)
    print('out_batch', out_batch.shape, out_batch)
    break
```

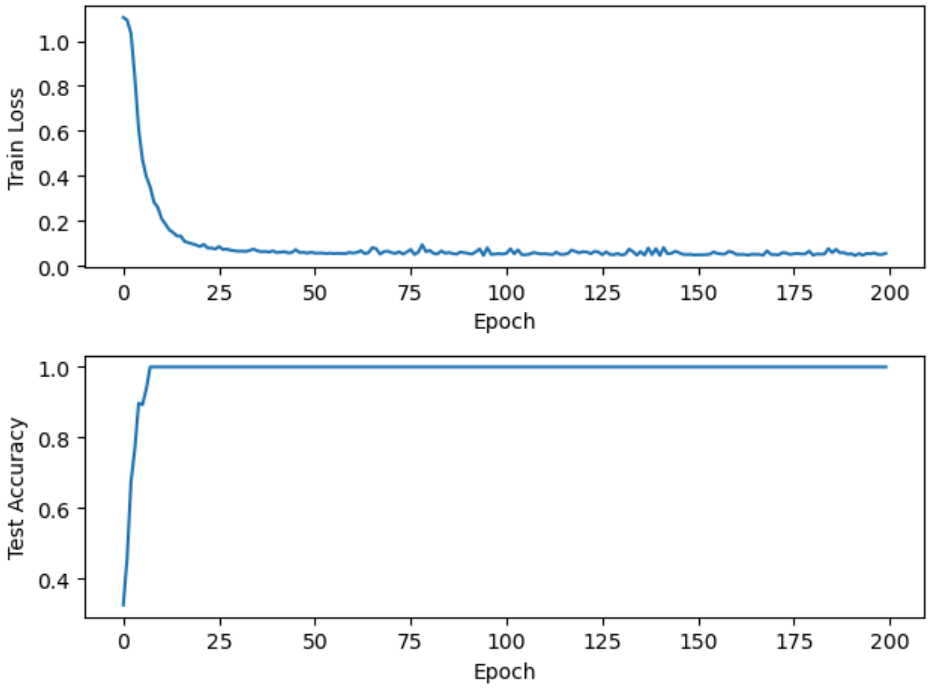
```
X_batch torch.Size([8, 4])
out_batch torch.Size([8, 3]) tensor([[ 0.1531,  0.1313, -0.0784],
        [ 0.1535,  0.1307, -0.0776],
        [ 0.1536,  0.1305, -0.0774],
        [ 0.1532,  0.1319, -0.0781],
        [ 0.1529,  0.1315, -0.0788],
        [ 0.1527,  0.1336, -0.0787],
        [ 0.1532,  0.1302, -0.0782],
        [ 0.1536,  0.1298, -0.0773]], grad_fn=<AddmmBackward0>)
```

Training

In [6]:

```python
# Training
lr = 0.03
batch_size = 16
num_epochs = 200

model = DNN(input_dim=4, num_classes=3, hidden_dim=30, num_layers=2)
cross_entropy = nn.CrossEntropyLoss()
opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)
accuracy = lambda y_hat, y: (y_hat.argmax(dim=1) == y).float().mean()

print("Start training")
all_losses, all_accuracies = [], []
for epoch in range(num_epochs):
    # Train for one epoch
    losses = []
    for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_train):
        # Use model to compute predictions
        yhat = model(X_batch)
        l = cross_entropy(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`

        # Compute gradients by back propagation
        l.backward()

        # Update parameters using their gradient
        opt.step()
        opt.zero_grad()
        losses.append(l.detach().item())

    # Measure accuracy on the test set
    acc = []
    for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
        yhat = model(X_batch)
        acc.append(accuracy(yhat, y_batch))

    all_losses.append(np.mean(losses))
    all_accuracies.append(np.mean(acc))

    if (epoch+1) % 10 == 0:
        # print(yhat)
        print(f"Epoch {epoch+1}: Train Loss {np.mean(losses):.3f} Test Accuracy {np.mean(acc):.3f}", flush=True)

# Evaluation
print("\nStart evaluation")
with torch.no_grad():
    yhat, y = [], []
    for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
        yhat.append(model(X_batch))
        y.append(y_batch)

yhat = torch.cat(yhat, dim=0).argmax(dim=1)
y = torch.cat(y, dim=0)
cm = to1hot(y).T@to1hot(yhat)
print('CM = \n', cm.numpy())

plt.subplot(2, 1, 1)
plt.plot(all_losses)
plt.ylabel('Train Loss');
plt.xlabel('Epoch');
plt.subplot(2, 1, 2)
plt.plot(all_accuracies)
plt.ylabel('Test Accuracy');
plt.xlabel('Epoch');
plt.tight_layout()
plt.show()
```

```
Start training
Epoch 10: Train Loss 0.259 Test Accuracy 1.000
Epoch 20: Train Loss 0.092 Test Accuracy 1.000
Epoch 30: Train Loss 0.066 Test Accuracy 1.000
Epoch 40: Train Loss 0.066 Test Accuracy 1.000
Epoch 50: Train Loss 0.060 Test Accuracy 1.000
Epoch 60: Train Loss 0.059 Test Accuracy 1.000
Epoch 70: Train Loss 0.065 Test Accuracy 1.000
Epoch 80: Train Loss 0.062 Test Accuracy 1.000
Epoch 90: Train Loss 0.058 Test Accuracy 1.000
Epoch 100: Train Loss 0.052 Test Accuracy 1.000
Epoch 110: Train Loss 0.052 Test Accuracy 1.000
Epoch 120: Train Loss 0.057 Test Accuracy 1.000
Epoch 130: Train Loss 0.055 Test Accuracy 1.000
Epoch 140: Train Loss 0.076 Test Accuracy 1.000
Epoch 150: Train Loss 0.048 Test Accuracy 1.000
Epoch 160: Train Loss 0.060 Test Accuracy 1.000
Epoch 170: Train Loss 0.051 Test Accuracy 1.000
Epoch 180: Train Loss 0.065 Test Accuracy 1.000
Epoch 190: Train Loss 0.051 Test Accuracy 1.000
Epoch 200: Train Loss 0.055 Test Accuracy 1.000

Start evaluation
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



(1) How many epochs was required to achieve convergence?

Every time I ran the code, the result is different.
The training loss converges after like 90-100 epochs, the test accuracy seems to keep fluctuating after 100 epochs.

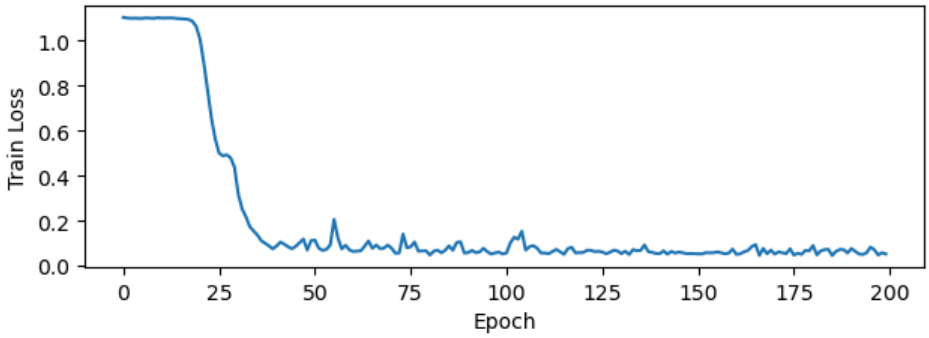
(2) Observe what happens as you increase the number of layers. Train the model with 3 layers, 4 layers, 5 layers, 6 layers. Why did the model stop learning?

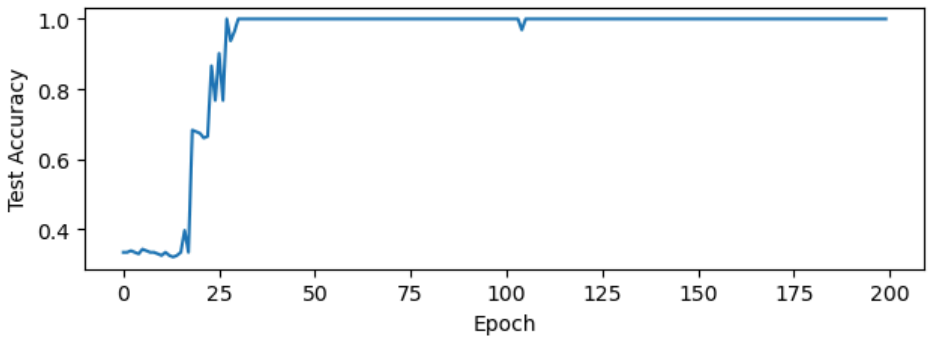
As the number of layers increases, the gradient calculated during back propagation may continue to decrease due to the chain rule, causing the parameter updates of the previous layers to become negligible, making it difficult for the model to learn effective features.
Also, for deep networks, as the depth of the network increases, the loss surface becomes more and more complex, and the optimization process becomes more difficult. Standard deep networks may fall into local optimality or cause training instability.

In [7]:
```python
# Change number of layers into 3\4\5\6
for l_num in range(3, 7):

    model = DNN(input_dim=4, num_classes=3, hidden_dim=30, num_layers=l_num)
    cross_entropy = nn.CrossEntropyLoss()
```

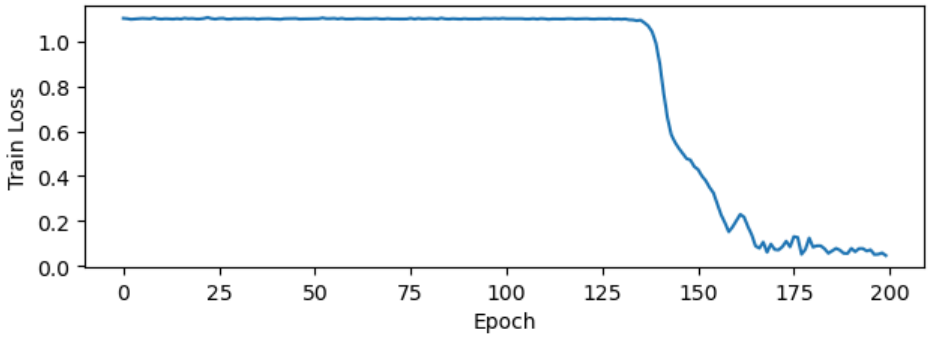
```
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)
    accuracy = lambda y_hat, y: (y_hat.argmax(dim=1) == y).float().mean()

    print(f"Start training using DNN with {l_num} layers")
    all_losses, all_accuracies = [], []
    for epoch in range(num_epochs):
        # Train for one epoch
        losses = []
        for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_train):
            # Use model to compute predictions
            yhat = model(X_batch)
            l = cross_entropy(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`

            # Compute gradients by back propagation
            l.backward()

            # Update parameters using their gradient
            opt.step()
            opt.zero_grad()
            losses.append(l.detach().item())

        # Measure accuracy on the test set
        acc = []
        for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
            yhat = model(X_batch)
            acc.append(accuracy(yhat, y_batch))

        all_losses.append(np.mean(losses))
        all_accuracies.append(np.mean(acc))

        if (epoch+1) % 10 == 0:
            # print(yhat)
            print(f"Epoch {epoch+1}: Train Loss {np.mean(losses):.3f} Test Accuracy {np.mean(acc):.3f}", flush=Tr

    # Evaluation
    print(f"\nStart evaluation DNN with {l_num} layers")
    with torch.no_grad():
        yhat, y = [], []
        for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
            yhat.append(model(X_batch))
            y.append(y_batch)

    yhat = torch.cat(yhat, dim=0).argmax(dim=1)
    y = torch.cat(y, dim=0)
    cm = to1hot(y).T@to1hot(yhat)
    print('CM = \n', cm.numpy())

    plt.subplot(2, 1, 1)
    plt.plot(all_losses)
    plt.ylabel('Train Loss')
    plt.xlabel('Epoch')
    plt.subplot(2, 1, 2)
    plt.plot(all_accuracies)
    plt.ylabel('Test Accuracy')
    plt.xlabel('Epoch')
    plt.tight_layout()
    plt.show()
```

