

```
In [23]: import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

seed = 14
torch.manual_seed(seed)
np.random.seed(seed)

# Transformacja do tensora (niedbana dla sieci)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Pobranie danych treningowych i testowych
train_dataset = torchvision.datasets.FashionMNIST("./data", train=True, download=True)
test_dataset = torchvision.datasets.FashionMNIST("./data", train=False, download=True)

# Dataloadery
batch_size = 16
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size)

# Sprawdzenie kształtu danych
for X, y in train_loader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break
```

```
Shape of X [N, C, H, W]: torch.Size([16, 1, 28, 28])
Shape of y: torch.Size([16]) torch.int64
```

```
In [24]: class NeuralNetwork(nn.Module):
    def __init__(self, input_size=28*28, hidden_size=512, num_classes=10, num_hidden_layers=0):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()

        layers = []
        if num_hidden_layers == 0:
            # Sieć jednowarstwowa (tylko warstwa wyjściowa)
            layers.append(nn.Linear(input_size, num_classes))
        else:
            # Sieć dwuwarstwowa (warstwa ukryta + wyjściowa)
            layers.append(nn.Linear(input_size, hidden_size))
            layers.append(nn.ReLU())
            layers.append(nn.Linear(hidden_size, num_classes))

        self.linear_relu_stack = nn.Sequential(*layers)

    def forward(self, x):
        x = self.flatten(x)
```

```

        logits = self.linear_relu_stack(x)
        return logits

device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")

```

Using cpu device

```

In [25]: def train_model(model, train_loader, criterion, optimizer, epochs=5):
    model.train()
    loss_history = []

    for epoch in range(epochs):
        running_loss = 0.0
        for batch, (X, y) in enumerate(train_loader):
            X, y = X.to(device), y.to(device)

            # Obliczenie błędu
            pred = model(X)
            loss = criterion(pred, y)

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

            running_loss += loss.item()

        avg_loss = running_loss / len(train_loader)
        loss_history.append(avg_loss)
        print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.4f}")

    return loss_history

def evaluate_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for X, y in test_loader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            predicted = pred.argmax(1)
            total += y.size(0)
            correct += (predicted == y).sum().item()

    accuracy = correct / total
    return accuracy

```

```

In [26]: # Eksperyment 1: Liczba neuronów w warstwie ukrytej
hidden_sizes = [4, 16, 32, 128, 512]
results_exp1 = {}

print("--- Eksperyment 1: Liczba neuronów ---")

# 1. Sieć jednowarstwowa (0 warstw ukrytych)

