1. Hyperparameter optimization

To install Optuna: run `pip install optuna` in console after activating your environment.

In [2]:
```
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, random_split, ConcatDataset, Subset
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np
import math
import optuna
import optuna.visualization as vis

seed = 42
torch.manual_seed(seed)
```

Out[2]:
```
<torch._C.Generator at 0x1d32a2c6970>
```

Load MNIST data. **DO NOT CHANGE THIS.**

In [47]:
```
def get_mnist_loaders(batch_size, valid_ratio, seed, downsample_ratio):
    # Define the transformation: convert to tensor and normalize
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])

    # Load the full training set (train=True returns the 60k training images)
    train_dataset = torchvision.datasets.MNIST(
        root="data",
        train=True,
        download=True,
        transform=transform
    )

    # Optionally downsample the training data
    if downsample_ratio < 1.0:
        subset_size = int(len(train_dataset) * downsample_ratio)
        indices = list(range(subset_size))
        train_dataset = Subset(train_dataset, indices)

    # Load the test set (train=False returns the 10k testing images)
    test_dataset = torchvision.datasets.MNIST(
        root="data",
        train=False,
        download=True,
        transform=transform
    )

    # Split training set into training and validation set
    train_size = int((1.0 - valid_ratio) * len(train_dataset))
```

```
    valid_size = len(train_dataset) - train_size
    gen = torch.Generator().manual_seed(seed)
    train_dataset, valid_dataset = random_split(train_dataset, [train_size, vali

    # DataLoaders
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True
    valid_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=Fals
    test_loader  = DataLoader(test_dataset,  batch_size=batch_size, shuffle=Fals

    return train_loader, valid_loader, test_loader

batch_size = 128
valid_ratio = 0.2
downsample_ratio = 0.5

# Create the DataLoaders
train_loader, valid_loader, test_loader = get_mnist_loaders(batch_size=batch_siz
                                                            valid_ratio=valid_
                                                            seed=seed,
                                                            downsample_ratio=d

# Total number of images in each dataset
print("Total training images:", len(train_loader.dataset))
print("Total validation images:", len(valid_loader.dataset))
print("Total test images:", len(test_loader.dataset))
```

```
Total training images: 24000
Total validation images: 6000
Total test images: 10000
```

(1a) Implement CNN

Write here the network structure as described in the exercise sheet.

Remember to **make the kernel size a parameter**, so that it can be controlled later.

In [8]:
```
class CNN(nn.Module):
    def __init__(self, kernel_size):
        super(CNN, self).__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=16, kernel_size=kernel_size),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(out_features=32), # infer the input shape
            nn.ReLU(),
            nn.Linear(in_features=32, out_features=10),
            nn.LogSoftmax()
        )

    def forward(self, x):
        #x = x.view(x.size(0), -1)
        return self.layers(x)

device = "cuda" if torch.cuda.is_available() else "cpu"
```

(1b & 1c) Train&Test Functions

To help focus on tuning optimizers and networks, we provided 3 pre-written functions for training and testing.

By having clear functions, we can easily compare different settings and keep the process consistent.

You do not need to change anything in this part.

In [49]:
```
# Training function.
def train(net, train_loader, parameters, epochs=1):
    optimizer = torch.optim.SGD(net.parameters(), lr=parameters["lr"], momentum=
    criterion = nn.NLLLoss()
    net.train()
    for epoch in range(epochs):
        for images, labels in train_loader:
            images = images.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()
            outputs = net(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
    return net

# Evaluation function.
def evaluate(net, data_loader):
    net.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in data_loader:
            images = images.to(device)
            labels = labels.to(device)

            outputs = net(images)
            predictions = outputs.argmax(dim=1)
            correct += (predictions == labels).sum().item()
            total += labels.size(0)
    accuracy = correct / total
    return accuracy

# Function for final training using best params on combined train and validation
def train_test(parameters):
    combined_dataset = ConcatDataset([train_loader.dataset, valid_loader.dataset
    combined_loader = DataLoader(combined_dataset, batch_size=batch_size, shuffl

    net = CNN(kernel_size=parameters["kernel_size"]).to(device)
    net = train(net, combined_loader, parameters)
    test_accuracy = evaluate(net, test_loader)
    return test_accuracy
```

(1b) Bayesian optimization using Optuna

Optuna requires an objective for the optimization.

Our objective returns validation accuracy, and we use **direction="maximize".**

So, we aim to maximize the accuracy of the validation dataset.

Here, your task is to define the hyperparameter search spaces (suggested hyperparameters).

Here you can see the documentary page for search spaces:

https://optuna.readthedocs.io/en/stable/tutorial/10_key_features/002_configurations.html

In [51]:
```python
def objective(trial, device=device):
    # Define hyperparameter search spaces.
    lr = trial.suggest_float("lr", 1e-6, 1e-2, log=True)
    momentum = trial.suggest_float("momentum", 0.0, 1.0)
    kernel_size = trial.suggest_categorical("kernel_size", [3, 5, 7])
    parameters = {"lr": lr, "momentum": momentum, "kernel_size": kernel_size}

    # Initialize the model with the chosen kernel_size.
    net = CNN(kernel_size=kernel_size).to(device)

    # Train the model on the training set.
    net = train(net, train_loader, parameters)

    # Evaluate validation sets.
    val_accuracy = evaluate(net, valid_loader)

    # Return the validation accuracy (this is what we want to maximize).
    return val_accuracy

# Create Optuna study and optimize.
n_trials = 75
sampler = optuna.samplers.TPESampler(seed=seed)
study = optuna.create_study(sampler=sampler, direction="maximize")
study.optimize(objective, n_trials=n_trials)

print("Best hyperparameters:", study.best_trial.params)
print("Best validation accuracy:", study.best_trial.value)
```

```
[I 2025-04-03 20:43:24,355] A new study created in memory with name: no-name-65ad
2872-f4f6-41a4-8e1e-d2649cc90b13
[I 2025-04-03 20:43:32,149] Trial 0 finished with value: 0.30783333333333335 and
parameters: {'lr': 3.1489116479568635e-05, 'momentum': 0.9507143064099162, 'kerne
l_size': 3}. Best is trial 0 with value: 0.30783333333333335.
[I 2025-04-03 20:43:39,814] Trial 1 finished with value: 0.10633333333333334 and
parameters: {'lr': 4.207053950287936e-06, 'momentum': 0.05808361216819946, 'kerne
l_size': 3}. Best is trial 0 with value: 0.30783333333333335.
[I 2025-04-03 20:43:47,491] Trial 2 finished with value: 0.1295 and parameters:
{'lr': 1.2087541473056965e-06, 'momentum': 0.9699098521619943, 'kernel_size': 3}.
Best is trial 0 with value: 0.30783333333333335.
[I 2025-04-03 20:43:55,263] Trial 3 finished with value: 0.06316666666666666 and
parameters: {'lr': 5.415244119402541e-06, 'momentum': 0.3042422429595377, 'kernel
_size': 3}. Best is trial 0 with value: 0.30783333333333335.
[I 2025-04-03 20:44:02,964] Trial 4 finished with value: 0.20516666666666666 and
parameters: {'lr': 0.0002801635158716264, 'momentum': 0.13949386065204183, 'kerne
l_size': 7}. Best is trial 0 with value: 0.30783333333333335.
[I 2025-04-03 20:44:10,567] Trial 5 finished with value: 0.6121666666666666 and p
arameters: {'lr': 0.0013826232179369874, 'momentum': 0.19967378215835974, 'kernel
_size': 5}. Best is trial 5 with value: 0.6121666666666666.
[I 2025-04-03 20:44:18,332] Trial 6 finished with value: 0.13566666666666666 and
parameters: {'lr': 0.000269264691008618, 'momentum': 0.17052412368729153, 'kernel
_size': 7}. Best is trial 5 with value: 0.6121666666666666.
[I 2025-04-03 20:44:26,198] Trial 7 finished with value: 0.693 and parameters:
{'lr': 0.0017123375973163992, 'momentum': 0.3046137691733707, 'kernel_size': 5}.
Best is trial 7 with value: 0.693.
[I 2025-04-03 20:44:34,242] Trial 8 finished with value: 0.059833333333333336 and
parameters: {'lr': 3.0771802712506896e-06, 'momentum': 0.4951769101112702, 'kerne
l_size': 5}. Best is trial 7 with value: 0.693.
[I 2025-04-03 20:44:42,260] Trial 9 finished with value: 0.43416666666666665 and
parameters: {'lr': 0.0004467752817973908, 'momentum': 0.31171107608941095, 'kerne
l_size': 5}. Best is trial 7 with value: 0.693.
[I 2025-04-03 20:44:50,095] Trial 10 finished with value: 0.9081666666666667 and
parameters: {'lr': 0.00804836998643663, 'momentum': 0.6998750099215975, 'kernel_s
ize': 5}. Best is trial 10 with value: 0.9081666666666667.
[I 2025-04-03 20:44:57,991] Trial 11 finished with value: 0.9028333333333334 and
parameters: {'lr': 0.006359752317113245, 'momentum': 0.7151696121967952, 'kernel_
size': 5}. Best is trial 10 with value: 0.9081666666666667.
[I 2025-04-03 20:45:05,754] Trial 12 finished with value: 0.9226666666666666 and
parameters: {'lr': 0.008690693327164533, 'momentum': 0.7763994737686188, 'kernel_
size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:45:13,673] Trial 13 finished with value: 0.913 and parameters:
{'lr': 0.009789042109785687, 'momentum': 0.7274274902192149, 'kernel_size': 5}. B
est is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:45:21,094] Trial 14 finished with value: 0.8558333333333333 and
parameters: {'lr': 0.0021256335538485, 'momentum': 0.7628213145000068, 'kernel_si
ze': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:45:28,810] Trial 15 finished with value: 0.09316666666666666 and
parameters: {'lr': 7.392865547996866e-05, 'momentum': 0.5732909776512292, 'kernel
_size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:45:36,569] Trial 16 finished with value: 0.895 and parameters:
{'lr': 0.003704799518133014, 'momentum': 0.8406022520905049, 'kernel_size': 7}. B
est is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:45:44,472] Trial 17 finished with value: 0.8935 and parameters:
{'lr': 0.009886969146409335, 'momentum': 0.5556314909508107, 'kernel_size': 5}. B
est is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:45:52,138] Trial 18 finished with value: 0.741 and parameters:
{'lr': 0.0006377498741418543, 'momentum': 0.8340391599421042, 'kernel_size': 5}.
Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:45:59,839] Trial 19 finished with value: 0.1425 and parameters:
```