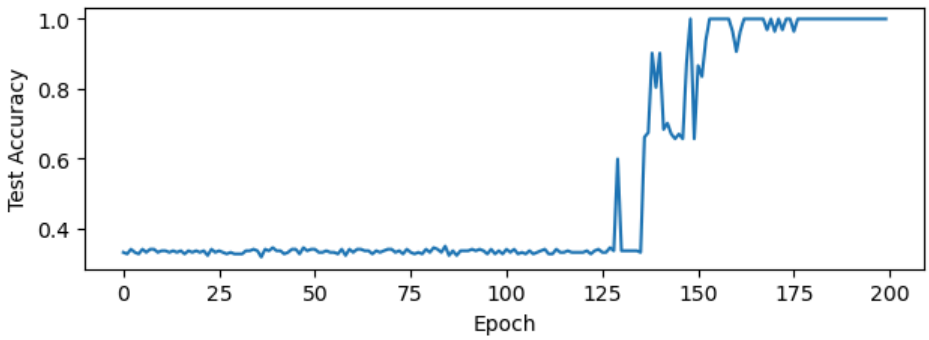
```
Start training using DNN with 3 layers
Epoch 10: Train Loss 1.102 Test Accuracy 0.330
Epoch 20: Train Loss 1.064 Test Accuracy 0.679
Epoch 30: Train Loss 0.438 Test Accuracy 0.964
Epoch 40: Train Loss 0.074 Test Accuracy 1.000
Epoch 50: Train Loss 0.111 Test Accuracy 1.000
Epoch 60: Train Loss 0.070 Test Accuracy 1.000
Epoch 70: Train Loss 0.091 Test Accuracy 1.000
Epoch 80: Train Loss 0.066 Test Accuracy 1.000
Epoch 90: Train Loss 0.055 Test Accuracy 1.000
Epoch 100: Train Loss 0.052 Test Accuracy 1.000
Epoch 110: Train Loss 0.056 Test Accuracy 1.000
Epoch 120: Train Loss 0.058 Test Accuracy 1.000
Epoch 130: Train Loss 0.064 Test Accuracy 1.000
Epoch 140: Train Loss 0.053 Test Accuracy 1.000
Epoch 150: Train Loss 0.053 Test Accuracy 1.000
Epoch 160: Train Loss 0.073 Test Accuracy 1.000
Epoch 170: Train Loss 0.069 Test Accuracy 1.000
Epoch 180: Train Loss 0.065 Test Accuracy 1.000
Epoch 190: Train Loss 0.056 Test Accuracy 1.000
Epoch 200: Train Loss 0.051 Test Accuracy 1.000

Start evaluation DNN with 3 layers
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



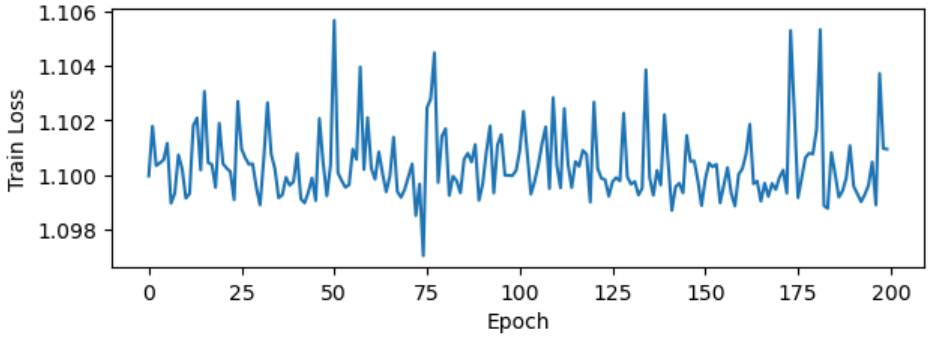


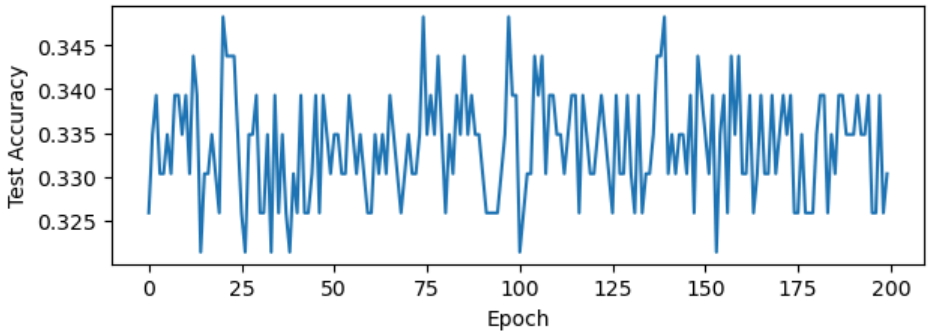
```
Start training using DNN with 4 layers
Epoch 10: Train Loss 1.101 Test Accuracy 0.330
Epoch 20: Train Loss 1.100 Test Accuracy 0.335
Epoch 30: Train Loss 1.101 Test Accuracy 0.326
Epoch 40: Train Loss 1.100 Test Accuracy 0.344
Epoch 50: Train Loss 1.100 Test Accuracy 0.339
Epoch 60: Train Loss 1.100 Test Accuracy 0.339
Epoch 70: Train Loss 1.102 Test Accuracy 0.339
Epoch 80: Train Loss 1.101 Test Accuracy 0.339
Epoch 90: Train Loss 1.101 Test Accuracy 0.335
Epoch 100: Train Loss 1.103 Test Accuracy 0.326
Epoch 110: Train Loss 1.102 Test Accuracy 0.335
Epoch 120: Train Loss 1.100 Test Accuracy 0.330
Epoch 130: Train Loss 1.100 Test Accuracy 0.598
Epoch 140: Train Loss 0.992 Test Accuracy 0.804
Epoch 150: Train Loss 0.442 Test Accuracy 0.656
Epoch 160: Train Loss 0.174 Test Accuracy 0.964
Epoch 170: Train Loss 0.097 Test Accuracy 1.000
Epoch 180: Train Loss 0.124 Test Accuracy 1.000
Epoch 190: Train Loss 0.055 Test Accuracy 1.000
Epoch 200: Train Loss 0.046 Test Accuracy 1.000

Start evaluation DNN with 4 layers
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



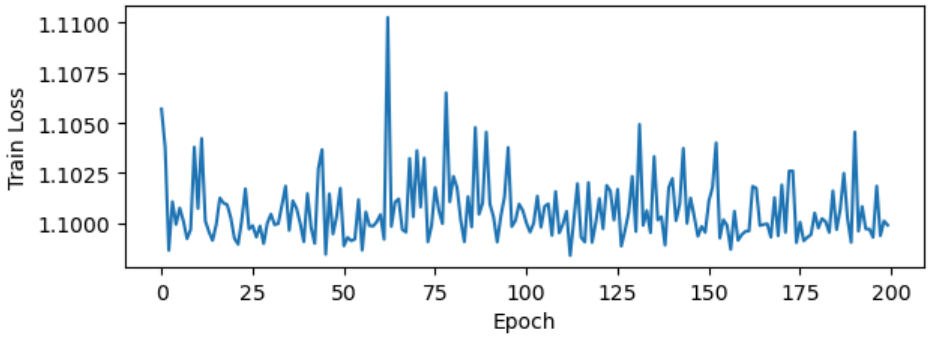


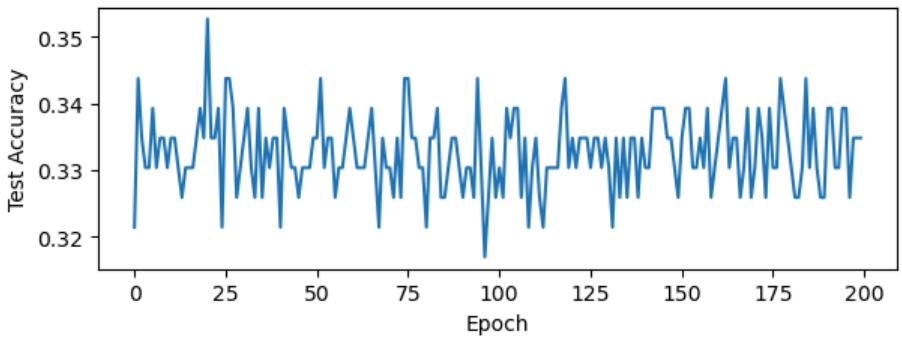
```
Start training using DNN with 5 layers
Epoch 10: Train Loss 1.100 Test Accuracy 0.335
Epoch 20: Train Loss 1.102 Test Accuracy 0.326
Epoch 30: Train Loss 1.100 Test Accuracy 0.339
Epoch 40: Train Loss 1.100 Test Accuracy 0.330
Epoch 50: Train Loss 1.100 Test Accuracy 0.330
Epoch 60: Train Loss 1.102 Test Accuracy 0.326
Epoch 70: Train Loss 1.099 Test Accuracy 0.330
Epoch 80: Train Loss 1.101 Test Accuracy 0.335
Epoch 90: Train Loss 1.099 Test Accuracy 0.335
Epoch 100: Train Loss 1.100 Test Accuracy 0.339
Epoch 110: Train Loss 1.103 Test Accuracy 0.339
Epoch 120: Train Loss 1.099 Test Accuracy 0.330
Epoch 130: Train Loss 1.100 Test Accuracy 0.339
Epoch 140: Train Loss 1.102 Test Accuracy 0.348
Epoch 150: Train Loss 1.099 Test Accuracy 0.339
Epoch 160: Train Loss 1.100 Test Accuracy 0.344
Epoch 170: Train Loss 1.099 Test Accuracy 0.330
Epoch 180: Train Loss 1.101 Test Accuracy 0.326
Epoch 190: Train Loss 1.101 Test Accuracy 0.335
Epoch 200: Train Loss 1.101 Test Accuracy 0.330

Start evaluation DNN with 5 layers
CM =
 [[ 0.  0. 10.]
 [ 0.  0. 10.]
 [ 0.  0. 10.]]
```





```
Start training using DNN with 6 layers
Epoch 10: Train Loss 1.104 Test Accuracy 0.330
Epoch 20: Train Loss 1.100 Test Accuracy 0.335
Epoch 30: Train Loss 1.100 Test Accuracy 0.330
Epoch 40: Train Loss 1.099 Test Accuracy 0.335
Epoch 50: Train Loss 1.102 Test Accuracy 0.335
Epoch 60: Train Loss 1.100 Test Accuracy 0.339
Epoch 70: Train Loss 1.100 Test Accuracy 0.330
Epoch 80: Train Loss 1.101 Test Accuracy 0.330
Epoch 90: Train Loss 1.105 Test Accuracy 0.330
Epoch 100: Train Loss 1.101 Test Accuracy 0.326
Epoch 110: Train Loss 1.100 Test Accuracy 0.330
Epoch 120: Train Loss 1.100 Test Accuracy 0.330
Epoch 130: Train Loss 1.102 Test Accuracy 0.335
Epoch 140: Train Loss 1.102 Test Accuracy 0.335
Epoch 150: Train Loss 1.100 Test Accuracy 0.326
Epoch 160: Train Loss 1.099 Test Accuracy 0.330
Epoch 170: Train Loss 1.099 Test Accuracy 0.326
Epoch 180: Train Loss 1.101 Test Accuracy 0.335
Epoch 190: Train Loss 1.099 Test Accuracy 0.326
Epoch 200: Train Loss 1.100 Test Accuracy 0.335

Start evaluation DNN with 6 layers
CM =
 [[ 0.  0. 10.]
 [ 0.  0. 10.]
 [ 0.  0. 10.]]
```





(3) Implement a new model called Residual DNN. This model should be the same as the DNN model, but instead of computing the output of each layer by `x = self.relu(layer(x))` we will add a residual connection `x = x + self.relu(layer(x))`. Can all layers have a residual connection?

The input and output layers can't have residual connections because the input should not have other information directly added to it, and the output is usually the final result, which does not require residual adjustment.