```

```

print(f"Training 1-layer network (Linear)...")
model = NeuralNetwork(num_hidden_layers=0).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_hist = train_model(model, train_loader, criterion, optimizer, epochs=5)
acc = evaluate_model(model, test_loader)
results_exp1['1-layer'] = {'loss': loss_hist, 'acc': acc}
print(f"1-layer Accuracy: {acc:.4f}")

# 2. Sieć dwuwarstwowa z różną liczbą neuronów
for h_size in hidden_sizes:
    print(f"Training 2-layer network with {h_size} hidden neurons...")
    model = NeuralNetwork(hidden_size=h_size, num_hidden_layers=1).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    loss_hist = train_model(model, train_loader, criterion, optimizer, epochs=5)
    acc = evaluate_model(model, test_loader)
    results_exp1[f'2-layer ({h_size})'] = {'loss': loss_hist, 'acc': acc}
    print(f"2-layer ({h_size}) Accuracy: {acc:.4f}")

# Wizualizacja
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
for name, res in results_exp1.items():
    plt.plot(res['loss'], label=name)
plt.title("Learning Curves (Loss)")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.ylim(bottom=0)
plt.legend()

plt.subplot(1, 2, 2)
names = list(results_exp1.keys())
accs = [results_exp1[n]['acc'] for n in names]
plt.bar(names, accs)
plt.title("Test Accuracy")
plt.ylabel("Accuracy")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

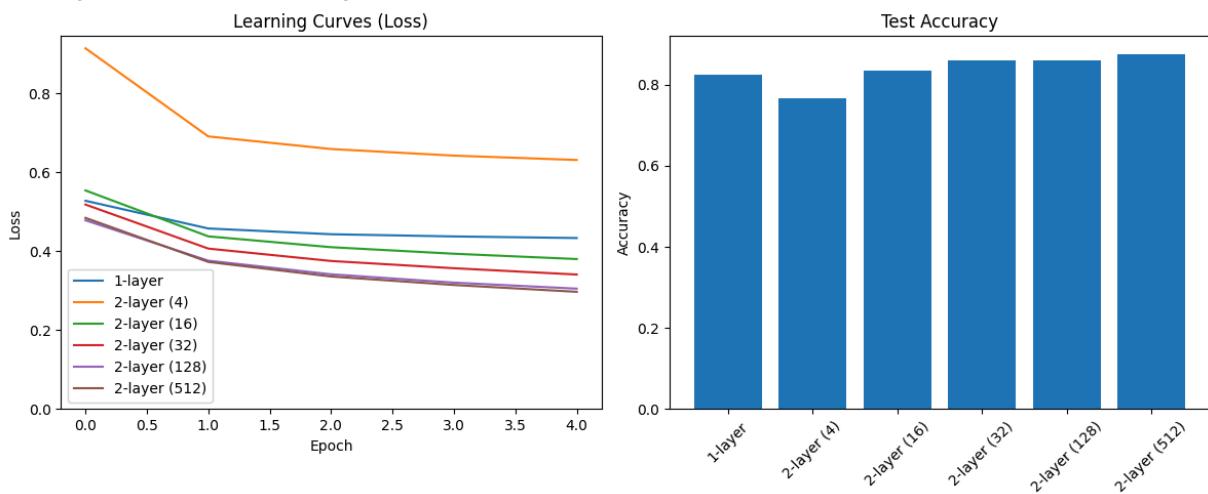
```

--- Eksperyment 1: Liczba neuronów ---  
Training 1-layer network (Linear)...  
Epoch 1/5, Loss: 0.5278  
Epoch 1/5, Loss: 0.5278  
Epoch 2/5, Loss: 0.4576  
Epoch 2/5, Loss: 0.4576  
Epoch 3/5, Loss: 0.4430  
Epoch 3/5, Loss: 0.4430  
Epoch 4/5, Loss: 0.4375  
Epoch 4/5, Loss: 0.4375  
Epoch 5/5, Loss: 0.4335  
Epoch 5/5, Loss: 0.4335  
1-layer Accuracy: 0.8253  
Training 2-layer network with 4 hidden neurons...  
1-layer Accuracy: 0.8253  
Training 2-layer network with 4 hidden neurons...  
Epoch 1/5, Loss: 0.9139  
Epoch 1/5, Loss: 0.9139  