{'lr': 2.350699706323715e-05, 'momentum': 0.6231471223936182, 'kernel_size': 7}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:46:07,245] Trial 20 finished with value: 0.5905 and parameters: {'lr': 0.0008387744815011899, 'momentum': 0.41692460832750067, 'kernel_size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:46:14,875] Trial 21 finished with value: 0.8881666666666667 and parameters: {'lr': 0.004172606654984391, 'momentum': 0.6882295119501993, 'kernel_size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:46:22,390] Trial 22 finished with value: 0.9196666666666666 and parameters: {'lr': 0.009425259895470882, 'momentum': 0.8338902615189306, 'kernel_size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:46:30,025] Trial 23 finished with value: 0.8933333333333333 and parameters: {'lr': 0.003296141715525719, 'momentum': 0.8474121593576213, 'kernel_size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:46:37,756] Trial 24 finished with value: 0.9226666666666666 and parameters: {'lr': 0.008793200523133444, 'momentum': 0.9012244418301453, 'kernel_size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:46:45,523] Trial 25 finished with value: 0.9065 and parameters: {'lr': 0.0026888372747929435, 'momentum': 0.9168467349094407, 'kernel_size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:46:53,262] Trial 26 finished with value: 0.8626666666666667 and parameters: {'lr': 0.0010480351955366762, 'momentum': 0.8943870116881033, 'kernel_size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:47:00,728] Trial 27 finished with value: 0.8735 and parameters: {'lr': 0.005122765982001854, 'momentum': 0.998353191024608, 'kernel_size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:47:08,388] Trial 28 finished with value: 0.855 and parameters: {'lr': 0.001922833312450667, 'momentum': 0.7889420478841087, 'kernel_size': 3}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:47:16,043] Trial 29 finished with value: 0.14483333333333334 and parameters: {'lr': 9.537707326117227e-05, 'momentum': 0.6293286660298155, 'kernel_size': 7}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:47:23,557] Trial 30 finished with value: 0.1945 and parameters: {'lr': 2.7317423776726492e-05, 'momentum': 0.8984199737670919, 'kernel_size': 3}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:47:31,146] Trial 31 finished with value: 0.9036666666666666 and parameters: {'lr': 0.009394996561327059, 'momentum': 0.753133993234806, 'kernel_size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:47:39,498] Trial 32 finished with value: 0.8998333333333334 and parameters: {'lr': 0.005286419276776037, 'momentum': 0.8153815397318148, 'kernel_size': 5}. Best is trial 12 with value: 0.9226666666666666.
[I 2025-04-03 20:47:46,947] Trial 33 finished with value: 0.9361666666666667 and parameters: {'lr': 0.006021076278324688, 'momentum': 0.9482512398565922, 'kernel_size': 5}. Best is trial 33 with value: 0.9361666666666667.
[I 2025-04-03 20:47:54,715] Trial 34 finished with value: 0.904 and parameters: {'lr': 0.003139981694462376, 'momentum': 0.9344054569925702, 'kernel_size': 3}. Best is trial 33 with value: 0.9361666666666667.
[I 2025-04-03 20:48:02,199] Trial 35 finished with value: 0.94 and parameters: {'lr': 0.004546944746283349, 'momentum': 0.949881955759221, 'kernel_size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:48:10,427] Trial 36 finished with value: 0.9371666666666667 and parameters: {'lr': 0.005097670010864789, 'momentum': 0.9766250107093642, 'kernel_size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:48:18,214] Trial 37 finished with value: 0.8326666666666667 and parameters: {'lr': 0.00020973196426575736, 'momentum': 0.9938711903257784, 'kernel_size': 3}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:48:26,037] Trial 38 finished with value: 0.8985 and parameters: {'lr': 0.0014464408316890475, 'momentum': 0.9515797161254608, 'kernel_size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:48:33,778] Trial 39 finished with value: 0.8766666666666667 and

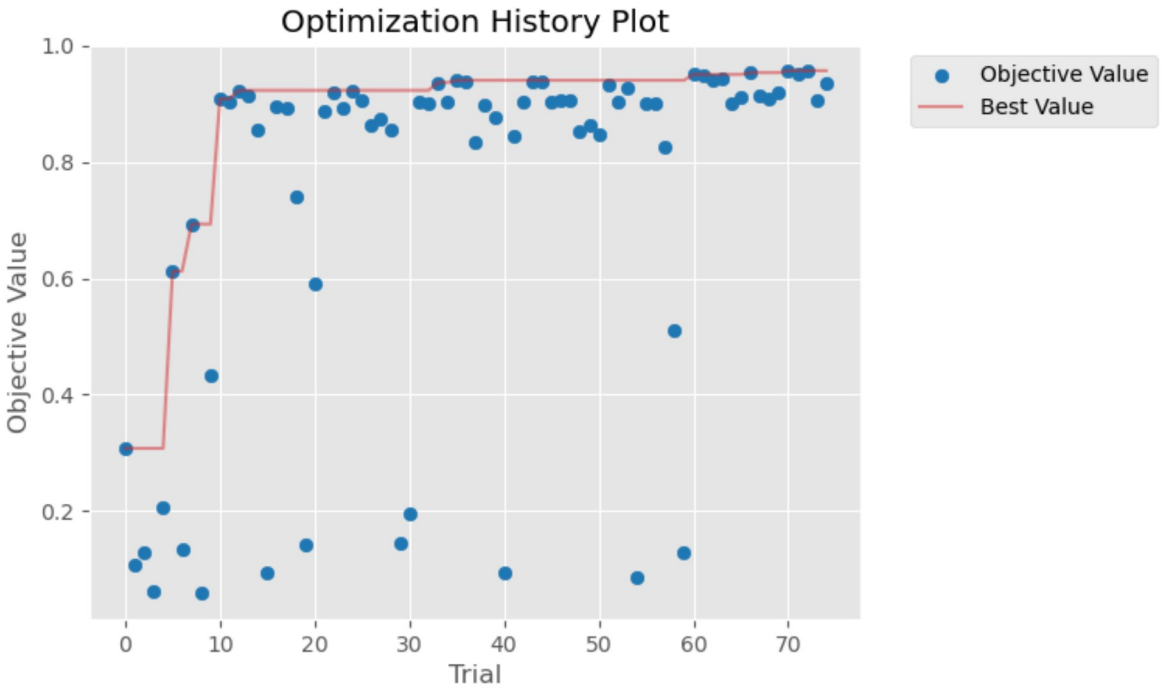
```
parameters: {'lr': 0.0006558301188405286, 'momentum': 0.9511472856217792, 'kernel
_size': 7}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:48:41,337] Trial 40 finished with value: 0.09516666666666666 and
parameters: {'lr': 7.770812556474302e-06, 'momentum': 0.8824663824648549, 'kernel
_size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:48:49,228] Trial 41 finished with value: 0.8436666666666667 and
parameters: {'lr': 0.005754021219027928, 'momentum': 0.015021211625687259, 'kerne
l_size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:48:56,925] Trial 42 finished with value: 0.904 and parameters:
{'lr': 0.002912877606666304, 'momentum': 0.8731424285218724, 'kernel_size': 5}. B
est is trial 35 with value: 0.94.
[I 2025-04-03 20:49:04,863] Trial 43 finished with value: 0.9383333333333334 and
parameters: {'lr': 0.0059122506057212145, 'momentum': 0.952316590053456, 'kernel_
size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:49:12,960] Trial 44 finished with value: 0.9376666666666666 and
parameters: {'lr': 0.004767719464708509, 'momentum': 0.9627980486022851, 'kernel_
size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:49:21,184] Trial 45 finished with value: 0.9021666666666667 and
parameters: {'lr': 0.0011885552709944261, 'momentum': 0.9608771169328997, 'kernel
_size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:49:29,164] Trial 46 finished with value: 0.9045 and parameters:
{'lr': 0.0016797641961200248, 'momentum': 0.9696048638251106, 'kernel_size': 5}.
Best is trial 35 with value: 0.94.
[I 2025-04-03 20:49:36,813] Trial 47 finished with value: 0.9048333333333334 and
parameters: {'lr': 0.005134027833294398, 'momentum': 0.9948683118418747, 'kernel_
size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:49:46,422] Trial 48 finished with value: 0.8518333333333333 and
parameters: {'lr': 0.0004506072267802553, 'momentum': 0.940206992576823, 'kernel_
size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:49:55,174] Trial 49 finished with value: 0.8621666666666666 and
parameters: {'lr': 0.002269163553358945, 'momentum': 0.7927834872364001, 'kernel_
size': 3}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:50:04,400] Trial 50 finished with value: 0.846 and parameters:
{'lr': 0.0041241463974962915, 'momentum': 0.4103969867133104, 'kernel_size': 7}.
Best is trial 35 with value: 0.94.
[I 2025-04-03 20:50:12,873] Trial 51 finished with value: 0.9315 and parameters:
{'lr': 0.00659261821958841, 'momentum': 0.9256261373006919, 'kernel_size': 5}. Be
st is trial 35 with value: 0.94.
[I 2025-04-03 20:50:20,690] Trial 52 finished with value: 0.9033333333333333 and
parameters: {'lr': 0.00614580278743663, 'momentum': 0.8655678658018988, 'kernel_s
ize': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:50:28,636] Trial 53 finished with value: 0.9275 and parameters:
{'lr': 0.006439468072323514, 'momentum': 0.9272108912690962, 'kernel_size': 5}. B
est is trial 35 with value: 0.94.
[I 2025-04-03 20:50:36,412] Trial 54 finished with value: 0.086 and parameters:
{'lr': 1.3189728028275167e-06, 'momentum': 0.9974679868442358, 'kernel_size': 5}.
Best is trial 35 with value: 0.94.
[I 2025-04-03 20:50:44,601] Trial 55 finished with value: 0.8993333333333333 and
parameters: {'lr': 0.0022478818086571575, 'momentum': 0.9166417602821981, 'kernel
_size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:50:52,323] Trial 56 finished with value: 0.901 and parameters:
{'lr': 0.004620418460225269, 'momentum': 0.8663379067234439, 'kernel_size': 5}. B
est is trial 35 with value: 0.94.
[I 2025-04-03 20:50:59,824] Trial 57 finished with value: 0.8256666666666667 and
parameters: {'lr': 0.0035833155728069014, 'momentum': 0.24387556050184256, 'kerne
l_size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:51:07,396] Trial 58 finished with value: 0.51 and parameters:
{'lr': 0.0001798094113210682, 'momentum': 0.8093057853129726, 'kernel_size': 5}.
Best is trial 35 with value: 0.94.
[I 2025-04-03 20:51:14,867] Trial 59 finished with value: 0.12866666666666668 and
```

```
parameters: {'lr': 5.5167891299689404e-05, 'momentum': 0.1332282148956756, 'kerne
l_size': 5}. Best is trial 35 with value: 0.94.
[I 2025-04-03 20:51:22,293] Trial 60 finished with value: 0.95 and parameters:
{'lr': 0.006910945017891391, 'momentum': 0.966013113804598, 'kernel_size': 5}. Be
st is trial 60 with value: 0.95.
[I 2025-04-03 20:51:30,255] Trial 61 finished with value: 0.947 and parameters:
{'lr': 0.008020150018793838, 'momentum': 0.944767570150608, 'kernel_size': 5}. Be
st is trial 60 with value: 0.95.
[I 2025-04-03 20:51:38,162] Trial 62 finished with value: 0.9403333333333334 and
parameters: {'lr': 0.0072489751365700855, 'momentum': 0.9647248812852208, 'kernel
_size': 5}. Best is trial 60 with value: 0.95.
[I 2025-04-03 20:51:45,863] Trial 63 finished with value: 0.942 and parameters:
{'lr': 0.007535794349967179, 'momentum': 0.9717257100050071, 'kernel_size': 5}. B
est is trial 60 with value: 0.95.
[I 2025-04-03 20:51:53,562] Trial 64 finished with value: 0.899 and parameters:
{'lr': 0.0025886631023247556, 'momentum': 0.8949265244294172, 'kernel_size': 5}.
Best is trial 60 with value: 0.95.
[I 2025-04-03 20:52:01,031] Trial 65 finished with value: 0.9115 and parameters:
{'lr': 0.008323073626435503, 'momentum': 0.8469241635774853, 'kernel_size': 5}. B
est is trial 60 with value: 0.95.
[I 2025-04-03 20:52:08,633] Trial 66 finished with value: 0.9531666666666667 and
parameters: {'lr': 0.007575488158169283, 'momentum': 0.9654246182773373, 'kernel_
size': 7}. Best is trial 66 with value: 0.9531666666666667.
[I 2025-04-03 20:52:16,313] Trial 67 finished with value: 0.913 and parameters:
{'lr': 0.007199612406912263, 'momentum': 0.7309704860387868, 'kernel_size': 7}. B
est is trial 66 with value: 0.9531666666666667.
[I 2025-04-03 20:52:24,113] Trial 68 finished with value: 0.9088333333333334 and
parameters: {'lr': 0.0037073917859266475, 'momentum': 0.9009648419813396, 'kernel
_size': 7}. Best is trial 66 with value: 0.9531666666666667.
[I 2025-04-03 20:52:31,553] Trial 69 finished with value: 0.9196666666666666 and
parameters: {'lr': 0.00744030376008257, 'momentum': 0.8398384561816135, 'kernel_s
ize': 7}. Best is trial 66 with value: 0.9531666666666667.
[I 2025-04-03 20:52:39,422] Trial 70 finished with value: 0.9565 and parameters:
{'lr': 0.009225749809973626, 'momentum': 0.9738151097595154, 'kernel_size': 7}. B
est is trial 70 with value: 0.9565.
[I 2025-04-03 20:52:47,276] Trial 71 finished with value: 0.9503333333333334 and
parameters: {'lr': 0.009888454088669183, 'momentum': 0.9716016784027977, 'kernel_
size': 7}. Best is trial 70 with value: 0.9565.
[I 2025-04-03 20:52:55,132] Trial 72 finished with value: 0.9558333333333333 and
parameters: {'lr': 0.008321778916063569, 'momentum': 0.9709621163163298, 'kernel_
size': 7}. Best is trial 70 with value: 0.9565.
[I 2025-04-03 20:53:02,637] Trial 73 finished with value: 0.9066666666666666 and
parameters: {'lr': 0.009838045848344029, 'momentum': 0.9859697657840024, 'kernel_
size': 7}. Best is trial 70 with value: 0.9565.
[I 2025-04-03 20:53:10,425] Trial 74 finished with value: 0.9336666666666666 and
parameters: {'lr': 0.007667344214043302, 'momentum': 0.9171047750493966, 'kernel_
size': 7}. Best is trial 70 with value: 0.9565.
Best hyperparameters: {'lr': 0.009225749809973626, 'momentum': 0.9738151097595154
, 'kernel_size': 7}
Best validation accuracy: 0.9565
```