Define Residual DNN model

In [8]:
```python
# Define Model
from torch import nn

class Residual_DNN(nn.Module):
    def __init__(self, input_dim, hidden_dim=30, num_classes=3, num_layers=2):
        super(Residual_DNN, self).__init__()

        # Define layers
        layers = [nn.Linear(input_dim, hidden_dim)]
        for i in range(1, num_layers-1):
            layers.append(nn.Linear(hidden_dim, hidden_dim))
```

```
        layers.append(nn.Linear(hidden_dim, num_classes))
        self.layers = nn.ModuleList(layers)
        self.relu = nn.ReLU()

        # Initialize weights
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, mean=0, std=0.01)

    def forward(self, x):
        # First layer
        x_res = self.layers[0](x)
        x = self.relu(x_res)

        for i in range(1, len(self.layers) - 1):
            x_res = self.layers[i](x)
            x = x + self.relu(x_res)  # Residual connection
            x = self.relu(x)  # Apply ReLU after the residual addition

        # Final layer without ReLU
        x = self.layers[-1](x)
        return x

# Check model
model = Residual_DNN(input_dim=4, num_classes=3, hidden_dim=30, num_layers=2)
for X_batch, y_batch in data_iter(batch_size=8, db=db_train):
    print('X_batch', X_batch.shape)
    out_batch = model(X_batch)
    print('out_batch', out_batch.shape, out_batch)
    break
```

```
X_batch torch.Size([8, 4])
out_batch torch.Size([8, 3]) tensor([[ 0.0555, -0.0020,  0.1179],
        [ 0.0536, -0.0016,  0.1206],
        [ 0.0525, -0.0010,  0.1214],
        [ 0.0532, -0.0010,  0.1198],
        [ 0.0530, -0.0012,  0.1199],
        [ 0.0527, -0.0010,  0.1208],
        [ 0.0528, -0.0010,  0.1204],
        [ 0.0539, -0.0016,  0.1199]], grad_fn=<AddmmBackward0>)
```

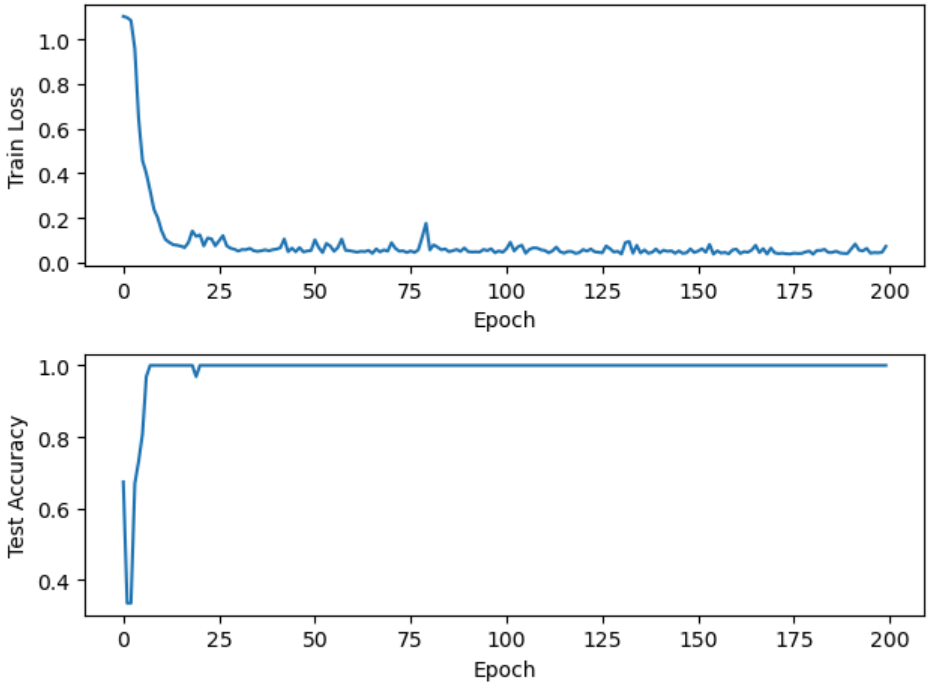
Training and evaluating

In [9]:

```
# Change number of layers into 3\4\5\6
model = Residual_DNN(input_dim=4, num_classes=3, hidden_dim=30, num_layers=4)
cross_entropy = nn.CrossEntropyLoss()
opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)
accuracy = lambda y_hat, y: (y_hat.argmax(dim=1) == y).float().mean()

print(f"Start training using Residual DNN with 4 layers")
all_losses, all_accuracies = [], []
for epoch in range(num_epochs):
    # Train for one epoch
    losses = []
    for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_train):
        # Use model to compute predictions
        yhat = model(X_batch)
        l = cross_entropy(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`

        # Compute gradients by back propagation
        l.backward()

        # Update parameters using their gradient
        opt.step()
        opt.zero_grad()
        losses.append(l.detach().item())

    # Measure accuracy on the test set
    acc = []
    for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
        yhat = model(X_batch)
        acc.append(accuracy(yhat, y_batch))

    all_losses.append(np.mean(losses))
    all_accuracies.append(np.mean(acc))

    if (epoch+1) % 10 == 0:
        # print(yhat)
        print(f"Epoch {epoch+1}: Train Loss {np.mean(losses):.3f} Test Accuracy {np.mean(acc):.3f}", flush=True)
```

```python
# Evaluation
print("\nStart evaluation Residual DNN with 4 layers")
with torch.no_grad():
    yhat, y = [], []
    for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
        yhat.append(model(X_batch))
        y.append(y_batch)

yhat = torch.cat(yhat, dim=0).argmax(dim=1)
y = torch.cat(y, dim=0)
cm = to1hot(y).T@to1hot(yhat)
print('CM = \n', cm.numpy())

plt.subplot(2, 1, 1)
plt.plot(all_losses)
plt.ylabel('Train Loss')
plt.xlabel('Epoch')
plt.subplot(2, 1, 2)
plt.plot(all_accuracies)
plt.ylabel('Test Accuracy')
plt.xlabel('Epoch')
plt.tight_layout()
plt.show()
```

```
Start training using Residual DNN with 4 layers
Epoch 10: Train Loss 0.199 Test Accuracy 1.000
Epoch 20: Train Loss 0.117 Test Accuracy 0.969
Epoch 30: Train Loss 0.059 Test Accuracy 1.000
Epoch 40: Train Loss 0.057 Test Accuracy 1.000
Epoch 50: Train Loss 0.053 Test Accuracy 1.000
Epoch 60: Train Loss 0.053 Test Accuracy 1.000
Epoch 70: Train Loss 0.051 Test Accuracy 1.000
Epoch 80: Train Loss 0.175 Test Accuracy 1.000
Epoch 90: Train Loss 0.065 Test Accuracy 1.000
Epoch 100: Train Loss 0.045 Test Accuracy 1.000
Epoch 110: Train Loss 0.056 Test Accuracy 1.000
Epoch 120: Train Loss 0.044 Test Accuracy 1.000
Epoch 130: Train Loss 0.050 Test Accuracy 1.000
Epoch 140: Train Loss 0.060 Test Accuracy 1.000
Epoch 150: Train Loss 0.044 Test Accuracy 1.000
Epoch 160: Train Loss 0.056 Test Accuracy 1.000
Epoch 170: Train Loss 0.064 Test Accuracy 1.000
Epoch 180: Train Loss 0.051 Test Accuracy 1.000
Epoch 190: Train Loss 0.040 Test Accuracy 1.000
Epoch 200: Train Loss 0.072 Test Accuracy 1.000

Start evaluation Residual DNN with 4 layers
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```

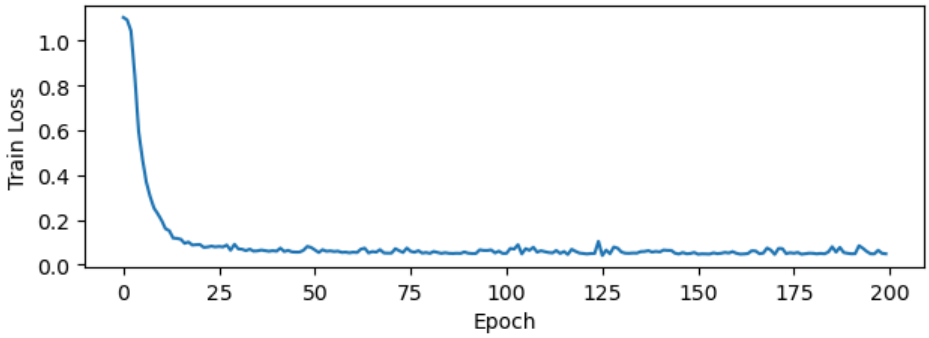


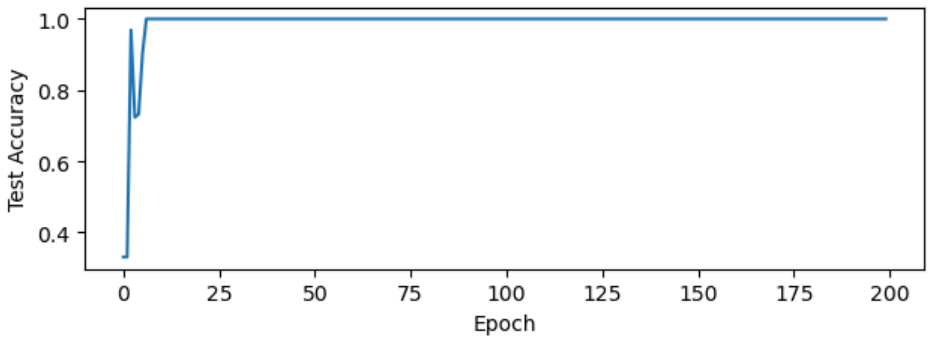
(4) Train the new Residual DNN model with increasing number of layers (2, 3, 4, 5, 10, ..., 100). Report and discuss your observations. Why do residual connections help train deep neural network?

DNN using residual connection can obtain more accurate validation results faster.
Residual connections allow gradient connections to propagate directly to earlier layers, thereby maintaining the strength of the gradient and helping the network learn. By adding the input directly to the output, each layer of the network can effectively receive the gradient signal and keep updating even in very deep networks.
By introducing residual connections, the network can reduce the possibility of overfitting while maintaining complexity.

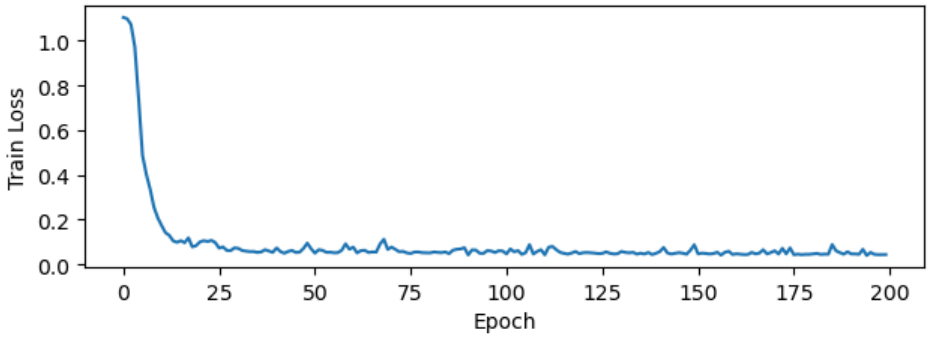
In [10]:
```
# Change number of layers into 2\3\4\5\10\...100
for numlayers in range(2, 101):
    if numlayers <= 5 or numlayers % 10 == 0:
        model = Residual_DNN(input_dim=4, num_classes=3, hidden_dim=30, num_layers=numlayers)
        cross_entropy = nn.CrossEntropyLoss()
        opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)
        accuracy = lambda y_hat, y: (y_hat.argmax(dim=1) == y).float().mean()

        print(f"Start training using Residual DNN with {numlayers} epochs")
        all_losses, all_accuracies = [], []
        for epoch in range(num_epochs):
            # Train for one epoch
            losses = []
            for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_train):
                # Use model to compute predictions
                yhat = model(X_batch)
                l = cross_entropy(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`

                # Compute gradients by back propagation
                l.backward()

                # Update parameters using their gradient
                opt.step()
                opt.zero_grad()
                losses.append(l.detach().item())

            # Measure accuracy on the test set
            acc = []
            for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
                yhat = model(X_batch)
                acc.append(accuracy(yhat, y_batch))

            all_losses.append(np.mean(losses))
            all_accuracies.append(np.mean(acc))

            if (epoch+1) % 10 == 0:
                # print(yhat)
                print(f"Epoch {epoch+1}: Train Loss {np.mean(losses):.3f} Test Accuracy {np.mean(acc):.3f}", flus