Epoch 2/5, Loss: 0.6908  
Epoch 2/5, Loss: 0.6908  
Epoch 3/5, Loss: 0.6589  
Epoch 3/5, Loss: 0.6589  
Epoch 4/5, Loss: 0.6421  
Epoch 4/5, Loss: 0.6421  
Epoch 5/5, Loss: 0.6311  
Epoch 5/5, Loss: 0.6311  
2-layer (4) Accuracy: 0.7656  
Training 2-layer network with 16 hidden neurons...  
2-layer (4) Accuracy: 0.7656  
Training 2-layer network with 16 hidden neurons...  
Epoch 1/5, Loss: 0.5540  
Epoch 1/5, Loss: 0.5540  
Epoch 2/5, Loss: 0.4377  
Epoch 2/5, Loss: 0.4377  
Epoch 3/5, Loss: 0.4103  
Epoch 3/5, Loss: 0.4103  
Epoch 4/5, Loss: 0.3937  
Epoch 4/5, Loss: 0.3937  
Epoch 5/5, Loss: 0.3802  
Epoch 5/5, Loss: 0.3802  
2-layer (16) Accuracy: 0.8336  
Training 2-layer network with 32 hidden neurons...  
2-layer (16) Accuracy: 0.8336  
Training 2-layer network with 32 hidden neurons...  
Epoch 1/5, Loss: 0.5182  
Epoch 1/5, Loss: 0.5182  
Epoch 2/5, Loss: 0.4067  
Epoch 2/5, Loss: 0.4067  
Epoch 3/5, Loss: 0.3753  
Epoch 3/5, Loss: 0.3753  
Epoch 4/5, Loss: 0.3570  
Epoch 4/5, Loss: 0.3570  
Epoch 5/5, Loss: 0.3410  
Epoch 5/5, Loss: 0.3410  
2-layer (32) Accuracy: 0.8597  
Training 2-layer network with 128 hidden neurons...

```

2-layer (32) Accuracy: 0.8597
Training 2-layer network with 128 hidden neurons...
Epoch 1/5, Loss: 0.4783
Epoch 1/5, Loss: 0.4783
Epoch 2/5, Loss: 0.3761
Epoch 2/5, Loss: 0.3761
Epoch 3/5, Loss: 0.3416
Epoch 3/5, Loss: 0.3416
Epoch 4/5, Loss: 0.3205
Epoch 4/5, Loss: 0.3205
Epoch 5/5, Loss: 0.3052
Epoch 5/5, Loss: 0.3052
2-layer (128) Accuracy: 0.8606
Training 2-layer network with 512 hidden neurons...
2-layer (128) Accuracy: 0.8606
Training 2-layer network with 512 hidden neurons...
Epoch 1/5, Loss: 0.4845
Epoch 1/5, Loss: 0.4845
Epoch 2/5, Loss: 0.3729
Epoch 2/5, Loss: 0.3729
Epoch 3/5, Loss: 0.3360
Epoch 3/5, Loss: 0.3360
Epoch 4/5, Loss: 0.3146
Epoch 4/5, Loss: 0.3146
Epoch 5/5, Loss: 0.2972
Epoch 5/5, Loss: 0.2972
2-layer (512) Accuracy: 0.8755
2-layer (512) Accuracy: 0.8755

