DL Lab 02: Write a program to demonstrate the working of a deep neural network for classification task.

```
import math
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

def set_seed(seed=42):
    return np.random.default_rng(seed)

def one_hot(y, num_classes):
    z = np.zeros((y.size, num_classes), dtype=np.float32)
    z[np.arange(y.size), y] = 1.0
    return z

def accuracy(logits, y_true):
    return float((logits.argmax(axis=1) == y_true).mean())

class Linear:
    def __init__(self, in_dim, out_dim, rng=None):
        if rng is None: rng = set_seed(42)
        limit = math.sqrt(6.0 / (in_dim + out_dim))  # Xavier init
        self.W = rng.uniform(-limit, limit, size=(in_dim,
out_dim)).astype(np.float32)
        self.b = np.zeros((1, out_dim), dtype=np.float32)
        self.x = None
        self.dW = np.zeros_like(self.W)
        self.db = np.zeros_like(self.b)

    def forward(self, x):
        self.x = x
        return x @ self.W + self.b

    def backward(self, grad_out, l2_reg=0.0):
        self.dW = self.x.T @ grad_out + l2_reg * self.W
        self.db = grad_out.sum(axis=0, keepdims=True)
        return grad_out @ self.W.T

    def step(self, lr):
        self.W -= lr * self.dW
        self.b -= lr * self.db

class ReLU:
    def __init__(self): self.mask = None
```

```
    def forward(self, x):
        self.mask = x > 0
        return x * self.mask
    def backward(self, grad_out):
        return grad_out * self.mask

def softmax_logits(logits):
    z = logits - logits.max(axis=1, keepdims=True)
    expz = np.exp(z)
    return expz / (expz.sum(axis=1, keepdims=True) + 1e-12)

def cross_entropy_logits(logits, y_onehot):
    probs = softmax_logits(logits)
    loss = -np.sum(y_onehot * np.log(probs + 1e-12)) / logits.shape[0]
    return loss, probs

class MLP:
    def __init__(self, input_dim, hidden_dims, num_classes, rng=None):
        if rng is None: rng = set_seed(42)
        dims = [input_dim] + hidden_dims + [num_classes]
        self.layers = []
        for i in range(len(dims) - 1):
            self.layers.append(Linear(dims[i], dims[i+1], rng=rng))
            if i < len(dims) - 2:
                self.layers.append(ReLU())
        self.l2_reg = 0.0

    def forward(self, x):
        out = x
        for layer in self.layers:
            out = layer.forward(out) if isinstance(layer, Linear) else
layer.forward(out)
        return out

    def backward(self, grad_out):
        for layer in reversed(self.layers):
            if isinstance(layer, Linear):
                grad_out = layer.backward(grad_out,
l2_reg=self.l2_reg)
            else:
                grad_out = layer.backward(grad_out)

    def step(self, lr):
        for layer in self.layers:
            if isinstance(layer, Linear):
                layer.step(lr)

    def zero_grads(self):
        for layer in self.layers:
```

```
            if isinstance(layer, Linear):
                layer.dW.fill(0.0); layer.db.fill(0.0)

def iterate_minibatches(X, y, batch_size, rng=None):
    if rng is None: rng = set_seed(42)
    N = X.shape[0]; idx = np.arange(N); rng.shuffle(idx)
    for start in range(0, N, batch_size):
        batch = idx[start:start+batch_size]
        yield X[batch], y[batch]

def train(model, X_tr, y_tr, X_te, y_te,
          epochs=200, lr=0.05, batch_size=16, l2_reg=1e-4):
    rng = set_seed(123)
    model.l2_reg = l2_reg
    K = int(y_tr.max()) + 1
    for epoch in range(1, epochs + 1):
        for xb, yb in iterate_minibatches(X_tr, y_tr, batch_size,
rng=rng):
            yb_oh = one_hot(yb, K)
            model.zero_grads()
            logits = model.forward(xb)
            loss, probs = cross_entropy_logits(logits, yb_oh)
            grad_logits = (probs - yb_oh) / xb.shape[0]
            model.backward(grad_logits)
            model.step(lr)
        if epoch % 20 == 0 or epoch == 1:
            tr_acc = accuracy(model.forward(X_tr), y_tr)
            te_acc = accuracy(model.forward(X_te), y_te)
            print(f"Epoch {epoch:3d}/{epochs} | "
                  f"Train {tr_acc*100:5.2f}% | Test {te_acc*100:5.2f}
%")
    return accuracy(model.forward(X_tr), y_tr),
accuracy(model.forward(X_te), y_te)

def main():

    iris = load_iris()
    X, y = iris.data.astype(np.float32), iris.target.astype(np.int64)

    # Split into train and test
    X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2,
stratify=y, random_state=42)

    # Standardize
    scaler = StandardScaler()
    X_tr = scaler.fit_transform(X_tr)
    X_te = scaler.transform(X_te)

    print("Train shape:", X_tr.shape)
    print("Test shape:", X_te.shape)
```

```
    print("Classes:", np.unique(y))

    # Model: 4 -> 32 -> 32 -> 3
    model = MLP(input_dim=4, hidden_dims=[32, 32], num_classes=3)

    # Train
    train(model, X_tr, y_tr, X_te, y_te, epochs=200, lr=0.05,
batch_size=16)

    print("Final Train acc:", accuracy(model.forward(X_tr), y_tr)*100)
    print("Final Test  acc:", accuracy(model.forward(X_te), y_te)*100)

if __name__ == "__main__":
    main()
```

```
Train shape: (120, 4)
Test shape: (30, 4)
Classes: [0 1 2]
Epoch   1/200 | Train 61.67% | Test 66.67%
Epoch  20/200 | Train 90.83% | Test 86.67%
Epoch  40/200 | Train 96.67% | Test 93.33%
Epoch  60/200 | Train 97.50% | Test 93.33%
Epoch  80/200 | Train 98.33% | Test 93.33%
Epoch 100/200 | Train 98.33% | Test 93.33%
Epoch 120/200 | Train 98.33% | Test 96.67%
Epoch 140/200 | Train 98.33% | Test 96.67%
Epoch 160/200 | Train 98.33% | Test 96.67%
Epoch 180/200 | Train 98.33% | Test 96.67%
Epoch 200/200 | Train 98.33% | Test 96.67%
Final Train acc: 98.33333333333333
Final Test  acc: 96.66666666666667
```