        # Evaluation
        print(f"\nStart evaluation using Residual DNN with {numlayers} epochs")
        with torch.no_grad():
            yhat, y = [], []
            for X_batch, y_batch in data_iter(batch_size=16, db=db_test):
                yhat.append(model(X_batch))
                y.append(y_batch)

        yhat = torch.cat(yhat, dim=0).argmax(dim=1)
        y = torch.cat(y, dim=0)
        cm = to1hot(y).T@to1hot(yhat)
        print('CM = \n', cm.numpy())

        plt.subplot(2, 1, 1)
        plt.plot(all_losses)
        plt.ylabel('Train Loss')
        plt.xlabel('Epoch')
        plt.subplot(2, 1, 2)
        plt.plot(all_accuracies)
        plt.ylabel('Test Accuracy')
        plt.xlabel('Epoch')
        plt.tight_layout()
        plt.show()
```

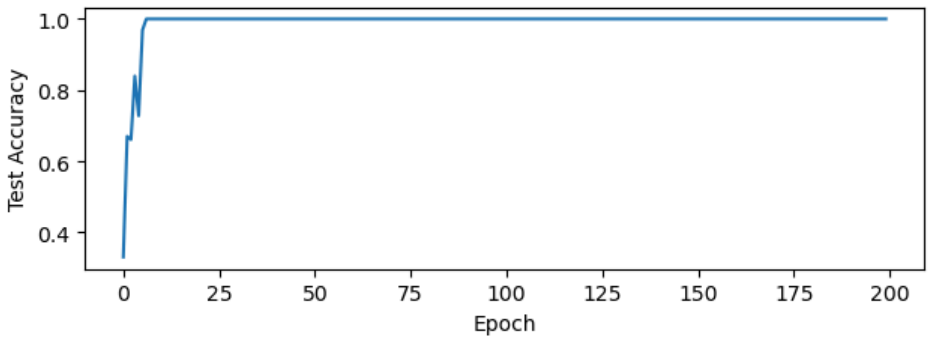
```
Start training using Residual DNN with 2 epochs
Epoch 10: Train Loss 0.227 Test Accuracy 1.000
Epoch 20: Train Loss 0.089 Test Accuracy 1.000
Epoch 30: Train Loss 0.091 Test Accuracy 1.000
Epoch 40: Train Loss 0.063 Test Accuracy 1.000
Epoch 50: Train Loss 0.077 Test Accuracy 1.000
Epoch 60: Train Loss 0.054 Test Accuracy 1.000
Epoch 70: Train Loss 0.052 Test Accuracy 1.000
Epoch 80: Train Loss 0.055 Test Accuracy 1.000
Epoch 90: Train Loss 0.057 Test Accuracy 1.000
Epoch 100: Train Loss 0.051 Test Accuracy 1.000
Epoch 110: Train Loss 0.063 Test Accuracy 1.000
Epoch 120: Train Loss 0.052 Test Accuracy 1.000
Epoch 130: Train Loss 0.074 Test Accuracy 1.000
Epoch 140: Train Loss 0.060 Test Accuracy 1.000
Epoch 150: Train Loss 0.055 Test Accuracy 1.000
Epoch 160: Train Loss 0.059 Test Accuracy 1.000
Epoch 170: Train Loss 0.065 Test Accuracy 1.000
Epoch 180: Train Loss 0.052 Test Accuracy 1.000
Epoch 190: Train Loss 0.051 Test Accuracy 1.000
Epoch 200: Train Loss 0.049 Test Accuracy 1.000

Start evaluation using Residual DNN with 2 epochs
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



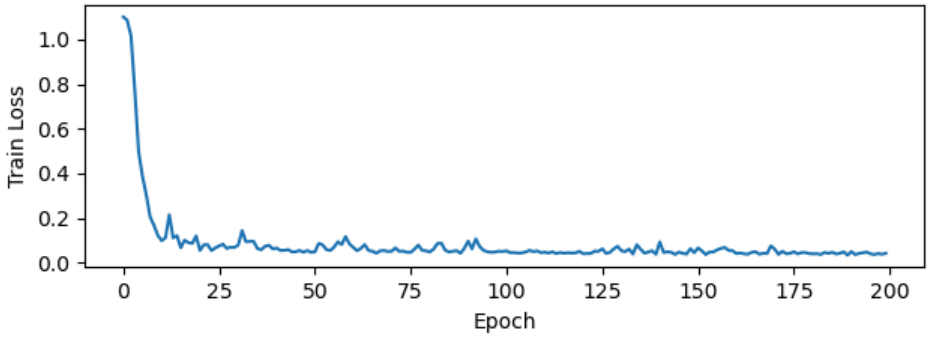


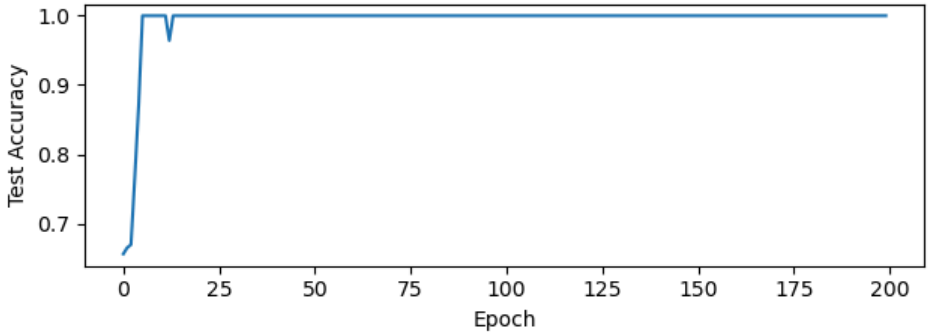
```
Start training using Residual DNN with 3 epochs
Epoch 10: Train Loss 0.207 Test Accuracy 1.000
Epoch 20: Train Loss 0.083 Test Accuracy 1.000
Epoch 30: Train Loss 0.073 Test Accuracy 1.000
Epoch 40: Train Loss 0.053 Test Accuracy 1.000
Epoch 50: Train Loss 0.069 Test Accuracy 1.000
Epoch 60: Train Loss 0.067 Test Accuracy 1.000
Epoch 70: Train Loss 0.066 Test Accuracy 1.000
Epoch 80: Train Loss 0.051 Test Accuracy 1.000
Epoch 90: Train Loss 0.074 Test Accuracy 1.000
Epoch 100: Train Loss 0.059 Test Accuracy 1.000
Epoch 110: Train Loss 0.065 Test Accuracy 1.000
Epoch 120: Train Loss 0.047 Test Accuracy 1.000
Epoch 130: Train Loss 0.047 Test Accuracy 1.000
Epoch 140: Train Loss 0.049 Test Accuracy 1.000
Epoch 150: Train Loss 0.087 Test Accuracy 1.000
Epoch 160: Train Loss 0.043 Test Accuracy 1.000
Epoch 170: Train Loss 0.052 Test Accuracy 1.000
Epoch 180: Train Loss 0.044 Test Accuracy 1.000
Epoch 190: Train Loss 0.055 Test Accuracy 1.000
Epoch 200: Train Loss 0.043 Test Accuracy 1.000

Start evaluation using Residual DNN with 3 epochs
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



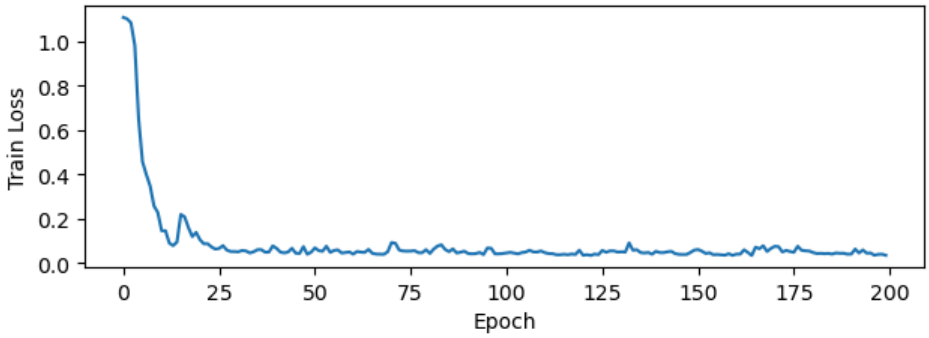


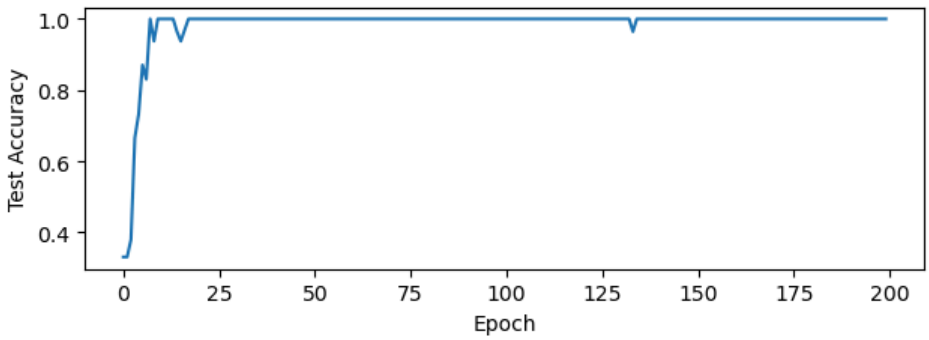
```
Start training using Residual DNN with 4 epochs
Epoch 10: Train Loss 0.123 Test Accuracy 1.000
Epoch 20: Train Loss 0.120 Test Accuracy 1.000
Epoch 30: Train Loss 0.068 Test Accuracy 1.000
Epoch 40: Train Loss 0.062 Test Accuracy 1.000
Epoch 50: Train Loss 0.047 Test Accuracy 1.000
Epoch 60: Train Loss 0.084 Test Accuracy 1.000
Epoch 70: Train Loss 0.051 Test Accuracy 1.000
Epoch 80: Train Loss 0.053 Test Accuracy 1.000
Epoch 90: Train Loss 0.065 Test Accuracy 1.000
Epoch 100: Train Loss 0.051 Test Accuracy 1.000
Epoch 110: Train Loss 0.045 Test Accuracy 1.000
Epoch 120: Train Loss 0.052 Test Accuracy 1.000
Epoch 130: Train Loss 0.074 Test Accuracy 1.000
Epoch 140: Train Loss 0.038 Test Accuracy 1.000
Epoch 150: Train Loss 0.045 Test Accuracy 1.000
Epoch 160: Train Loss 0.054 Test Accuracy 1.000
Epoch 170: Train Loss 0.076 Test Accuracy 1.000
Epoch 180: Train Loss 0.041 Test Accuracy 1.000
Epoch 190: Train Loss 0.034 Test Accuracy 1.000
Epoch 200: Train Loss 0.042 Test Accuracy 1.000

Start evaluation using Residual DNN with 4 epochs
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