```



## Wnioski - Eksperyment 1

- **Sieć jednowarstwowa (liniowa)** osiąga przyzwoity wynik, ale zazwyczaj gorszy od sieci głębszych, ponieważ nie potrafi modelować nieliniowych zależności.
- **Mała liczba neuronów (np. 4)** w warstwie ukrytej działa jak "wąskie gardło" (bottleneck), ograniczając przepływ informacji i prowadząc do underfittingu.
- **Zwiększenie liczby neuronów** (do 128, 512) zazwyczaj poprawia dokładność, pozwalając sieci nauczyć się bardziej złożonych cech. Jednak zbyt duża liczba

może prowadzić do overfittingu (choć przy 5 epokach może to nie być jeszcze widoczne) i wydłuża czas obliczeń.

```
In [27]: # Eksperyment 2: Rozmiar batcha
batch_sizes = [16, 64, 256]
results_exp2 = {}
fixed_hidden_size = 128 # Wybieramy jedną architekturę dla porównania

print("\n--- Eksperyment 2: Rozmiar batcha ---")

for b_size in batch_sizes:
    print(f"Training with batch size {b_size}...")
    # Musimy utworzyć nowy DataLoader dla każdego rozmiaru batcha
    train_loader_b = torch.utils.data.DataLoader(train_dataset, batch_size=b_size)

    model = NeuralNetwork(hidden_size=fixed_hidden_size, num_hidden_layers=1)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

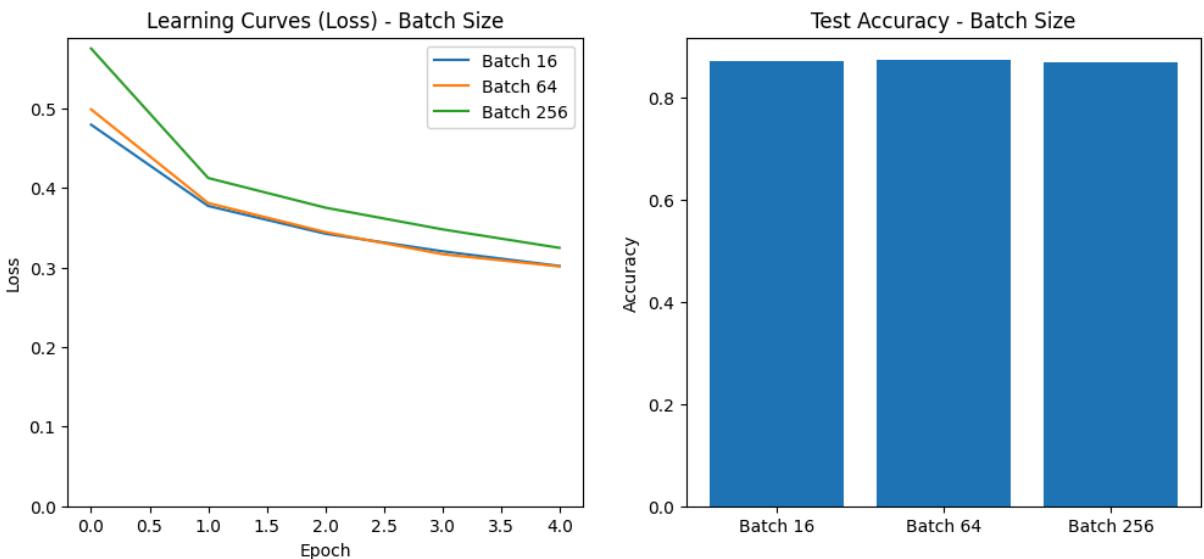
    loss_hist = train_model(model, train_loader_b, criterion, optimizer, epochs=5)
    acc = evaluate_model(model, test_loader)

    results_exp2[f'Batch {b_size}'] = {'loss': loss_hist, 'acc': acc}
    print(f"Batch {b_size} Accuracy: {acc:.4f}")

# Wizualizacja
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
for name, res in results_exp2.items():
    plt.plot(res['loss'], label=name)
plt.title("Learning Curves (Loss) – Batch Size")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.ylim(bottom=0)
plt.legend()

plt.subplot(1, 2, 2)
names = list(results_exp2.keys())
accs = [results_exp2[n]['acc'] for n in names]
plt.bar(names, accs)
plt.title("Test Accuracy – Batch Size")
plt.ylabel("Accuracy")
plt.show()
```

--- Eksperiment 2: Rozmiar batcha ---  
Training with batch size 16...  
Epoch 1/5, Loss: 0.4797  
Epoch 1/5, Loss: 0.4797  
Epoch 2/5, Loss: 0.3775  
Epoch 2/5, Loss: 0.3775  
Epoch 3/5, Loss: 0.3426  
Epoch 3/5, Loss: 0.3426  
Epoch 4/5, Loss: 0.3205  
Epoch 4/5, Loss: 0.3205  
Epoch 5/5, Loss: 0.3020  
Epoch 5/5, Loss: 0.3020  
Batch 16 Accuracy: 0.8711  
Training with batch size 64...  
Batch 16 Accuracy: 0.8711  
Training with batch size 64...  
Epoch 1/5, Loss: 0.4990  
Epoch 1/5, Loss: 0.4990  
Epoch 2/5, Loss: 0.3812  
Epoch 2/5, Loss: 0.3812  
Epoch 3/5, Loss: 0.3446  
Epoch 3/5, Loss: 0.3446  
Epoch 4/5, Loss: 0.3168  
Epoch 4/5, Loss: 0.3168  
Epoch 5/5, Loss: 0.3014  
Epoch 5/5, Loss: 0.3014  
Batch 64 Accuracy: 0.8743  
Training with batch size 256...  
Batch 64 Accuracy: 0.8743  
Training with batch size 256...  
Epoch 1/5, Loss: 0.5755  
Epoch 1/5, Loss: 0.5755  
Epoch 2/5, Loss: 0.4126  
Epoch 2/5, Loss: 0.4126  
Epoch 3/5, Loss: 0.3754  
Epoch 3/5, Loss: 0.3754  
Epoch 4/5, Loss: 0.3481  
Epoch 4/5, Loss: 0.3481  
Epoch 5/5, Loss: 0.3248  
Epoch 5/5, Loss: 0.3248  
Batch 256 Accuracy: 0.8696  
Batch 256 Accuracy: 0.8696