In [52]:

```python
# Visualize the optimization history. (Interactive)
#fig_history = vis.plot_optimization_history(study)
#fig_history.show()

# Alternative Visualization. You can use this if the plot is not visible in .pdf
fig_history = optuna.visualization.matplotlib.plot_optimization_history(study)
plt.show()
```

```
C:\Users\Matias\AppData\Local\Temp\ipykernel_20196\2604171193.py:6: ExperimentalW
arning: plot_optimization_history is experimental (supported from v2.2.0). The in
terface can change in the future.
  fig_history = optuna.visualization.matplotlib.plot_optimization_history(study)
```



In [55]:
```python
# Evaluate on the test set using the best hyperparameters.
test_accuracy = train_test(study.best_trial.params)
print("Test accuracy with best hyperparameters:", test_accuracy)
```

```
Test accuracy with best hyperparameters: 0.9659
```

(1c) Grid search

Implement grid search for the same three parameters.

Remember that you **need to cover 75 alternatives in total.**

So, think about how to best allocate those to cover the three hyperparameters.

In [73]:
```python
lr = np.geomspace(1e-6, 1e-2, 5)
momentum = np.arange(0.1, 1.0, 0.2)
kernel_size = np.array([3, 5, 7])

parameters = {"lr": lr, "momentum": momentum, "kernel_size": kernel_size}

best_acc = 0.0
best_acc_history = {"trial": [], "acc": []}
best_params = {}
val_accs = []
n_trial = 1

for l in lr:
    for m in momentum:
        for k in kernel_size:
            net = CNN(kernel_size=k).to(device)
            parameters = {"lr": l, "momentum": m, "kernel_size": k}
            net = train(net, train_loader, parameters)
```

```
            val_accuracy = evaluate(net, valid_loader)

            if val_accuracy > best_acc:
                best_acc = val_accuracy
                best_acc_history["trial"].append(n_trial)
                best_acc_history["acc"].append(best_acc)
                best_params = parameters

            val_accs.append(val_accuracy)

            print(f"Trial {n_trial} finished")
            print(f"Parameters : {parameters}")
            print(f"Validation accuracy: {val_accuracy}")
            print()
            n_trial += 1

print(f"Best parameters: {best_params}")
print(f"Best validation accuracy: {best_acc}")
```

```
c:\Users\Matias\miniconda3\Lib\site-packages\torch\nn\modules\module.py:1739: UserWarning: Implicit dimension choice for log_softmax has been deprecated. Change the call to include dim=X as an argument.
  return self._call_impl(*args, **kwargs)
```

```
Trial 1 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.1), 'kernel_size'
: np.int64(3)}
Validation accuracy: 0.06383333333333334

Trial 2 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.1), 'kernel_size'
: np.int64(5)}
Validation accuracy: 0.0745

Trial 3 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.1), 'kernel_size'
: np.int64(7)}
Validation accuracy: 0.09366666666666666

Trial 4 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.3000000000000000
4), 'kernel_size': np.int64(3)}
Validation accuracy: 0.09266666666666666

Trial 5 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.3000000000000000
4), 'kernel_size': np.int64(5)}
Validation accuracy: 0.14316666666666666

Trial 6 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.3000000000000000
4), 'kernel_size': np.int64(7)}
Validation accuracy: 0.09216666666666666

Trial 7 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.5000000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.11716666666666667

Trial 8 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.5000000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.11083333333333334

Trial 9 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.5000000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.1005

Trial 10 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.7000000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.10733333333333334

Trial 11 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.7000000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.08066666666666666

Trial 12 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.7000000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.09716666666666667
```

```
Trial 13 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.900000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.08283333333333333

Trial 14 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.900000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.0955

Trial 15 finished
Parameters : {'lr': np.float64(1e-06), 'momentum': np.float64(0.900000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.11483333333333333

Trial 16 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.1), 'kernel_size'
: np.int64(3)}
Validation accuracy: 0.092

Trial 17 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.1), 'kernel_size'
: np.int64(5)}
Validation accuracy: 0.1435

Trial 18 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.1), 'kernel_size'
: np.int64(7)}
Validation accuracy: 0.0935

Trial 19 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.300000000000000
4), 'kernel_size': np.int64(3)}
Validation accuracy: 0.12

Trial 20 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.300000000000000
4), 'kernel_size': np.int64(5)}
Validation accuracy: 0.08783333333333333

Trial 21 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.300000000000000
4), 'kernel_size': np.int64(7)}
Validation accuracy: 0.069

Trial 22 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.500000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.13416666666666666

Trial 23 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.500000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.09216666666666666

Trial 24 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.500000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.043166666666666666
```

```
Trial 25 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.700000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.11466666666666667

Trial 26 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.700000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.11116666666666666

Trial 27 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.700000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.137

Trial 28 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.900000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.226

Trial 29 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.900000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.08316666666666667

Trial 30 finished
Parameters : {'lr': np.float64(1e-05), 'momentum': np.float64(0.900000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.12183333333333334

Trial 31 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.1), 'kernel_size
': np.int64(3)}
Validation accuracy: 0.17516666666666666

Trial 32 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.1), 'kernel_size
': np.int64(5)}
Validation accuracy: 0.131

Trial 33 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.1), 'kernel_size
': np.int64(7)}
Validation accuracy: 0.13183333333333333

Trial 34 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.300000000000000
4), 'kernel_size': np.int64(3)}
Validation accuracy: 0.28

Trial 35 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.300000000000000
4), 'kernel_size': np.int64(5)}
Validation accuracy: 0.2945

Trial 36 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.300000000000000
4), 'kernel_size': np.int64(7)}
Validation accuracy: 0.1585
```

```
Trial 37 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.500000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.15

Trial 38 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.500000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.11166666666666666

Trial 39 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.500000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.19816666666666666

Trial 40 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.700000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.2786666666666667

Trial 41 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.700000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.2985

Trial 42 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.700000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.20283333333333334

Trial 43 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.900000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.16983333333333334

Trial 44 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.900000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.5056666666666667

Trial 45 finished
Parameters : {'lr': np.float64(0.0001), 'momentum': np.float64(0.900000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.40066666666666667

Trial 46 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.1), 'kernel_size'
: np.int64(3)}
Validation accuracy: 0.5953333333333334

Trial 47 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.1), 'kernel_size'
: np.int64(5)}
Validation accuracy: 0.559

Trial 48 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.1), 'kernel_size'
: np.int64(7)}
Validation accuracy: 0.5093333333333333
```

```
Trial 49 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.3000000000000000
4), 'kernel_size': np.int64(3)}
Validation accuracy: 0.6576666666666666

Trial 50 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.3000000000000000
4), 'kernel_size': np.int64(5)}
Validation accuracy: 0.58

Trial 51 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.3000000000000000
4), 'kernel_size': np.int64(7)}
Validation accuracy: 0.5596666666666666

Trial 52 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.5000000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.6743333333333333

Trial 53 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.5000000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.7183333333333334

Trial 54 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.5000000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.6113333333333333

Trial 55 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.7000000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.8135

Trial 56 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.7000000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.673

Trial 57 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.7000000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.7268333333333333

Trial 58 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.9000000000000000
1), 'kernel_size': np.int64(3)}
Validation accuracy: 0.8556666666666667

Trial 59 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.9000000000000000
1), 'kernel_size': np.int64(5)}
Validation accuracy: 0.8668333333333333

Trial 60 finished
Parameters : {'lr': np.float64(0.001), 'momentum': np.float64(0.9000000000000000
1), 'kernel_size': np.int64(7)}
Validation accuracy: 0.8593333333333333
```