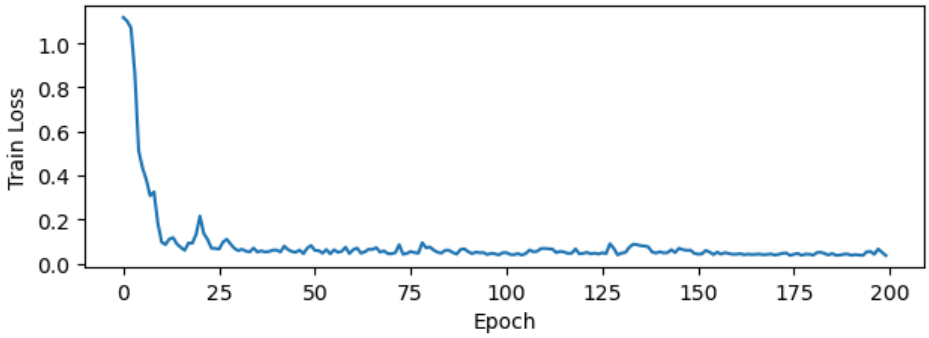


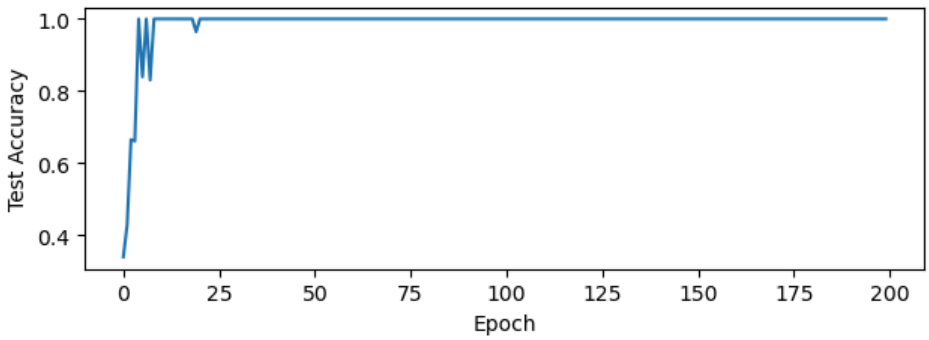
```
Start training using Residual DNN with 5 epochs
Epoch 10: Train Loss 0.229 Test Accuracy 1.000
Epoch 20: Train Loss 0.139 Test Accuracy 1.000
Epoch 30: Train Loss 0.053 Test Accuracy 1.000
Epoch 40: Train Loss 0.078 Test Accuracy 1.000
Epoch 50: Train Loss 0.050 Test Accuracy 1.000
Epoch 60: Train Loss 0.051 Test Accuracy 1.000
Epoch 70: Train Loss 0.051 Test Accuracy 1.000
Epoch 80: Train Loss 0.061 Test Accuracy 1.000
Epoch 90: Train Loss 0.054 Test Accuracy 1.000
Epoch 100: Train Loss 0.044 Test Accuracy 1.000
Epoch 110: Train Loss 0.055 Test Accuracy 1.000
Epoch 120: Train Loss 0.059 Test Accuracy 1.000
Epoch 130: Train Loss 0.050 Test Accuracy 1.000
Epoch 140: Train Loss 0.054 Test Accuracy 1.000
Epoch 150: Train Loss 0.059 Test Accuracy 1.000
Epoch 160: Train Loss 0.035 Test Accuracy 1.000
Epoch 170: Train Loss 0.066 Test Accuracy 1.000
Epoch 180: Train Loss 0.055 Test Accuracy 1.000
Epoch 190: Train Loss 0.041 Test Accuracy 1.000
Epoch 200: Train Loss 0.036 Test Accuracy 1.000

Start evaluation using Residual DNN with 5 epochs
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



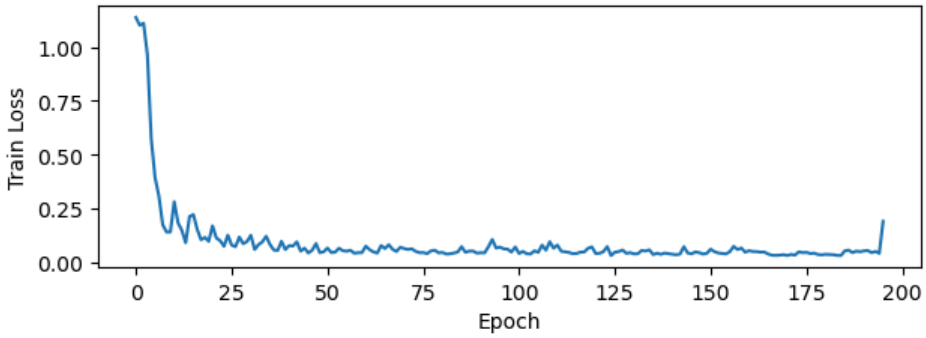


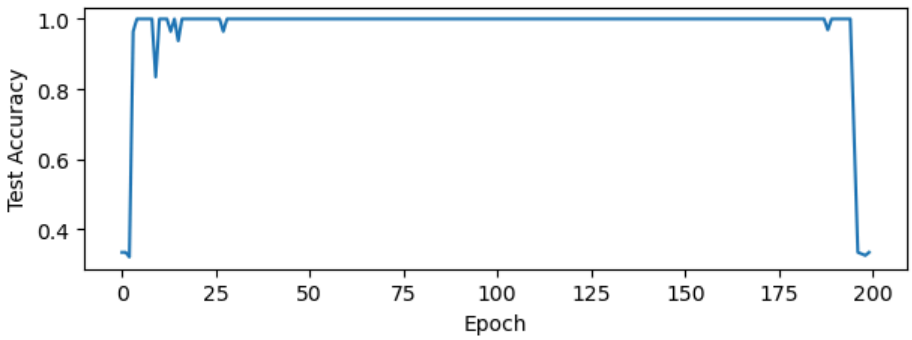
```
Start training using Residual DNN with 10 epochs
Epoch 10: Train Loss 0.184 Test Accuracy 1.000
Epoch 20: Train Loss 0.132 Test Accuracy 0.964
Epoch 30: Train Loss 0.068 Test Accuracy 1.000
Epoch 40: Train Loss 0.061 Test Accuracy 1.000
Epoch 50: Train Loss 0.081 Test Accuracy 1.000
Epoch 60: Train Loss 0.045 Test Accuracy 1.000
Epoch 70: Train Loss 0.045 Test Accuracy 1.000
Epoch 80: Train Loss 0.072 Test Accuracy 1.000
Epoch 90: Train Loss 0.067 Test Accuracy 1.000
Epoch 100: Train Loss 0.049 Test Accuracy 1.000
Epoch 110: Train Loss 0.068 Test Accuracy 1.000
Epoch 120: Train Loss 0.044 Test Accuracy 1.000
Epoch 130: Train Loss 0.039 Test Accuracy 1.000
Epoch 140: Train Loss 0.048 Test Accuracy 1.000
Epoch 150: Train Loss 0.046 Test Accuracy 1.000
Epoch 160: Train Loss 0.042 Test Accuracy 1.000
Epoch 170: Train Loss 0.043 Test Accuracy 1.000
Epoch 180: Train Loss 0.042 Test Accuracy 1.000
Epoch 190: Train Loss 0.043 Test Accuracy 1.000
Epoch 200: Train Loss 0.036 Test Accuracy 1.000

Start evaluation using Residual DNN with 10 epochs
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



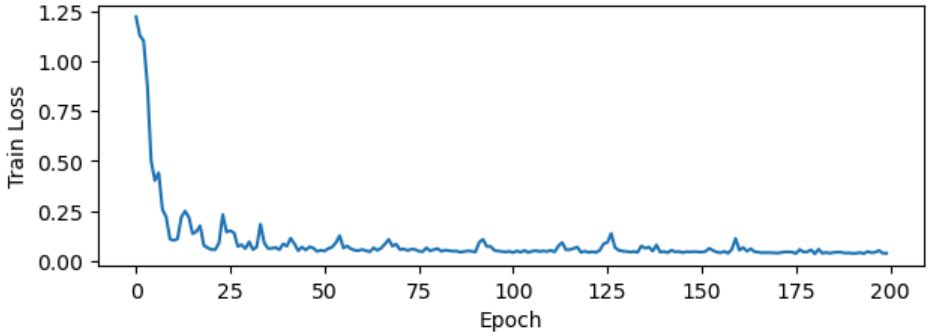


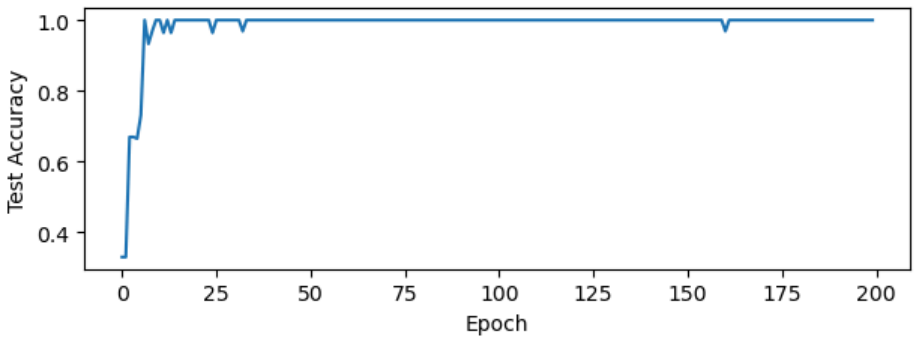
```
Start training using Residual DNN with 20 epochs
Epoch 10: Train Loss 0.141 Test Accuracy 0.835
Epoch 20: Train Loss 0.097 Test Accuracy 1.000
Epoch 30: Train Loss 0.096 Test Accuracy 1.000
Epoch 40: Train Loss 0.059 Test Accuracy 1.000
Epoch 50: Train Loss 0.049 Test Accuracy 1.000
Epoch 60: Train Loss 0.045 Test Accuracy 1.000
Epoch 70: Train Loss 0.070 Test Accuracy 1.000
Epoch 80: Train Loss 0.044 Test Accuracy 1.000
Epoch 90: Train Loss 0.042 Test Accuracy 1.000
Epoch 100: Train Loss 0.071 Test Accuracy 1.000
Epoch 110: Train Loss 0.065 Test Accuracy 1.000
Epoch 120: Train Loss 0.071 Test Accuracy 1.000
Epoch 130: Train Loss 0.045 Test Accuracy 1.000
Epoch 140: Train Loss 0.040 Test Accuracy 1.000
Epoch 150: Train Loss 0.042 Test Accuracy 1.000
Epoch 160: Train Loss 0.047 Test Accuracy 1.000
Epoch 170: Train Loss 0.035 Test Accuracy 1.000
Epoch 180: Train Loss 0.034 Test Accuracy 1.000
Epoch 190: Train Loss 0.049 Test Accuracy 1.000
Epoch 200: Train Loss nan Test Accuracy 0.335

Start evaluation using Residual DNN with 20 epochs
CM =
 [[10.  0.  0.]
 [10.  0.  0.]
 [10.  0.  0.]]
```



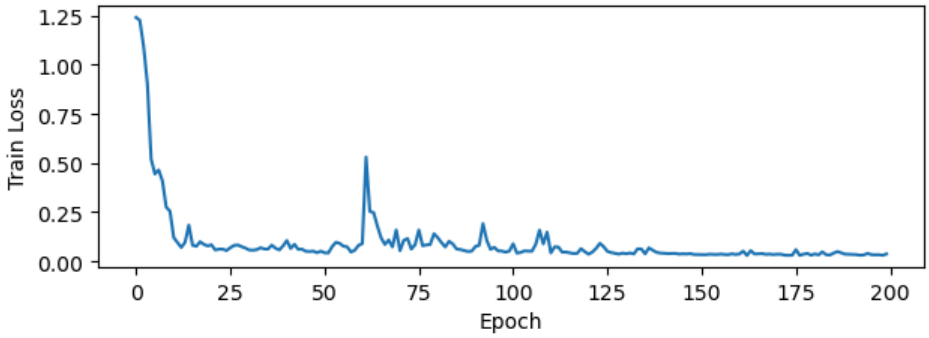


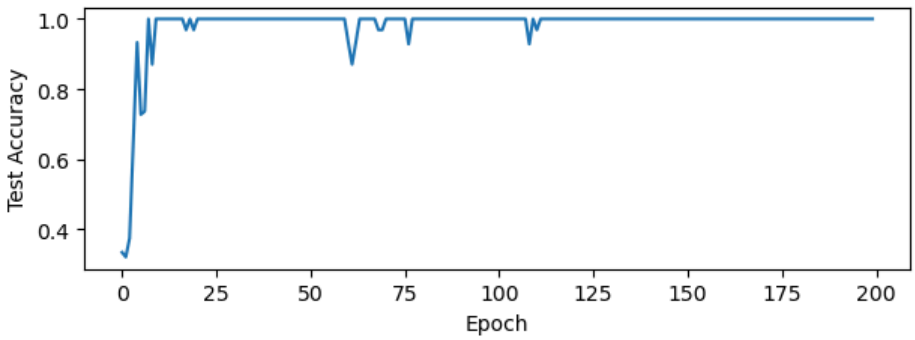
```
Start training using Residual DNN with 30 epochs
Epoch 10: Train Loss 0.108 Test Accuracy 1.000
Epoch 20: Train Loss 0.065 Test Accuracy 1.000
Epoch 30: Train Loss 0.061 Test Accuracy 1.000
Epoch 40: Train Loss 0.084 Test Accuracy 1.000
Epoch 50: Train Loss 0.052 Test Accuracy 1.000
Epoch 60: Train Loss 0.051 Test Accuracy 1.000
Epoch 70: Train Loss 0.083 Test Accuracy 1.000
Epoch 80: Train Loss 0.056 Test Accuracy 1.000
Epoch 90: Train Loss 0.047 Test Accuracy 1.000
Epoch 100: Train Loss 0.047 Test Accuracy 1.000
Epoch 110: Train Loss 0.046 Test Accuracy 1.000
Epoch 120: Train Loss 0.048 Test Accuracy 1.000
Epoch 130: Train Loss 0.048 Test Accuracy 1.000
Epoch 140: Train Loss 0.043 Test Accuracy 1.000
Epoch 150: Train Loss 0.044 Test Accuracy 1.000
Epoch 160: Train Loss 0.111 Test Accuracy 1.000
Epoch 170: Train Loss 0.040 Test Accuracy 1.000
Epoch 180: Train Loss 0.054 Test Accuracy 1.000
Epoch 190: Train Loss 0.039 Test Accuracy 1.000
Epoch 200: Train Loss 0.037 Test Accuracy 1.000

Start evaluation using Residual DNN with 30 epochs
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



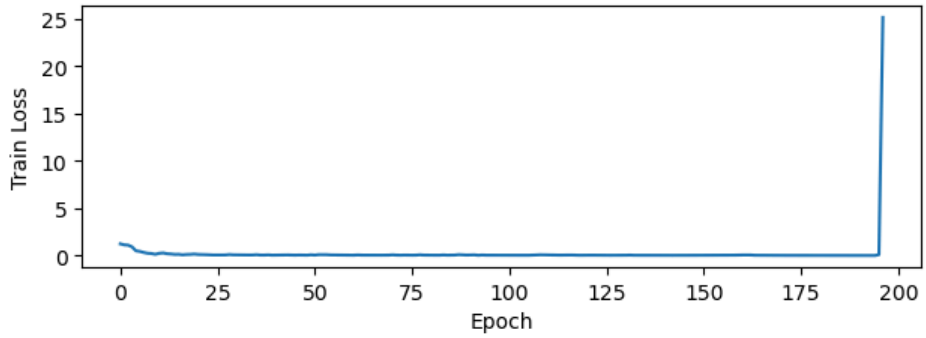


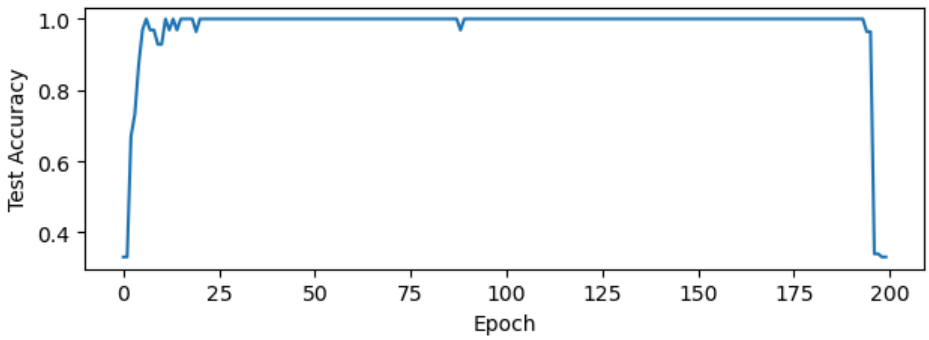
```
Start training using Residual DNN with 40 epochs
Epoch 10: Train Loss 0.255 Test Accuracy 1.000
Epoch 20: Train Loss 0.078 Test Accuracy 0.969
Epoch 30: Train Loss 0.067 Test Accuracy 1.000
Epoch 40: Train Loss 0.079 Test Accuracy 1.000
Epoch 50: Train Loss 0.052 Test Accuracy 1.000
Epoch 60: Train Loss 0.081 Test Accuracy 1.000
Epoch 70: Train Loss 0.160 Test Accuracy 0.969
Epoch 80: Train Loss 0.140 Test Accuracy 1.000
Epoch 90: Train Loss 0.052 Test Accuracy 1.000
Epoch 100: Train Loss 0.052 Test Accuracy 1.000
Epoch 110: Train Loss 0.149 Test Accuracy 1.000
Epoch 120: Train Loss 0.049 Test Accuracy 1.000
Epoch 130: Train Loss 0.041 Test Accuracy 1.000
Epoch 140: Train Loss 0.042 Test Accuracy 1.000
Epoch 150: Train Loss 0.034 Test Accuracy 1.000
Epoch 160: Train Loss 0.035 Test Accuracy 1.000
Epoch 170: Train Loss 0.034 Test Accuracy 1.000
Epoch 180: Train Loss 0.031 Test Accuracy 1.000
Epoch 190: Train Loss 0.036 Test Accuracy 1.000
Epoch 200: Train Loss 0.037 Test Accuracy 1.000