## Wnioski - Eksperyment 2

- **Mały batch (16):** Powoduje częstsze aktualizacje wag w jednej epoce, co często prowadzi do szybszego spadku funkcji straty na początku. Gradient jest bardziej zaszumiony, co może pomóc uciec z minimów lokalnych, ale trening trwa dłużej (więcej iteracji).
- **Duży batch (256):** Gradient jest dokładniejszym oszacowaniem prawdziwego gradientu, ale rzadsze aktualizacje mogą spowolnić proces uczenia w sensie liczby epok.
- Kompromis (np. 64) często daje najlepsze rezultaty pod względem czasu i stabilności.

```
In [28]: # Eksperyment 3: Rozmiar zbioru uczącego
dataset_fractions = [0.01, 0.1, 1.0]
results_exp3 = {}
fixed_hidden_size = 128
fixed_batch_size = 64

print("\n--- Eksperyment 3: Rozmiar zbioru uczącego ---")

for fraction in dataset_fractions:
    subset_size = int(len(train_dataset) * fraction)
    print(f"Training with {fraction*100}% data ({subset_size} samples)...")

# Tworzenie podzbioru
indices = np.random.choice(len(train_dataset), subset_size, replace=False)
subset = torch.utils.data.Subset(train_dataset, indices)
train_loader_subset = torch.utils.data.DataLoader(subset, batch_size=fix

model = NeuralNetwork(hidden_size=fixed_hidden_size, num_hidden_layers=1)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

loss_hist = train_model(model, train_loader_subset, criterion, optimizer)
```

```
acc = evaluate_model(model, test_loader)

results_exp3[f'{int(fraction*100)}% Data'] = {'loss': loss_hist, 'acc': acc}
print(f'{int(fraction*100)}% Data Accuracy: {acc:.4f}')

# Wizualizacja
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
for name, res in results_exp3.items():
    plt.plot(res['loss'], label=name)
plt.title("Learning Curves (Loss) - Dataset Size")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.ylim(bottom=0)
plt.legend()

plt.subplot(1, 2, 2)
names = list(results_exp3.keys())
accs = [results_exp3[n]['acc'] for n in names]
plt.bar(names, accs)
plt.title("Test Accuracy - Dataset Size")
plt.ylabel("Accuracy")
plt.show()
```

--- Eksperyment 3: Rozmiar zbioru uczącego ---

Training with 1.0% data (600 samples)...

Epoch 1/5, Loss: 1.7498  
 Epoch 2/5, Loss: 1.0008  
 Epoch 3/5, Loss: 0.7615  
 Epoch 4/5, Loss: 0.6459  
 Epoch 5/5, Loss: 0.5885  
 Epoch 5/5, Loss: 0.5885  
 1% Data Accuracy: 0.7422

Training with 10.0% data (6000 samples)...

1% Data Accuracy: 0.7422

Training with 10.0% data (6000 samples)...

Epoch 1/5, Loss: 0.8131  
 Epoch 1/5, Loss: 0.8131  
 Epoch 2/5, Loss: 0.5202  
 Epoch 2/5, Loss: 0.5202  
 Epoch 3/5, Loss: 0.4637  
 Epoch 3/5, Loss: 0.4637  
 Epoch 4/5, Loss: 0.4272  
 Epoch 4/5, Loss: 0.4272  
 Epoch 5/5, Loss: 0.3964  
 Epoch 5/5, Loss: 0.3964  
 10% Data Accuracy: 0.8142