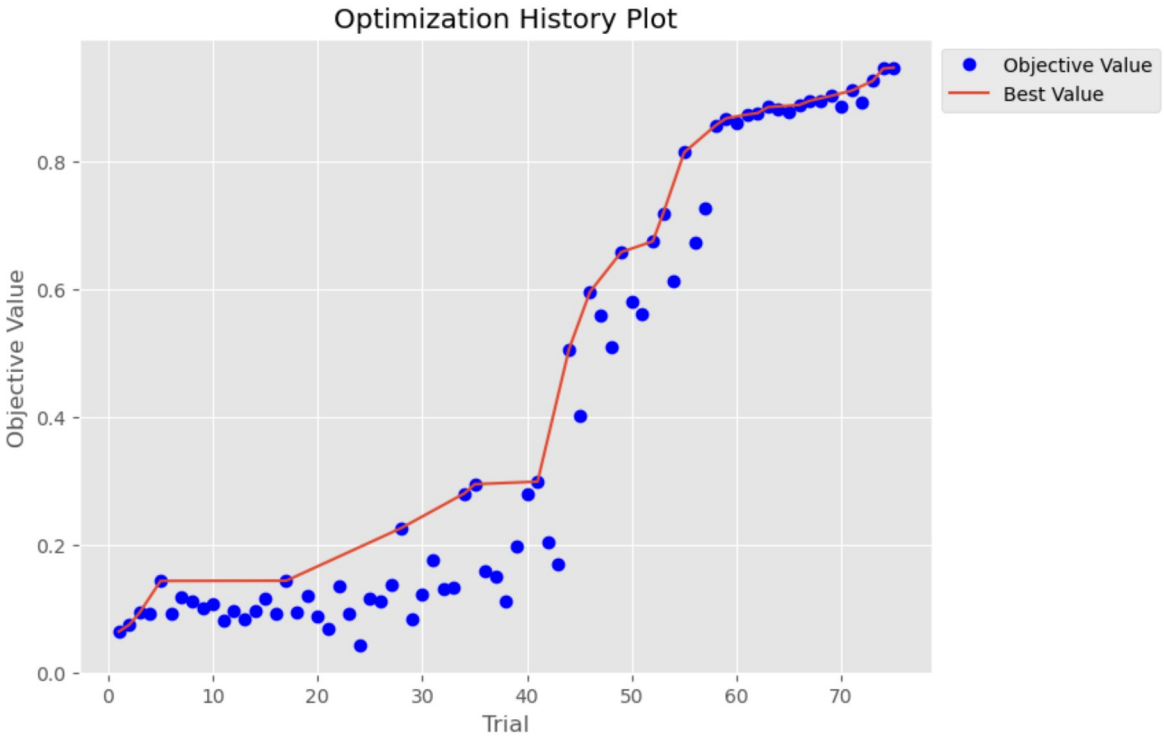
```
Trial 61 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.1), 'kernel_size':
np.int64(3)}
Validation accuracy: 0.8728333333333333

Trial 62 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.1), 'kernel_size':
np.int64(5)}
Validation accuracy: 0.875

Trial 63 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.1), 'kernel_size':
np.int64(7)}
Validation accuracy: 0.8843333333333333

Trial 64 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.3000000000000000
4), 'kernel_size': np.int64(3)}
Validation accuracy: 0.8808333333333334

Trial 65 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.3000000000000000
4), 'kernel_size': np.int64(5)}
Validation accuracy: 0.8766666666666667

Trial 66 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.3000000000000000
4), 'kernel_size': np.int64(7)}
Validation accuracy: 0.8878333333333334

Trial 67 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.5000000000000001),
'kernel_size': np.int64(3)}
Validation accuracy: 0.8943333333333333

Trial 68 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.5000000000000001),
'kernel_size': np.int64(5)}
Validation accuracy: 0.893

Trial 69 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.5000000000000001),
'kernel_size': np.int64(7)}
Validation accuracy: 0.9018333333333334

Trial 70 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.7000000000000001),
'kernel_size': np.int64(3)}
Validation accuracy: 0.886

Trial 71 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.7000000000000001),
'kernel_size': np.int64(5)}
Validation accuracy: 0.9101666666666667

Trial 72 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.7000000000000001),
'kernel_size': np.int64(7)}
Validation accuracy: 0.8916666666666667
```

```
Trial 73 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.9000000000000001),
'kernel_size': np.int64(3)}
Validation accuracy: 0.9253333333333333

Trial 74 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.9000000000000001),
'kernel_size': np.int64(5)}
Validation accuracy: 0.9446666666666667

Trial 75 finished
Parameters : {'lr': np.float64(0.01), 'momentum': np.float64(0.9000000000000001),
'kernel_size': np.int64(7)}
Validation accuracy: 0.9461666666666667

Best parameters: {'lr': np.float64(0.01), 'momentum': np.float64(0.90000000000000
01), 'kernel_size': np.int64(7)}
Best validation accuracy: 0.9461666666666667
```

In [74]:
```
test_accuracy = train_test(best_params)
print("Test accuracy with best hyperparameters:", test_accuracy)
```

```
Test accuracy with best hyperparameters: 0.9545
```

In [75]:
```
plt.figure(figsize=(8, 6))
plt.plot(range(1, len(val_accs) + 1), val_accs, "bo", label="Objective Value")
plt.plot(best_acc_history["trial"], best_acc_history["acc"], label="Best Value")
plt.xlabel("Trial")
plt.ylabel("Objective Value")
plt.title("Optimization History Plot")
plt.legend(loc="upper left", bbox_to_anchor=(1,1))
plt.show()
```



Both methods are close in test accuracy, but Bayesian optimization achieves better results faster and I would assume the difference is even larger with bigger search space. Grid search is just computationally so expensive and grows exponentially with the number of parameter values to test. If the search space is small then grid search would be suitable.

2. Transfer learning

In [9]:
```
import random

seed = 42
torch.manual_seed(seed)
random.seed(seed)
```

Model - **DO NOT CHANGE THIS**

In [10]:
```
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()

        # Block 1
        self.cnn1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5, str
        self.relu1 = nn.ReLU()
        self.bn1 = nn.BatchNorm2d(16)
        self.MaxPool1 = nn.MaxPool2d(kernel_size=2)

        # Block 2
        self.cnn2 = nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2)
        self.relu2 = nn.ReLU()
        self.bn2 = nn.BatchNorm2d(32)
        self.MaxPool2 = nn.MaxPool2d(kernel_size=2)

        # Block 3
        self.cnn3 = nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2)
        self.relu3 = nn.ReLU()
        self.bn3 = nn.BatchNorm2d(64)
        self.MaxPool3 = nn.MaxPool2d(kernel_size=2)

        self.fc1 = nn.Linear(576, 10)  # Fully connected layer

    def forward(self, x):
        out = self.MaxPool1(self.bn1(self.relu1(self.cnn1(x))))
        out = self.MaxPool2(self.bn2(self.relu2(self.cnn2(out))))
        out = self.MaxPool3(self.bn3(self.relu3(self.cnn3(out))))

        out = out.view(out.size(0), -1)
        out = self.fc1(out)

        return out
```

Load Fashion-MNIST data. 128 training samples, 10k test samples. **DO NOT CHANGE THIS**

In [11]:
```
transform = transforms.Compose([transforms.ToTensor()])

# Load full dataset
```

```
full_train_dataset = torchvision.datasets.FashionMNIST(root="./data", train=True
test_dataset = torchvision.datasets.FashionMNIST(root="./data", train=False, dow

batch_size = 128

# Slice small set of data
indices = random.sample(range(len(full_train_dataset)), batch_size)
train_dataset = Subset(full_train_dataset, indices)

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

print("Training set size:", len(train_loader.dataset))
print("Test set size:", len(test_loader.dataset))
```

```
Training set size: 128
Test set size: 10000
```

We recommend writing a general training function that implements all three alternative models.

Below **train_model()** function is a template that already has some functionality for storing the errors, doing evaluation over test dataset and printing results.

You should think about how to implement alternative ways of doing transfer learning.

Once you have this function, you can simply run the following cells to try out the alternative approaches.