Start evaluation using Residual DNN with 40 epochs
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



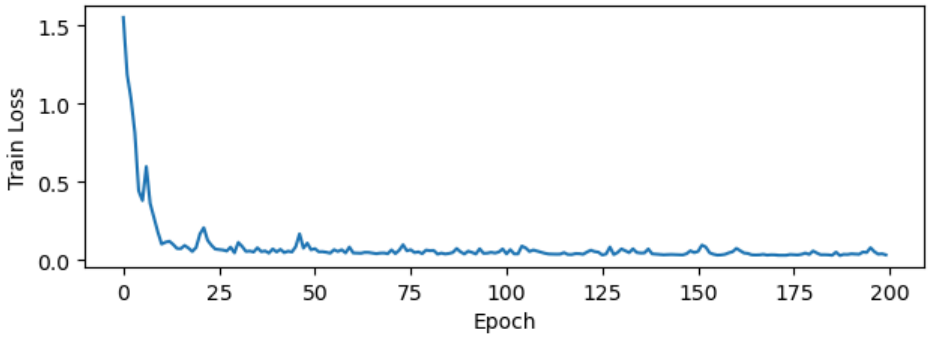


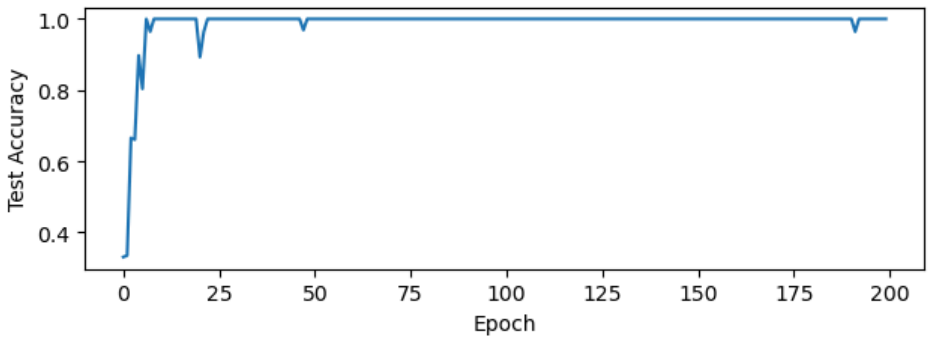
```
Start training using Residual DNN with 50 epochs
Epoch 10: Train Loss 0.133 Test Accuracy 0.929
Epoch 20: Train Loss 0.158 Test Accuracy 0.964
Epoch 30: Train Loss 0.079 Test Accuracy 1.000
Epoch 40: Train Loss 0.043 Test Accuracy 1.000
Epoch 50: Train Loss 0.081 Test Accuracy 1.000
Epoch 60: Train Loss 0.048 Test Accuracy 1.000
Epoch 70: Train Loss 0.047 Test Accuracy 1.000
Epoch 80: Train Loss 0.040 Test Accuracy 1.000
Epoch 90: Train Loss 0.055 Test Accuracy 1.000
Epoch 100: Train Loss 0.043 Test Accuracy 1.000
Epoch 110: Train Loss 0.073 Test Accuracy 1.000
Epoch 120: Train Loss 0.030 Test Accuracy 1.000
Epoch 130: Train Loss 0.028 Test Accuracy 1.000
Epoch 140: Train Loss 0.024 Test Accuracy 1.000
Epoch 150: Train Loss 0.027 Test Accuracy 1.000
Epoch 160: Train Loss 0.057 Test Accuracy 1.000
Epoch 170: Train Loss 0.021 Test Accuracy 1.000
Epoch 180: Train Loss 0.017 Test Accuracy 1.000
Epoch 190: Train Loss 0.010 Test Accuracy 1.000
Epoch 200: Train Loss nan Test Accuracy 0.330

Start evaluation using Residual DNN with 50 epochs
CM =
 [[10.  0.  0.]
 [10.  0.  0.]
 [10.  0.  0.]]
```



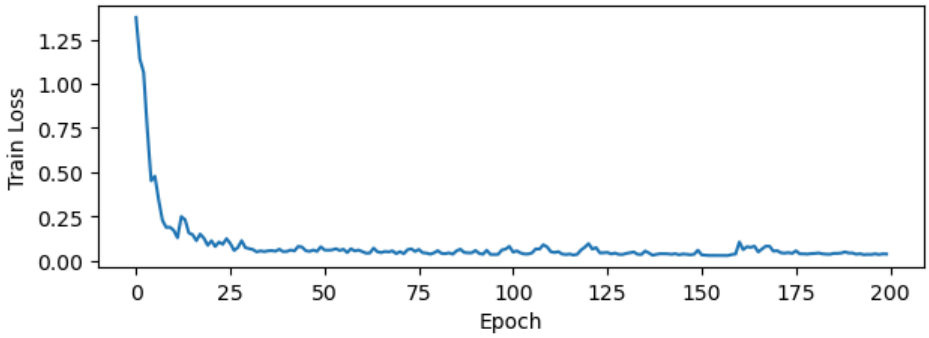


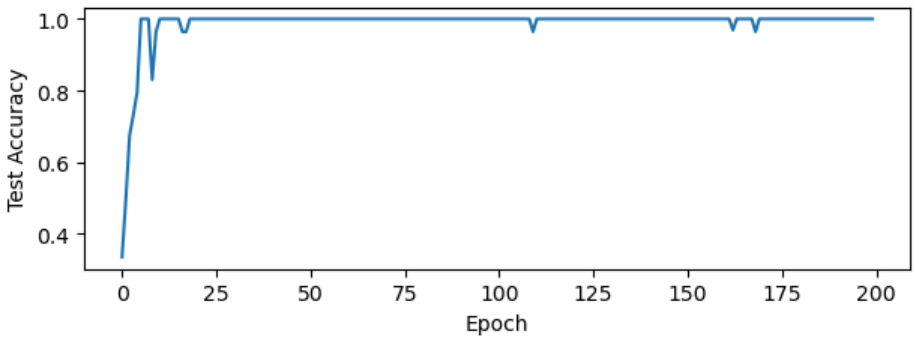
```
Start training using Residual DNN with 60 epochs
Epoch 10: Train Loss 0.178 Test Accuracy 1.000
Epoch 20: Train Loss 0.078 Test Accuracy 1.000
Epoch 30: Train Loss 0.044 Test Accuracy 1.000
Epoch 40: Train Loss 0.069 Test Accuracy 1.000
Epoch 50: Train Loss 0.064 Test Accuracy 1.000
Epoch 60: Train Loss 0.081 Test Accuracy 1.000
Epoch 70: Train Loss 0.039 Test Accuracy 1.000
Epoch 80: Train Loss 0.061 Test Accuracy 1.000
Epoch 90: Train Loss 0.037 Test Accuracy 1.000
Epoch 100: Train Loss 0.069 Test Accuracy 1.000
Epoch 110: Train Loss 0.047 Test Accuracy 1.000
Epoch 120: Train Loss 0.039 Test Accuracy 1.000
Epoch 130: Train Loss 0.046 Test Accuracy 1.000
Epoch 140: Train Loss 0.035 Test Accuracy 1.000
Epoch 150: Train Loss 0.046 Test Accuracy 1.000
Epoch 160: Train Loss 0.051 Test Accuracy 1.000
Epoch 170: Train Loss 0.031 Test Accuracy 1.000
Epoch 180: Train Loss 0.034 Test Accuracy 1.000
Epoch 190: Train Loss 0.033 Test Accuracy 1.000
Epoch 200: Train Loss 0.031 Test Accuracy 1.000

Start evaluation using Residual DNN with 60 epochs
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



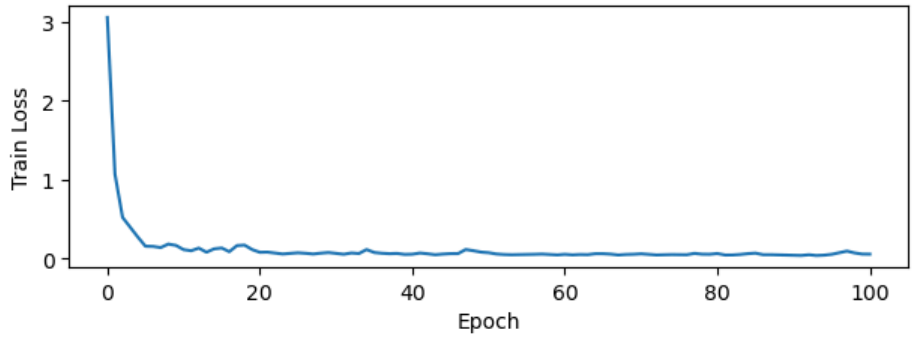


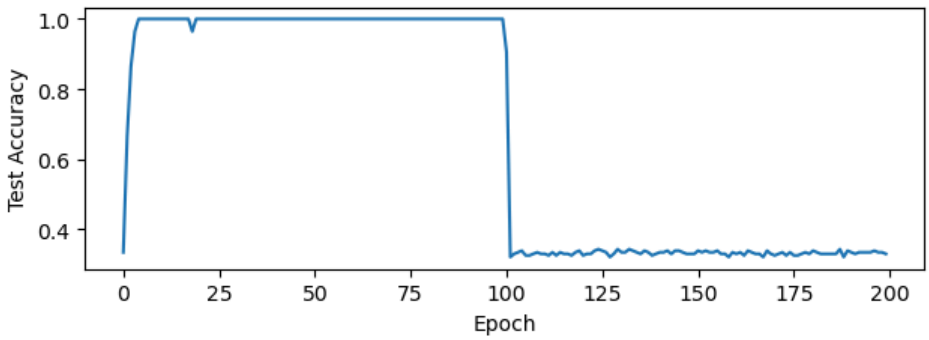
```
Start training using Residual DNN with 70 epochs
Epoch 10: Train Loss 0.188 Test Accuracy 0.964
Epoch 20: Train Loss 0.086 Test Accuracy 1.000
Epoch 30: Train Loss 0.072 Test Accuracy 1.000
Epoch 40: Train Loss 0.050 Test Accuracy 1.000
Epoch 50: Train Loss 0.077 Test Accuracy 1.000
Epoch 60: Train Loss 0.059 Test Accuracy 1.000
Epoch 70: Train Loss 0.039 Test Accuracy 1.000
Epoch 80: Train Loss 0.043 Test Accuracy 1.000
Epoch 90: Train Loss 0.043 Test Accuracy 1.000
Epoch 100: Train Loss 0.080 Test Accuracy 1.000
Epoch 110: Train Loss 0.077 Test Accuracy 0.964
Epoch 120: Train Loss 0.075 Test Accuracy 1.000
Epoch 130: Train Loss 0.034 Test Accuracy 1.000
Epoch 140: Train Loss 0.038 Test Accuracy 1.000
Epoch 150: Train Loss 0.058 Test Accuracy 1.000
Epoch 160: Train Loss 0.036 Test Accuracy 1.000
Epoch 170: Train Loss 0.053 Test Accuracy 1.000
Epoch 180: Train Loss 0.039 Test Accuracy 1.000
Epoch 190: Train Loss 0.042 Test Accuracy 1.000
Epoch 200: Train Loss 0.036 Test Accuracy 1.000

Start evaluation using Residual DNN with 70 epochs
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```



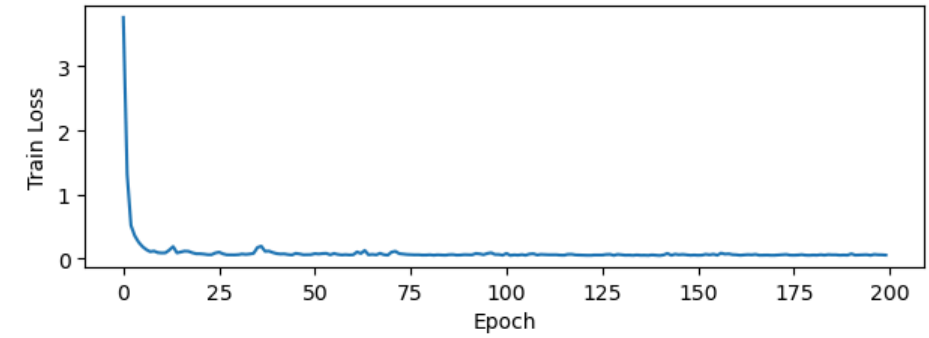


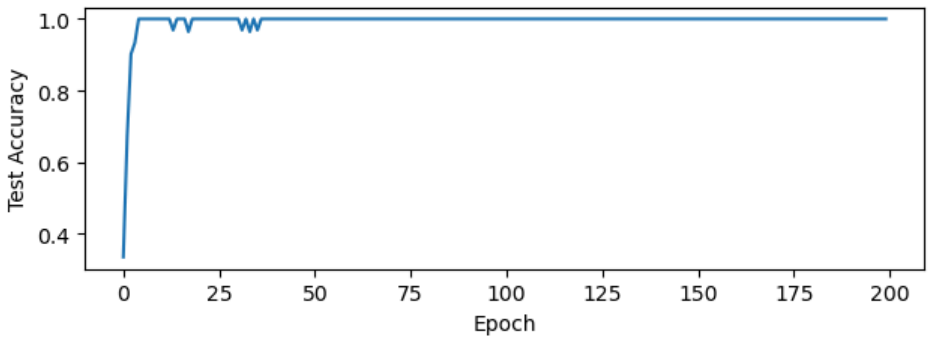
```
Start training using Residual DNN with 80 epochs
Epoch 10: Train Loss 0.166 Test Accuracy 1.000
Epoch 20: Train Loss 0.112 Test Accuracy 1.000
Epoch 30: Train Loss 0.075 Test Accuracy 1.000
Epoch 40: Train Loss 0.051 Test Accuracy 1.000
Epoch 50: Train Loss 0.081 Test Accuracy 1.000
Epoch 60: Train Loss 0.045 Test Accuracy 1.000
Epoch 70: Train Loss 0.053 Test Accuracy 1.000
Epoch 80: Train Loss 0.053 Test Accuracy 1.000
Epoch 90: Train Loss 0.041 Test Accuracy 1.000
Epoch 100: Train Loss 0.056 Test Accuracy 1.000
Epoch 110: Train Loss nan Test Accuracy 0.330
Epoch 120: Train Loss nan Test Accuracy 0.339
Epoch 130: Train Loss nan Test Accuracy 0.344
Epoch 140: Train Loss nan Test Accuracy 0.330
Epoch 150: Train Loss nan Test Accuracy 0.330
Epoch 160: Train Loss nan Test Accuracy 0.335
Epoch 170: Train Loss nan Test Accuracy 0.330
Epoch 180: Train Loss nan Test Accuracy 0.330
Epoch 190: Train Loss nan Test Accuracy 0.339
Epoch 200: Train Loss nan Test Accuracy 0.330

Start evaluation using Residual DNN with 80 epochs
CM =
 [[10.  0.  0.]
 [10.  0.  0.]
 [10.  0.  0.]]
```



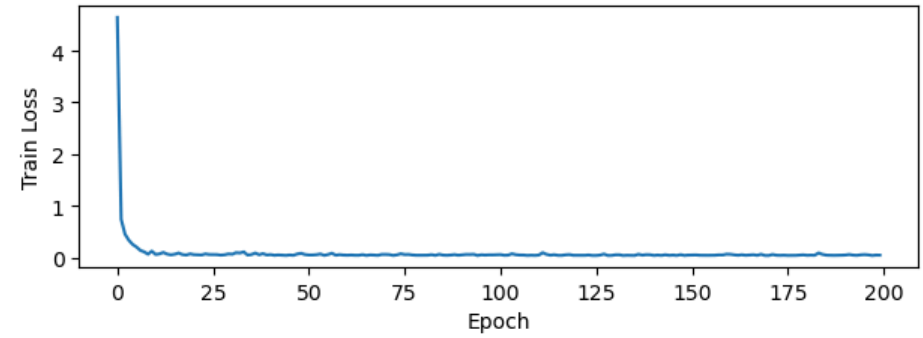


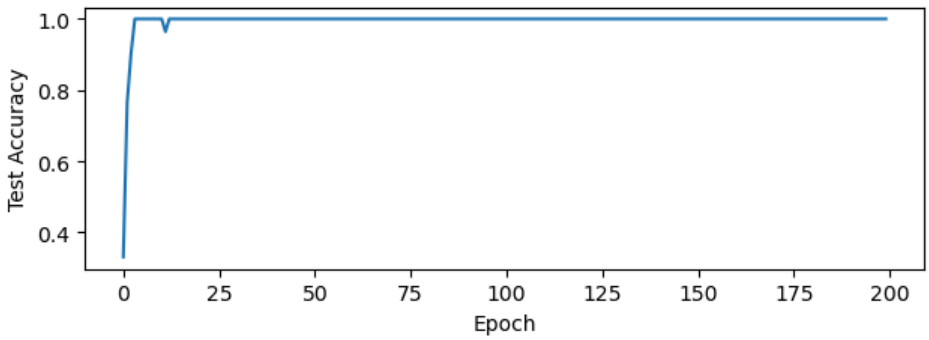
```
Start training using Residual DNN with 90 epochs
Epoch 10: Train Loss 0.090 Test Accuracy 1.000
Epoch 20: Train Loss 0.073 Test Accuracy 1.000
Epoch 30: Train Loss 0.055 Test Accuracy 1.000
Epoch 40: Train Loss 0.092 Test Accuracy 1.000
Epoch 50: Train Loss 0.059 Test Accuracy 1.000
Epoch 60: Train Loss 0.054 Test Accuracy 1.000
Epoch 70: Train Loss 0.050 Test Accuracy 1.000
Epoch 80: Train Loss 0.052 Test Accuracy 1.000
Epoch 90: Train Loss 0.058 Test Accuracy 1.000
Epoch 100: Train Loss 0.050 Test Accuracy 1.000
Epoch 110: Train Loss 0.062 Test Accuracy 1.000
Epoch 120: Train Loss 0.051 Test Accuracy 1.000
Epoch 130: Train Loss 0.062 Test Accuracy 1.000
Epoch 140: Train Loss 0.049 Test Accuracy 1.000
Epoch 150: Train Loss 0.052 Test Accuracy 1.000
Epoch 160: Train Loss 0.059 Test Accuracy 1.000
Epoch 170: Train Loss 0.049 Test Accuracy 1.000
Epoch 180: Train Loss 0.048 Test Accuracy 1.000
Epoch 190: Train Loss 0.050 Test Accuracy 1.000
Epoch 200: Train Loss 0.051 Test Accuracy 1.000

Start evaluation using Residual DNN with 90 epochs
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```





```
Start training using Residual DNN with 100 epochs
Epoch 10: Train Loss 0.132 Test Accuracy 1.000
Epoch 20: Train Loss 0.079 Test Accuracy 1.000
Epoch 30: Train Loss 0.079 Test Accuracy 1.000
Epoch 40: Train Loss 0.058 Test Accuracy 1.000
Epoch 50: Train Loss 0.065 Test Accuracy 1.000
Epoch 60: Train Loss 0.056 Test Accuracy 1.000
Epoch 70: Train Loss 0.068 Test Accuracy 1.000
Epoch 80: Train Loss 0.053 Test Accuracy 1.000
Epoch 90: Train Loss 0.053 Test Accuracy 1.000
Epoch 100: Train Loss 0.063 Test Accuracy 1.000
Epoch 110: Train Loss 0.050 Test Accuracy 1.000
Epoch 120: Train Loss 0.052 Test Accuracy 1.000
Epoch 130: Train Loss 0.049 Test Accuracy 1.000
Epoch 140: Train Loss 0.062 Test Accuracy 1.000
Epoch 150: Train Loss 0.055 Test Accuracy 1.000
Epoch 160: Train Loss 0.073 Test Accuracy 1.000
Epoch 170: Train Loss 0.048 Test Accuracy 1.000
Epoch 180: Train Loss 0.060 Test Accuracy 1.000
Epoch 190: Train Loss 0.054 Test Accuracy 1.000
Epoch 200: Train Loss 0.054 Test Accuracy 1.000

Start evaluation using Residual DNN with 100 epochs
CM =
 [[10.  0.  0.]
 [ 0. 10.  0.]
 [ 0.  0. 10.]]
```





Problem 3. Develop a MLP for wine quality problem. Compare the performance (# iterations of training to convergence, final loss value and accuracy (on testing dataset))

In [75]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt

# makes printing more human-friendly
np.set_printoptions(precision=3, suppress=True)
```

In [76]:
```
def to1hot(labels):
    """Converts an array of class labels into their 1hot encodings"""
    labels = torch.tensor(labels, dtype=torch.long)
    return torch.eye(7)[labels.long()]
```

Data: `winequality-white-1.csv`

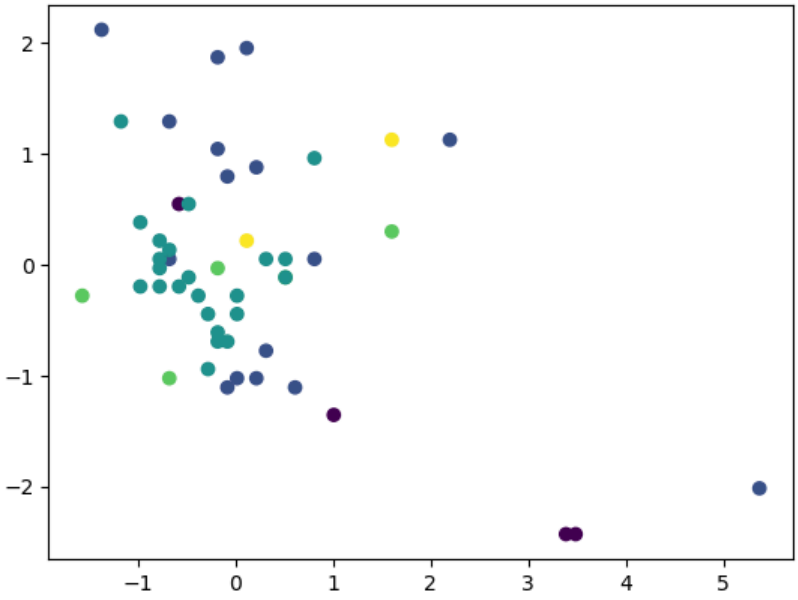
In [77]:
```
class Dataset:
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

    def __len__(self):
        return len(self.X)

# Import data
data = pd.read_csv('winequality-white-1.csv')

X, y = data.drop(columns=['quality']).values, data['quality'].values
# print(X.shape, y.shape)
y = y - 3  # Shift labels to be in range [0, 6]

# Feature standardization (Normalization)
scaler = StandardScaler()
X_normalized = scaler.fit_transform(X)

# Split data (60/20/20)
X_train, X_temp, y_train, y_temp = train_test_split(X_normalized, y, test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

db_train = Dataset(X_train, y_train)
db_test = Dataset(X_test, y_test)
db_val = Dataset(X_val, y_val)
print(len(db_train), db_train[0])
print(len(db_test), db_test[0])
print(len(db_val), db_val[0])
```

```
2938 (array([ 1.713, -0.281, -0.035, -0.767,  0.169, -1.371, -1.562, -0.765,
       -1.975, -0.086, -0.824]), 2.0)
980 (array([-1.013, -0.281,  0.544, -0.136,  0.169,  1.217,  0.321, -0.765,
       -0.187,  3.419, -0.012]), 3.0)
980 (array([-2.672,  1.701, -2.762, -0.984,  0.169, -1.429,  0.933, -0.765,
        3.985,  0.527, -0.255]), 2.0)
```