In [12]:
```
# Training function
def train_model(model, train_loader, test_loader, optimizer, criterion, epochs,

    if freeze_layers:
        # freeze all layers
        for param in model.parameters():
            param.requires_grad = False
        # unfreeze the last layer
        for param in model.fc1.parameters():
            param.requires_grad = True
        #num_ftrs = model.fc1.in_features
        #num_output = model.fc1.out_features
        #model.fc1 = nn.Linear(num_ftrs, num_output)


    train_losses = []
    test_losses = []
    train_accs = []
    test_accs = []

    # Training Loop
    for epoch in range(epochs):
        model.train()
        total_loss, correct, total = 0, 0, 0
        for images, labels in train_loader:
            images = images.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()
```

```
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            total_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)
        train_losses.append(total_loss / total)
        train_accs.append(100 * correct / total)

        # Evaluation
        model.eval()
        test_loss_total, correct, total = 0, 0, 0
        with torch.no_grad():
            for images, labels in test_loader:
                images = images.to(device)
                labels = labels.to(device)

                outputs = model(images)
                loss = criterion(outputs, labels)
                test_loss_total += loss.item() * images.size(0)
                _, predicted = torch.max(outputs, 1)
                correct += (predicted == labels).sum().item()
                total += labels.size(0)
        test_losses.append(test_loss_total / total)
        test_accs.append(100 * correct / total)

        print(f"Epoch {epoch+1:>2}/{epochs} | "
              f"Train Loss: {train_losses[-1]:.4f} | Train Acc: {train_accs[-1]:
              f"Test Loss: {test_losses[-1]:.4f} | Test Acc: {test_accs[-1]:.2f}

    return train_losses, test_losses, train_accs, test_accs
```

(2a) From Scratch

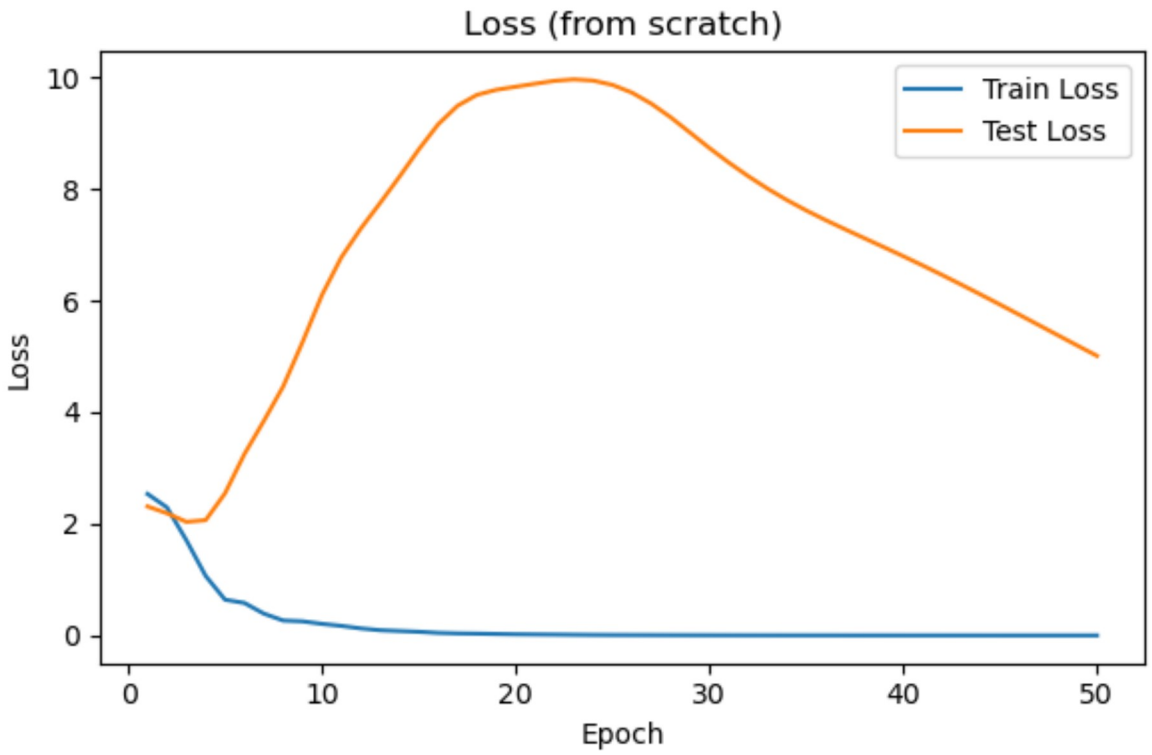
Create a new CNNModel and train from scratch.

In [13]:
```
model1 = CNNModel().to(device)
criterion = nn.CrossEntropyLoss()
optimizer1 = torch.optim.Adam(model1.parameters(), lr=0.01)
epochs = 50
freeze_layers = False
train_loss1, test_loss1, train_acc1, test_acc1 = train_model(model1, train_loade
                                                             criterion, epochs=e
```

```
Epoch  1/50 | Train Loss: 2.5355 | Train Acc: 10.16% | Test Loss: 2.3114 | Test A
cc: 18.27%
Epoch  2/50 | Train Loss: 2.2939 | Train Acc: 59.38% | Test Loss: 2.1896 | Test A
cc: 16.89%
Epoch  3/50 | Train Loss: 1.7081 | Train Acc: 48.44% | Test Loss: 2.0351 | Test A
cc: 29.10%
Epoch  4/50 | Train Loss: 1.0660 | Train Acc: 71.09% | Test Loss: 2.0655 | Test A
cc: 31.40%
Epoch  5/50 | Train Loss: 0.6407 | Train Acc: 80.47% | Test Loss: 2.5470 | Test A
cc: 19.26%
Epoch  6/50 | Train Loss: 0.5822 | Train Acc: 79.69% | Test Loss: 3.2496 | Test A
cc: 10.33%
Epoch  7/50 | Train Loss: 0.3915 | Train Acc: 81.25% | Test Loss: 3.8419 | Test A
cc: 10.02%
Epoch  8/50 | Train Loss: 0.2675 | Train Acc: 89.06% | Test Loss: 4.4610 | Test A
cc: 10.00%
Epoch  9/50 | Train Loss: 0.2527 | Train Acc: 90.62% | Test Loss: 5.2563 | Test A
cc: 10.02%
Epoch 10/50 | Train Loss: 0.2078 | Train Acc: 93.75% | Test Loss: 6.0982 | Test A
cc: 10.13%
Epoch 11/50 | Train Loss: 0.1720 | Train Acc: 95.31% | Test Loss: 6.7728 | Test A
cc: 10.08%
Epoch 12/50 | Train Loss: 0.1263 | Train Acc: 96.88% | Test Loss: 7.2865 | Test A
cc: 10.00%
Epoch 13/50 | Train Loss: 0.0940 | Train Acc: 97.66% | Test Loss: 7.7457 | Test A
cc: 10.00%
Epoch 14/50 | Train Loss: 0.0797 | Train Acc: 99.22% | Test Loss: 8.2176 | Test A
cc: 10.00%
Epoch 15/50 | Train Loss: 0.0661 | Train Acc: 99.22% | Test Loss: 8.7071 | Test A
cc: 10.00%
Epoch 16/50 | Train Loss: 0.0458 | Train Acc: 99.22% | Test Loss: 9.1570 | Test A
cc: 10.00%
Epoch 17/50 | Train Loss: 0.0367 | Train Acc: 100.00% | Test Loss: 9.4894 | Test
Acc: 10.00%
Epoch 18/50 | Train Loss: 0.0325 | Train Acc: 100.00% | Test Loss: 9.6841 | Test
Acc: 10.01%
Epoch 19/50 | Train Loss: 0.0265 | Train Acc: 100.00% | Test Loss: 9.7748 | Test
Acc: 10.03%
Epoch 20/50 | Train Loss: 0.0214 | Train Acc: 100.00% | Test Loss: 9.8300 | Test
Acc: 10.11%
Epoch 21/50 | Train Loss: 0.0184 | Train Acc: 100.00% | Test Loss: 9.8836 | Test
Acc: 10.28%
Epoch 22/50 | Train Loss: 0.0158 | Train Acc: 100.00% | Test Loss: 9.9355 | Test
Acc: 10.52%
Epoch 23/50 | Train Loss: 0.0125 | Train Acc: 100.00% | Test Loss: 9.9615 | Test
Acc: 10.83%
Epoch 24/50 | Train Loss: 0.0096 | Train Acc: 100.00% | Test Loss: 9.9402 | Test
Acc: 11.42%
Epoch 25/50 | Train Loss: 0.0078 | Train Acc: 100.00% | Test Loss: 9.8624 | Test
Acc: 12.07%
Epoch 26/50 | Train Loss: 0.0069 | Train Acc: 100.00% | Test Loss: 9.7242 | Test
Acc: 12.88%
Epoch 27/50 | Train Loss: 0.0061 | Train Acc: 100.00% | Test Loss: 9.5265 | Test
Acc: 14.09%
Epoch 28/50 | Train Loss: 0.0053 | Train Acc: 100.00% | Test Loss: 9.2823 | Test
Acc: 15.75%
Epoch 29/50 | Train Loss: 0.0046 | Train Acc: 100.00% | Test Loss: 9.0083 | Test
Acc: 17.60%
Epoch 30/50 | Train Loss: 0.0039 | Train Acc: 100.00% | Test Loss: 8.7316 | Test
Acc: 19.72%
```

```
Epoch 31/50 | Train Loss: 0.0034 | Train Acc: 100.00% | Test Loss: 8.4709 | Test
Acc: 21.79%
Epoch 32/50 | Train Loss: 0.0029 | Train Acc: 100.00% | Test Loss: 8.2284 | Test
Acc: 23.72%
Epoch 33/50 | Train Loss: 0.0026 | Train Acc: 100.00% | Test Loss: 8.0041 | Test
Acc: 25.51%
Epoch 34/50 | Train Loss: 0.0023 | Train Acc: 100.00% | Test Loss: 7.7995 | Test
Acc: 27.39%
Epoch 35/50 | Train Loss: 0.0021 | Train Acc: 100.00% | Test Loss: 7.6129 | Test
Acc: 28.88%
Epoch 36/50 | Train Loss: 0.0019 | Train Acc: 100.00% | Test Loss: 7.4403 | Test
Acc: 30.42%
Epoch 37/50 | Train Loss: 0.0017 | Train Acc: 100.00% | Test Loss: 7.2763 | Test
Acc: 32.02%
Epoch 38/50 | Train Loss: 0.0016 | Train Acc: 100.00% | Test Loss: 7.1159 | Test
Acc: 33.33%
Epoch 39/50 | Train Loss: 0.0014 | Train Acc: 100.00% | Test Loss: 6.9563 | Test
Acc: 34.98%
Epoch 40/50 | Train Loss: 0.0013 | Train Acc: 100.00% | Test Loss: 6.7945 | Test
Acc: 36.55%
Epoch 41/50 | Train Loss: 0.0012 | Train Acc: 100.00% | Test Loss: 6.6290 | Test
Acc: 38.09%
Epoch 42/50 | Train Loss: 0.0011 | Train Acc: 100.00% | Test Loss: 6.4589 | Test
Acc: 39.92%
Epoch 43/50 | Train Loss: 0.0010 | Train Acc: 100.00% | Test Loss: 6.2846 | Test
Acc: 41.66%
Epoch 44/50 | Train Loss: 0.0010 | Train Acc: 100.00% | Test Loss: 6.1071 | Test
Acc: 43.28%
Epoch 45/50 | Train Loss: 0.0009 | Train Acc: 100.00% | Test Loss: 5.9268 | Test
Acc: 44.94%
Epoch 46/50 | Train Loss: 0.0009 | Train Acc: 100.00% | Test Loss: 5.7442 | Test
Acc: 46.39%
Epoch 47/50 | Train Loss: 0.0008 | Train Acc: 100.00% | Test Loss: 5.5602 | Test
Acc: 47.79%
Epoch 48/50 | Train Loss: 0.0008 | Train Acc: 100.00% | Test Loss: 5.3756 | Test
Acc: 49.47%
Epoch 49/50 | Train Loss: 0.0007 | Train Acc: 100.00% | Test Loss: 5.1915 | Test
Acc: 51.17%
Epoch 50/50 | Train Loss: 0.0007 | Train Acc: 100.00% | Test Loss: 5.0089 | Test
Acc: 52.77%
```