In [78]:
```
import random

# Reading the dataset
def data_iter(batch_size, db):
    num_examples = len(db)

    # The examples are read at random, in no particular order
    indices = list(range(num_examples))
```

```
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        X, Y = [], []
        for j in indices[i:i + batch_size]:
            x, lbl = db[j]

            # Process image
            x = torch.from_numpy(x).float()
            lbl = torch.tensor(lbl).long()

            X.append(x), Y.append(lbl)
        yield torch.stack(X), torch.stack(Y)

# Check data reader
for X_batch, y_batch in data_iter(batch_size=50, db=db_train):
    print('X_batch', X_batch.shape)
    print('y_batch', y_batch.shape)
    plt.scatter(X_batch[:, 1].numpy(), X_batch[:, 2].numpy(), c=y_batch.numpy())
    break
```

```
X_batch torch.Size([50, 11])
y_batch torch.Size([50])
```



Model: 5-layer MLP with 11 inputs, 8 output and 3 hidden layers with H hidden neurons in each

In [92]:

```
class DNN(nn.Module):
    def __init__(self, input_dim, hidden_dim=64, num_classes=7, num_layers=5):
        super(DNN, self).__init__()

        # Define layers
        layers = [nn.Linear(input_dim, hidden_dim)]
        for i in range(1, num_layers-1):
            layers.append(nn.Linear(hidden_dim, hidden_dim))
        layers.append(nn.Linear(hidden_dim, num_classes))
        self.layers = nn.ModuleList(layers)
        self.relu = nn.ReLU()
        self.bn_layers = nn.ModuleList([nn.BatchNorm1d(hidden_dim) for _ in range(num_layers - 1)])


        # Initialize weights
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, mean=0, std=0.01)

    def forward(self, x):
        for i, layer in enumerate(self.layers):
            if i < len(self.layers) - 1:
                x = self.bn_layers[i](self.relu(layer(x)))  # Apply batch normalization
            else:
                x = layer(x)


# Define the MLP model
class MLP(nn.Module):
```

```
    def __init__(self, input_size, hidden_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, hidden_size)
        self.fc4 = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.fc4(x)
        # x = self.softmax(x)
        return x

# Initialize model, loss function, and optimizer
input_size = X_train.shape[1]
hidden_size = 64
output_size = 8

# Check Models
model = MLP(input_size, hidden_size, output_size)

for X_batch, y_batch in data_iter(batch_size=50, db=db_train):
    print('X_batch', X_batch.shape)
    out_batch = model(X_batch)
    print('out_batch', out_batch.shape)
    break
```

```
X_batch torch.Size([50, 11])
out_batch torch.Size([50, 8])
```

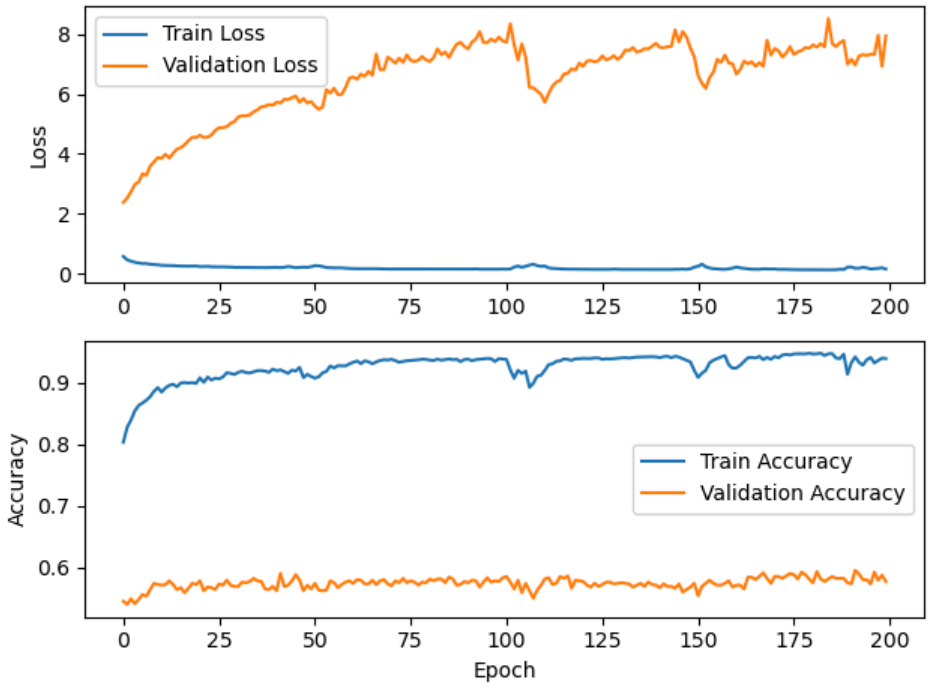
Training and Validation

In [ ]:

```
# Training
lr = 0.01
batch_size = 50
num_epochs = 200

criterion = nn.CrossEntropyLoss()
opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)
accuracy = lambda y_hat, y: (y_hat.argmax(dim=1) == y).float().mean()

print("Start training")
all_train_losses, all_val_losses = [], []
all_train_accuracies, all_val_accuracies = [], []

for epoch in range(num_epochs):
    # Training
    model.train()  # Set model to training mode
    train_losses = []
    for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_train):
        # Use model to compute predictions
        yhat = model(X_batch)
        l = criterion(yhat, y_batch)  # Minibatch loss in `X_batch` and `y_batch`

        # Compute gradients by back propagation
        l.backward()

        # Update parameters using their gradient
        opt.step()
        opt.zero_grad()
        train_losses.append(l.item())

# Validation
    model.eval()  # Set model to evaluation mode
    val_losses = []
    val_accuracies = []
    with torch.no_grad():  # Disable gradient computation during validation
        for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_val):
            yhat = model(X_batch)
            val_loss = criterion(yhat, y_batch)
            val_losses.append(val_loss.item())
            val_accuracies.append(accuracy(yhat, y_batch).item())

    # Log training and validation metrics
```

```
    all_train_losses.append(np.mean(train_losses))
    all_val_losses.append(np.mean(val_losses))
    all_train_accuracies.append(np.mean([accuracy(model(X_batch), y_batch).item()
                                         for X_batch, y_batch in data_iter(batch_size=batch_size, db=db_train)]))
    all_val_accuracies.append(np.mean(val_accuracies))

    # Print progress
    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch+1}: Train Loss {all_train_losses[-1]:.3f} Val Loss {all_val_losses[-1]:.3f} "
              f"Train Accuracy {all_train_accuracies[-1]:.3f} Val Accuracy {all_val_accuracies[-1]:.3f}")

# Plot training and validation metrics
plt.subplot(2, 1, 1)
plt.plot(all_train_losses, label='Train Loss')
plt.plot(all_val_losses, label='Validation Loss')
plt.ylabel('Loss')
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(all_train_accuracies, label='Train Accuracy')
plt.plot(all_val_accuracies, label='Validation Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.tight_layout()
plt.show()
```

```
Start training
Epoch 10: Train Loss 0.294 Val Loss 3.878 Train Accuracy 0.892 Val Accuracy 0.572
Epoch 20: Train Loss 0.252 Val Loss 4.551 Train Accuracy 0.899 Val Accuracy 0.572
Epoch 30: Train Loss 0.208 Val Loss 5.082 Train Accuracy 0.913 Val Accuracy 0.569
Epoch 40: Train Loss 0.199 Val Loss 5.631 Train Accuracy 0.922 Val Accuracy 0.569
Epoch 50: Train Loss 0.230 Val Loss 5.754 Train Accuracy 0.910 Val Accuracy 0.565
Epoch 60: Train Loss 0.171 Val Loss 6.537 Train Accuracy 0.931 Val Accuracy 0.578
Epoch 70: Train Loss 0.153 Val Loss 7.236 Train Accuracy 0.937 Val Accuracy 0.573
Epoch 80: Train Loss 0.155 Val Loss 7.142 Train Accuracy 0.938 Val Accuracy 0.575
Epoch 90: Train Loss 0.147 Val Loss 7.593 Train Accuracy 0.938 Val Accuracy 0.577
Epoch 100: Train Loss 0.150 Val Loss 7.784 Train Accuracy 0.939 Val Accuracy 0.583
Epoch 110: Train Loss 0.241 Val Loss 5.994 Train Accuracy 0.912 Val Accuracy 0.571
Epoch 120: Train Loss 0.145 Val Loss 7.037 Train Accuracy 0.939 Val Accuracy 0.578
Epoch 130: Train Loss 0.141 Val Loss 7.157 Train Accuracy 0.940 Val Accuracy 0.574
Epoch 140: Train Loss 0.140 Val Loss 7.697 Train Accuracy 0.942 Val Accuracy 0.571
Epoch 150: Train Loss 0.212 Val Loss 7.255 Train Accuracy 0.920 Val Accuracy 0.574
Epoch 160: Train Loss 0.180 Val Loss 7.016 Train Accuracy 0.924 Val Accuracy 0.568
Epoch 170: Train Loss 0.152 Val Loss 7.309 Train Accuracy 0.938 Val Accuracy 0.574
Epoch 180: Train Loss 0.130 Val Loss 7.534 Train Accuracy 0.948 Val Accuracy 0.581
Epoch 190: Train Loss 0.214 Val Loss 7.001 Train Accuracy 0.914 Val Accuracy 0.575
Epoch 200: Train Loss 0.151 Val Loss 7.948 Train Accuracy 0.939 Val Accuracy 0.577
```



In [ ]:

Problem 4

1. $L_{sq} = \frac{1}{2N}(\hat{y} - y)^T(\hat{y} - y)$

$$\frac{\partial L_{sq}}{\partial \hat{y}^{(i)}} = \frac{1}{N}(\hat{y}^{(i)} - y^{(i)})$$

$$\frac{\partial \hat{y}^{(i)}}{\partial u_2^{(i)}} = \sigma(u_2^{(i)})(1 - \sigma(u_2^{(i)}))$$

$$= z_2^{(i)}(1 - z_2^{(i)})$$

$$\delta_2^{(i)} = \frac{\partial L_{sq}}{\partial \hat{y}^{(i)}} \cdot \frac{\partial \hat{y}^{(i)}}{\partial u_2^{(i)}} = \frac{1}{N}(\hat{y}^{(i)} - y^{(i)}) \cdot z_2^{(i)}(1 - z_2^{(i)})$$

$$\delta_1^{(i)} = \underbrace{\frac{\partial L_{sq}}{\partial \hat{y}^{(i)}} \cdot \frac{\partial \hat{y}^{(i)}}{\partial u_2^{(i)}}}_{\delta_2^{(i)}} \cdot \frac{\partial u_2^{(i)}}{\partial z_1^{(i)}} \cdot \frac{\partial z_1^{(i)}}{\partial u_1^{(i)}} = \frac{1}{N}(\hat{y}^{(i)} - y^{(i)}) \cdot z_2^{(i)}(1 - z_2^{(i)}) \cdot w_2 \cdot z_1^{(i)}(1 - z_1^{(i)})$$

2. $\frac{\partial L_{sq}}{\partial w_2} = \sum_{i=1}^{N} \delta_2^{(i)} \cdot [1 \; z_1^{(i)}]$ $\quad$ $\frac{\partial L_{sq}}{\partial w_1} = \sum_{i=1}^{N} \delta_1^{(i)} [1 \; x^{(i)}]$

3. Denote learning rate as η.

$$w_2 = w_2 - \eta \frac{\partial L_{sq}}{\partial w_2} = w_2 - \eta \sum_{i=1}^{N} \delta_2^{(i)} [1 \; z_1^{(i)}]$$

$$w_1 = w_1 - \eta \frac{\partial L_{sq}}{\partial w_1} = w_1 - \eta \sum_{i=1}^{N} \delta_1^{(i)} [1 \; x^{(i)}]$$