In [14]:
```python
epoch_range = range(1, epochs + 1)
plt.figure(figsize=(6, 4))
plt.plot(epoch_range, train_loss1, label="Train Loss")
plt.plot(epoch_range, test_loss1, label="Test Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss (from scratch)")
plt.legend()
plt.tight_layout()
plt.show()
```



(2b) Train only last layer of pre-trained model

Create a new CNNModel, load pre-trained weights, fix the weights of all other layers and only train the last layer.

Remember to implement what happens when **freeze_layers = True** in the given **train_model()** function.

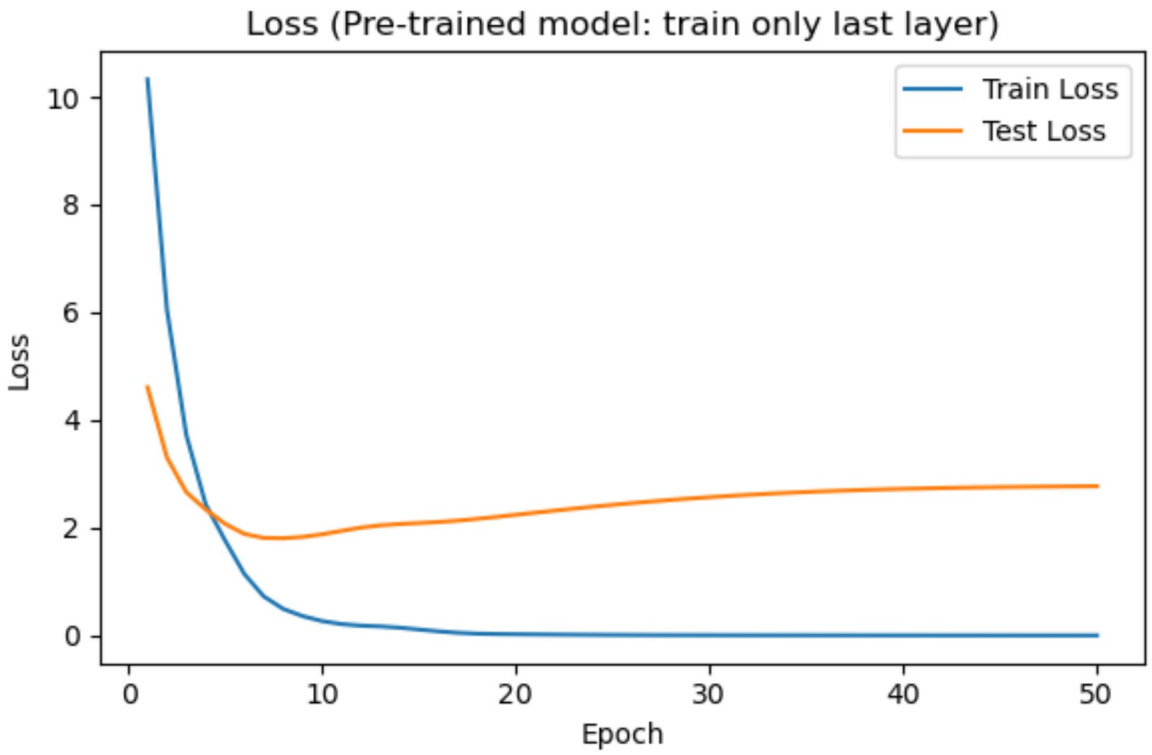
In [15]:
```
# Transfer Learning: Freeze all but last layer
model2 = CNNModel().to(device)
model2.load_state_dict(torch.load("pretrained_MNIST_model.pt"))
optimizer2 = torch.optim.Adam(model2.parameters(), lr=0.01)
epochs = 50
freeze_layers = True
train_loss2, test_loss2, train_acc2, test_acc2 = train_model(model2, train_loade
                                                             criterion, epochs=e
```

```
Epoch  1/50 | Train Loss: 10.3249 | Train Acc: 13.28% | Test Loss: 4.6077 | Test Acc: 22.53%
Epoch  2/50 | Train Loss: 6.0540 | Train Acc: 25.78% | Test Loss: 3.3054 | Test Acc: 29.61%
Epoch  3/50 | Train Loss: 3.7254 | Train Acc: 39.06% | Test Loss: 2.6625 | Test Acc: 38.05%
Epoch  4/50 | Train Loss: 2.4371 | Train Acc: 48.44% | Test Loss: 2.3402 | Test Acc: 44.78%
Epoch  5/50 | Train Loss: 1.7668 | Train Acc: 56.25% | Test Loss: 2.0748 | Test Acc: 50.14%
Epoch  6/50 | Train Loss: 1.1403 | Train Acc: 64.06% | Test Loss: 1.8857 | Test Acc: 56.47%
Epoch  7/50 | Train Loss: 0.7263 | Train Acc: 80.47% | Test Loss: 1.8122 | Test Acc: 59.25%
Epoch  8/50 | Train Loss: 0.4949 | Train Acc: 85.16% | Test Loss: 1.8067 | Test Acc: 60.20%
Epoch  9/50 | Train Loss: 0.3610 | Train Acc: 89.06% | Test Loss: 1.8307 | Test Acc: 61.05%
Epoch 10/50 | Train Loss: 0.2688 | Train Acc: 91.41% | Test Loss: 1.8787 | Test Acc: 62.08%
Epoch 11/50 | Train Loss: 0.2134 | Train Acc: 93.75% | Test Loss: 1.9418 | Test Acc: 62.64%
Epoch 12/50 | Train Loss: 0.1839 | Train Acc: 96.09% | Test Loss: 2.0022 | Test Acc: 63.16%
Epoch 13/50 | Train Loss: 0.1699 | Train Acc: 96.09% | Test Loss: 2.0448 | Test Acc: 63.69%
Epoch 14/50 | Train Loss: 0.1459 | Train Acc: 96.09% | Test Loss: 2.0701 | Test Acc: 64.20%
Epoch 15/50 | Train Loss: 0.1120 | Train Acc: 96.88% | Test Loss: 2.0878 | Test Acc: 64.79%
Epoch 16/50 | Train Loss: 0.0784 | Train Acc: 98.44% | Test Loss: 2.1071 | Test Acc: 65.38%
Epoch 17/50 | Train Loss: 0.0511 | Train Acc: 98.44% | Test Loss: 2.1326 | Test Acc: 65.74%
Epoch 18/50 | Train Loss: 0.0348 | Train Acc: 100.00% | Test Loss: 2.1650 | Test Acc: 66.07%
Epoch 19/50 | Train Loss: 0.0283 | Train Acc: 100.00% | Test Loss: 2.2020 | Test Acc: 66.33%
Epoch 20/50 | Train Loss: 0.0246 | Train Acc: 100.00% | Test Loss: 2.2409 | Test Acc: 66.48%
Epoch 21/50 | Train Loss: 0.0215 | Train Acc: 100.00% | Test Loss: 2.2800 | Test Acc: 66.51%
Epoch 22/50 | Train Loss: 0.0185 | Train Acc: 100.00% | Test Loss: 2.3184 | Test Acc: 66.62%
Epoch 23/50 | Train Loss: 0.0157 | Train Acc: 100.00% | Test Loss: 2.3556 | Test Acc: 66.67%
Epoch 24/50 | Train Loss: 0.0132 | Train Acc: 100.00% | Test Loss: 2.3913 | Test Acc: 66.76%
Epoch 25/50 | Train Loss: 0.0112 | Train Acc: 100.00% | Test Loss: 2.4254 | Test Acc: 66.77%
Epoch 26/50 | Train Loss: 0.0096 | Train Acc: 100.00% | Test Loss: 2.4578 | Test Acc: 66.76%
Epoch 27/50 | Train Loss: 0.0085 | Train Acc: 100.00% | Test Loss: 2.4883 | Test Acc: 66.68%
Epoch 28/50 | Train Loss: 0.0076 | Train Acc: 100.00% | Test Loss: 2.5169 | Test Acc: 66.65%
Epoch 29/50 | Train Loss: 0.0069 | Train Acc: 100.00% | Test Loss: 2.5436 | Test Acc: 66.58%
Epoch 30/50 | Train Loss: 0.0063 | Train Acc: 100.00% | Test Loss: 2.5683 | Test Acc: 66.48%
```

```
Epoch 31/50 | Train Loss: 0.0059 | Train Acc: 100.00% | Test Loss: 2.5911 | Test
Acc: 66.40%
Epoch 32/50 | Train Loss: 0.0055 | Train Acc: 100.00% | Test Loss: 2.6121 | Test
Acc: 66.39%
Epoch 33/50 | Train Loss: 0.0052 | Train Acc: 100.00% | Test Loss: 2.6312 | Test
Acc: 66.40%
Epoch 34/50 | Train Loss: 0.0049 | Train Acc: 100.00% | Test Loss: 2.6487 | Test
Acc: 66.40%
Epoch 35/50 | Train Loss: 0.0046 | Train Acc: 100.00% | Test Loss: 2.6646 | Test
Acc: 66.31%
Epoch 36/50 | Train Loss: 0.0044 | Train Acc: 100.00% | Test Loss: 2.6789 | Test
Acc: 66.28%
Epoch 37/50 | Train Loss: 0.0041 | Train Acc: 100.00% | Test Loss: 2.6918 | Test
Acc: 66.32%
Epoch 38/50 | Train Loss: 0.0039 | Train Acc: 100.00% | Test Loss: 2.7033 | Test
Acc: 66.31%
Epoch 39/50 | Train Loss: 0.0037 | Train Acc: 100.00% | Test Loss: 2.7137 | Test
Acc: 66.26%
Epoch 40/50 | Train Loss: 0.0035 | Train Acc: 100.00% | Test Loss: 2.7229 | Test
Acc: 66.26%
Epoch 41/50 | Train Loss: 0.0033 | Train Acc: 100.00% | Test Loss: 2.7310 | Test
Acc: 66.21%
Epoch 42/50 | Train Loss: 0.0032 | Train Acc: 100.00% | Test Loss: 2.7383 | Test
Acc: 66.24%
Epoch 43/50 | Train Loss: 0.0030 | Train Acc: 100.00% | Test Loss: 2.7447 | Test
Acc: 66.20%
Epoch 44/50 | Train Loss: 0.0029 | Train Acc: 100.00% | Test Loss: 2.7503 | Test
Acc: 66.29%
Epoch 45/50 | Train Loss: 0.0027 | Train Acc: 100.00% | Test Loss: 2.7552 | Test
Acc: 66.31%
Epoch 46/50 | Train Loss: 0.0026 | Train Acc: 100.00% | Test Loss: 2.7596 | Test
Acc: 66.27%
Epoch 47/50 | Train Loss: 0.0025 | Train Acc: 100.00% | Test Loss: 2.7634 | Test
Acc: 66.25%
Epoch 48/50 | Train Loss: 0.0024 | Train Acc: 100.00% | Test Loss: 2.7667 | Test
Acc: 66.28%
Epoch 49/50 | Train Loss: 0.0023 | Train Acc: 100.00% | Test Loss: 2.7697 | Test
Acc: 66.28%
Epoch 50/50 | Train Loss: 0.0022 | Train Acc: 100.00% | Test Loss: 2.7723 | Test
Acc: 66.30%
```

In [16]:

```python
epoch_range = range(1, epochs + 1)
plt.figure(figsize=(6, 4))
plt.plot(epoch_range, train_loss2, label="Train Loss")
plt.plot(epoch_range, test_loss2, label="Test Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss (Pre-trained model: train only last layer)")
plt.legend()
plt.tight_layout()
plt.show()
```



(2c) Train all layers of pre-trained model

Create a new CNNModel, load pre-trained weights and fine-tune all weights.

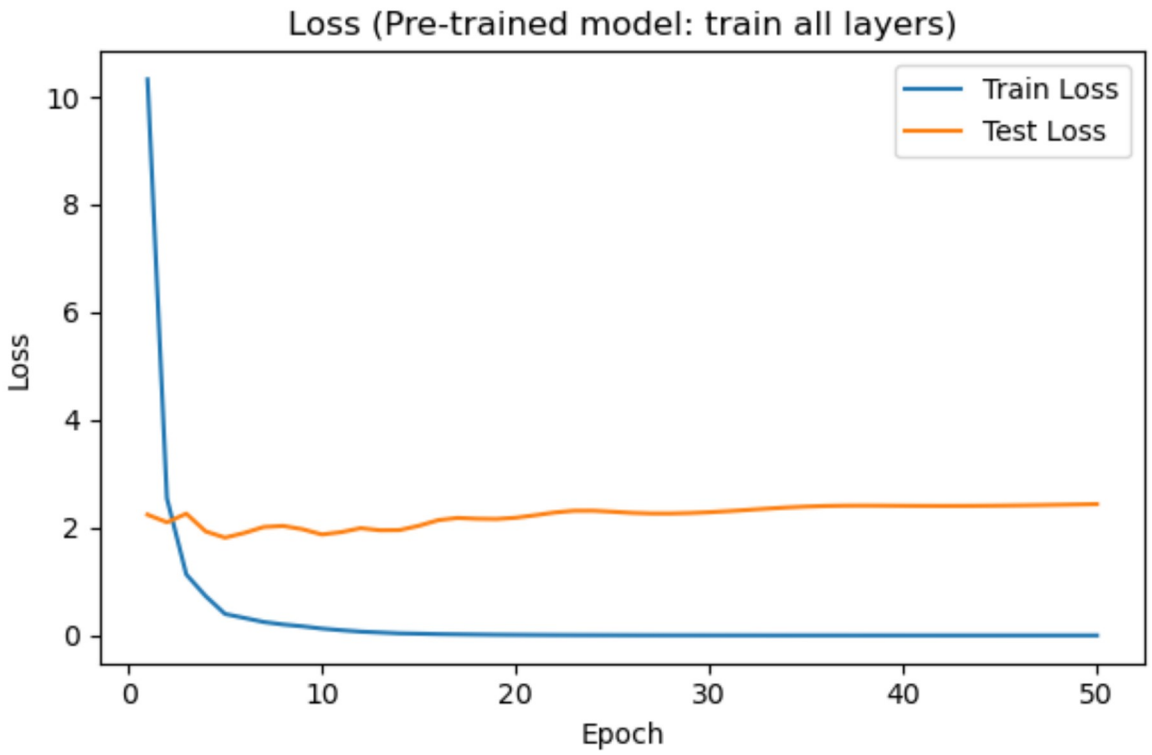
In [17]:
```
model3 = CNNModel().to(device)
model3.load_state_dict(torch.load("pretrained_MNIST_model.pt"))
optimizer3 = torch.optim.Adam(model3.parameters(), lr=0.01)
epochs = 50
freeze_layers = False
train_loss3, test_loss3, train_acc3, test_acc3 = train_model(model3, train_loade
                                                             criterion, epochs=e
```

```
Epoch  1/50 | Train Loss: 10.3249 | Train Acc: 13.28% | Test Loss: 2.2428 | Test Acc: 29.69%
Epoch  2/50 | Train Loss: 2.5462 | Train Acc: 40.62% | Test Loss: 2.1012 | Test Acc: 42.66%
Epoch  3/50 | Train Loss: 1.1327 | Train Acc: 64.06% | Test Loss: 2.2611 | Test Acc: 45.04%
Epoch  4/50 | Train Loss: 0.7305 | Train Acc: 74.22% | Test Loss: 1.9319 | Test Acc: 57.72%
Epoch  5/50 | Train Loss: 0.3988 | Train Acc: 82.81% | Test Loss: 1.8156 | Test Acc: 61.70%
Epoch  6/50 | Train Loss: 0.3244 | Train Acc: 88.28% | Test Loss: 1.9035 | Test Acc: 58.75%
Epoch  7/50 | Train Loss: 0.2493 | Train Acc: 89.84% | Test Loss: 2.0141 | Test Acc: 59.25%
Epoch  8/50 | Train Loss: 0.2035 | Train Acc: 93.75% | Test Loss: 2.0337 | Test Acc: 62.38%
Epoch  9/50 | Train Loss: 0.1694 | Train Acc: 95.31% | Test Loss: 1.9722 | Test Acc: 63.41%
Epoch 10/50 | Train Loss: 0.1272 | Train Acc: 97.66% | Test Loss: 1.8748 | Test Acc: 64.88%
Epoch 11/50 | Train Loss: 0.0971 | Train Acc: 97.66% | Test Loss: 1.9184 | Test Acc: 64.72%
Epoch 12/50 | Train Loss: 0.0707 | Train Acc: 99.22% | Test Loss: 1.9923 | Test Acc: 64.54%
Epoch 13/50 | Train Loss: 0.0547 | Train Acc: 100.00% | Test Loss: 1.9529 | Test Acc: 65.03%
Epoch 14/50 | Train Loss: 0.0387 | Train Acc: 100.00% | Test Loss: 1.9560 | Test Acc: 65.47%
Epoch 15/50 | Train Loss: 0.0321 | Train Acc: 100.00% | Test Loss: 2.0361 | Test Acc: 64.93%
Epoch 16/50 | Train Loss: 0.0245 | Train Acc: 100.00% | Test Loss: 2.1401 | Test Acc: 64.27%
Epoch 17/50 | Train Loss: 0.0206 | Train Acc: 100.00% | Test Loss: 2.1820 | Test Acc: 64.27%
Epoch 18/50 | Train Loss: 0.0168 | Train Acc: 100.00% | Test Loss: 2.1666 | Test Acc: 64.60%
Epoch 19/50 | Train Loss: 0.0132 | Train Acc: 100.00% | Test Loss: 2.1617 | Test Acc: 64.90%
Epoch 20/50 | Train Loss: 0.0110 | Train Acc: 100.00% | Test Loss: 2.1856 | Test Acc: 64.82%
Epoch 21/50 | Train Loss: 0.0090 | Train Acc: 100.00% | Test Loss: 2.2316 | Test Acc: 64.76%
Epoch 22/50 | Train Loss: 0.0072 | Train Acc: 100.00% | Test Loss: 2.2824 | Test Acc: 64.69%
Epoch 23/50 | Train Loss: 0.0061 | Train Acc: 100.00% | Test Loss: 2.3147 | Test Acc: 64.61%
Epoch 24/50 | Train Loss: 0.0054 | Train Acc: 100.00% | Test Loss: 2.3158 | Test Acc: 64.81%
Epoch 25/50 | Train Loss: 0.0046 | Train Acc: 100.00% | Test Loss: 2.2965 | Test Acc: 65.21%
Epoch 26/50 | Train Loss: 0.0038 | Train Acc: 100.00% | Test Loss: 2.2760 | Test Acc: 65.69%
Epoch 27/50 | Train Loss: 0.0032 | Train Acc: 100.00% | Test Loss: 2.2648 | Test Acc: 66.11%
Epoch 28/50 | Train Loss: 0.0028 | Train Acc: 100.00% | Test Loss: 2.2643 | Test Acc: 66.26%
Epoch 29/50 | Train Loss: 0.0025 | Train Acc: 100.00% | Test Loss: 2.2728 | Test Acc: 66.34%
Epoch 30/50 | Train Loss: 0.0022 | Train Acc: 100.00% | Test Loss: 2.2880 | Test Acc: 66.45%
```

```
Epoch 31/50 | Train Loss: 0.0019 | Train Acc: 100.00% | Test Loss: 2.3081 | Test
Acc: 66.33%
Epoch 32/50 | Train Loss: 0.0016 | Train Acc: 100.00% | Test Loss: 2.3312 | Test
Acc: 66.20%
Epoch 33/50 | Train Loss: 0.0014 | Train Acc: 100.00% | Test Loss: 2.3549 | Test
Acc: 66.01%
Epoch 34/50 | Train Loss: 0.0012 | Train Acc: 100.00% | Test Loss: 2.3762 | Test
Acc: 66.01%
Epoch 35/50 | Train Loss: 0.0011 | Train Acc: 100.00% | Test Loss: 2.3930 | Test
Acc: 65.99%
Epoch 36/50 | Train Loss: 0.0010 | Train Acc: 100.00% | Test Loss: 2.4043 | Test
Acc: 66.01%
Epoch 37/50 | Train Loss: 0.0009 | Train Acc: 100.00% | Test Loss: 2.4099 | Test
Acc: 66.07%
Epoch 38/50 | Train Loss: 0.0009 | Train Acc: 100.00% | Test Loss: 2.4112 | Test
Acc: 66.12%
Epoch 39/50 | Train Loss: 0.0008 | Train Acc: 100.00% | Test Loss: 2.4097 | Test
Acc: 66.22%
Epoch 40/50 | Train Loss: 0.0007 | Train Acc: 100.00% | Test Loss: 2.4075 | Test
Acc: 66.29%
Epoch 41/50 | Train Loss: 0.0006 | Train Acc: 100.00% | Test Loss: 2.4056 | Test
Acc: 66.42%
Epoch 42/50 | Train Loss: 0.0006 | Train Acc: 100.00% | Test Loss: 2.4048 | Test
Acc: 66.38%
Epoch 43/50 | Train Loss: 0.0005 | Train Acc: 100.00% | Test Loss: 2.4053 | Test
Acc: 66.41%
Epoch 44/50 | Train Loss: 0.0005 | Train Acc: 100.00% | Test Loss: 2.4072 | Test
Acc: 66.39%
Epoch 45/50 | Train Loss: 0.0005 | Train Acc: 100.00% | Test Loss: 2.4104 | Test
Acc: 66.48%
Epoch 46/50 | Train Loss: 0.0005 | Train Acc: 100.00% | Test Loss: 2.4146 | Test
Acc: 66.49%
Epoch 47/50 | Train Loss: 0.0004 | Train Acc: 100.00% | Test Loss: 2.4198 | Test
Acc: 66.63%
Epoch 48/50 | Train Loss: 0.0004 | Train Acc: 100.00% | Test Loss: 2.4256 | Test
Acc: 66.65%
Epoch 49/50 | Train Loss: 0.0004 | Train Acc: 100.00% | Test Loss: 2.4320 | Test
Acc: 66.60%
Epoch 50/50 | Train Loss: 0.0004 | Train Acc: 100.00% | Test Loss: 2.4388 | Test
Acc: 66.63%
```

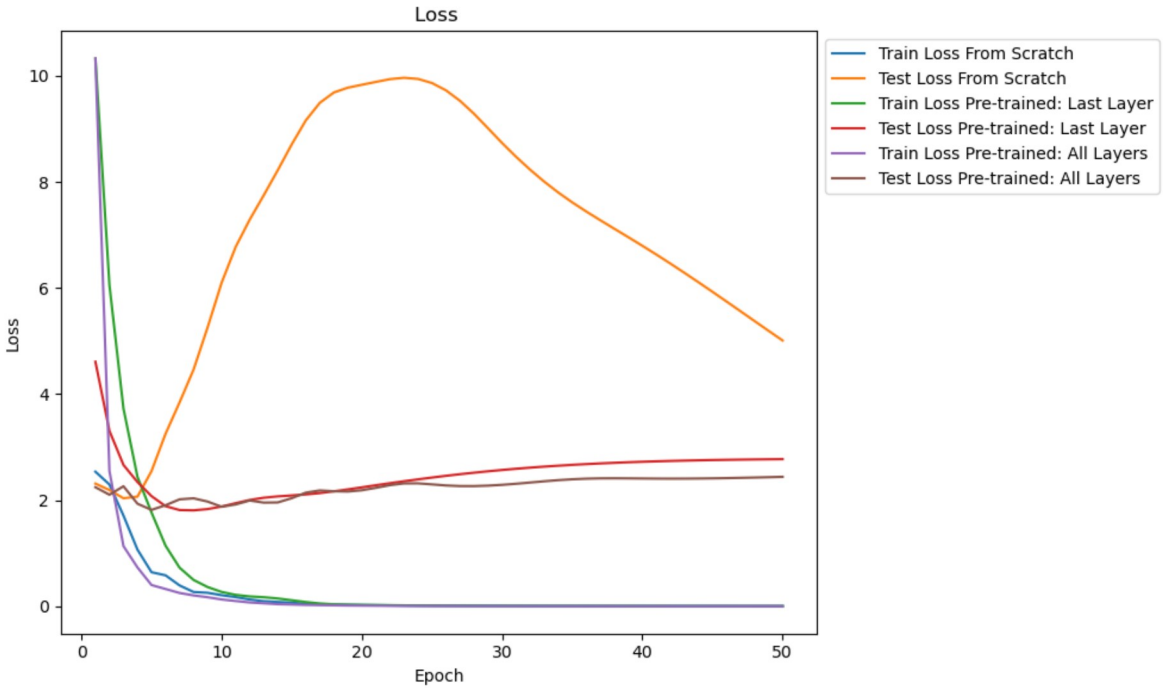
In [18]:

```python
epoch_range = range(1, epochs + 1)
plt.figure(figsize=(6, 4))
plt.plot(epoch_range, train_loss3, label="Train Loss")
plt.plot(epoch_range, test_loss3, label="Test Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss (Pre-trained model: train all layers)")
plt.legend()
plt.tight_layout()
plt.show()
```



(2d) Plot and Report

In [19]:
```
epoch_range = range(1, epochs + 1)
plt.figure(figsize=(10, 6))
for train, test, label in zip([train_loss1, train_loss2, train_loss3],
                              [test_loss1, test_loss2, test_loss3],
                              ["From Scratch", "Pre-trained: Last Layer", "Pre-trained:
    plt.plot(epoch_range, train, label=f"Train Loss {label}")
    plt.plot(epoch_range, test, label=f"Test Loss {label}")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss ")
plt.legend(loc="upper left", bbox_to_anchor=(1,1))
plt.tight_layout()
plt.show()
```



From scratch:

- Train Loss: 0.0007 | Train Acc: 100.00% | Test Loss: 5.0089 | Test Acc: 52.77%

Train only last layer:

- Train Loss: 0.0022 | Train Acc: 100.00% | Test Loss: 2.7723 | Test Acc: 66.30%

Train all layers:

- Train Loss: 0.0004 | Train Acc: 100.00% | Test Loss: 2.4388 | Test Acc: 66.63%

Both methods of transfer learning are superior to training from scratch, in terms of accuracy and computational time. Training from scratch could probably reach similar test accuracy on this data, but it just takes a lot longer. However, between the two transfer learning methods the difference is quite minimal in this case. Both converge to around the same test accuracy in roughly the same amount of epochs. Training only the last layer is perhaps slightly faster, giving it a slight edge.

3. Few-shot learning

Here **KShotCDataset()** function is given.

It is used to create a FashionMNIST dataloader for a given **K** and **C**.

DO NOT CHANGE THIS

In [20]:
```
from torch.utils.data import Dataset

# Set random seed for reproducibility
seed = 42
random.seed(seed)
torch.manual_seed(seed)
```

```
class KShotCDataset(Dataset):
    def __init__(self, fashion_mnist_dataset, k_shot, c_way):
        self.fashion_mnist_dataset = fashion_mnist_dataset
        self.k_shot = k_shot
        self.c_way = c_way

        self.data_indices = []

        self.class_indices = {label: [] for label in range(self.c_way)}
        self.create_balanced_dataset()

    def create_balanced_dataset(self):
        for idx, (_, label) in enumerate(self.fashion_mnist_dataset):
            if label < self.c_way:
                self.class_indices[label].append(idx)

        for label in range(self.c_way):
            self.data_indices.extend(self.class_indices[label][:self.k_shot])

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

    def __getitem__(self, index):
        fashion_mnist_index = self.data_indices[index]
        image, label = self.fashion_mnist_dataset[fashion_mnist_index]
        return image, label

# Load the Fashion MNIST training dataset
transform = transforms.Compose([transforms.ToTensor()])
fashionmnist_dataset = torchvision.datasets.FashionMNIST(
    root='./data',
    train=True,
    transform=transform,
    download=True
)
```

Below, you see an example of how to use **KShotCDataset().**

Implement the algorithm and follow the exercise sheet for details on what to analyse and report.

In [ ]:
```
K = 7  # Number of shots per class
C = 10  # Number of classes -- ordered labels are selected, e.g. C = 3 means lab
# Create the K-shot C-way dataset
k_shot_c_dataset = KShotCDataset(fashionmnist_dataset, K, C)

dataloader = DataLoader(k_shot_c_dataset, batch_size=K*C, shuffle=True)

model = CNNModel().to(device)
model.load_state_dict(torch.load("pretrained_MNIST_model.pt"))
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
epochs = 50
freeze_layers = True
train_loss, test_loss, train_acc, test_acc = train_model(model, train_loader, te
                                                         criterion, epochs=e
for images, labels in dataloader:
    # Use the model for feature extraction ...
```

```
    # ...
    pass
```