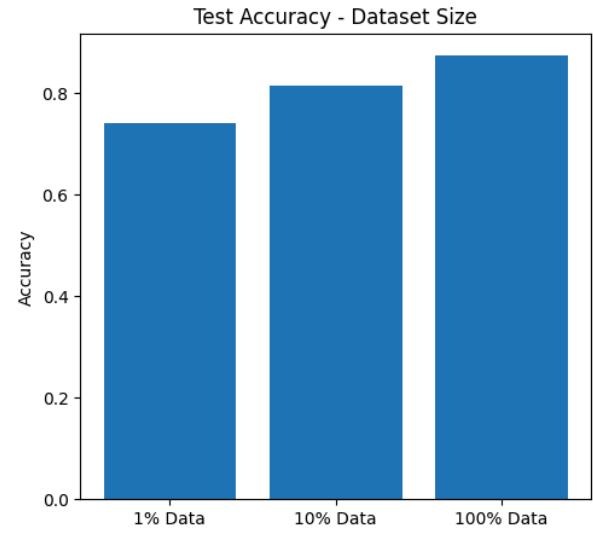
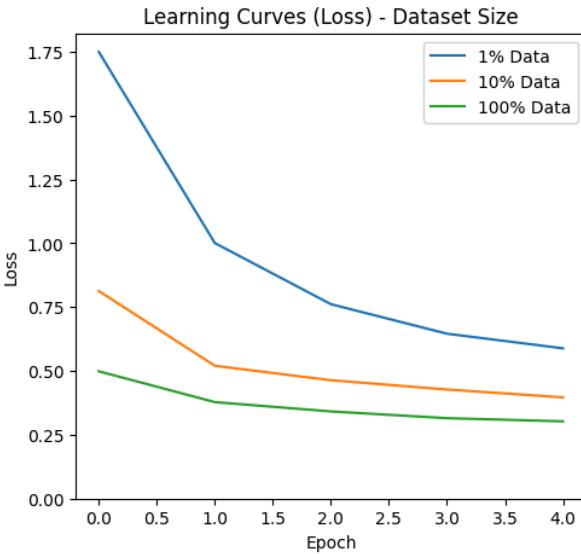
Training with 100.0% data (60000 samples)...

10% Data Accuracy: 0.8142

Training with 100.0% data (60000 samples)...

Epoch 1/5, Loss: 0.4986  
 Epoch 1/5, Loss: 0.4986  
 Epoch 2/5, Loss: 0.3778  
 Epoch 2/5, Loss: 0.3778  
 Epoch 3/5, Loss: 0.3415  
 Epoch 3/5, Loss: 0.3415  
 Epoch 4/5, Loss: 0.3151  
 Epoch 4/5, Loss: 0.3151  
 Epoch 5/5, Loss: 0.3027  
 Epoch 5/5, Loss: 0.3027  
 100% Data Accuracy: 0.8747

100% Data Accuracy: 0.8747



## Wnioski - Eksperyment 3

- **Ilość danych ma kluczowe znaczenie.** Model trenowany na 1% danych osiąga bardzo niską dokładność i prawdopodobnie szybko ulega przeuczeniu (overfitting) - zapamiętuje nieliczne przykłady zamiast uczyć się ogólnych reguł.
- Wraz ze wzrostem frakcji danych (10% -> 100%), dokładność na zbiorze testowym znaczco rośnie.
- Pokazuje to, że nawet najlepsza architektura nie pomoże, jeśli nie mamy wystarczającej ilości reprezentatywnych danych treningowych.

```
In [29]: # Eksperyment 4: Zaburzenia danych (Szum Gaussowski)
noise_std = 0.5
results_exp4 = {}

print("\n--- Eksperyment 4: Odporność na szum ---")

# Transformacja z szumem
# Dodajemy szum do tensora
transform_noisy = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)),
    transforms.Lambda(lambda x: x + torch.randn_like(x) * noise_std)
])

# Dataset z szumem (tylko do testów lub treningu w scenariuszu 2)
train_dataset_noisy = torchvision.datasets.FashionMNIST("./data", train=True)
test_dataset_noisy = torchvision.datasets.FashionMNIST("./data", train=False)

train_loader_noisy = torch.utils.data.DataLoader(train_dataset_noisy, batch_size=64, shuffle=True)
test_loader_noisy = torch.utils.data.DataLoader(test_dataset_noisy, batch_size=64, shuffle=False)

# Scenariusz 1: Trening na czystych danych, test na zaszumionych
print("Scenario 1: Train Clean, Test Noisy")
model_clean = NeuralNetwork(hidden_size=128, num_hidden_layers=1).to(device)
optimizer = optim.Adam(model_clean.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# Trenujemy na standardowym train_loader (czyste dane)
train_model(model_clean, train_loader, criterion, optimizer, epochs=5)
acc_clean_noisy = evaluate_model(model_clean, test_loader_noisy) # Testujemy
print(f"Train Clean, Test Noisy Accuracy: {acc_clean_noisy:.4f}")
results_exp4['Train Clean / Test Noisy'] = acc_clean_noisy

# Scenariusz 2: Trening na zaszumionych danych, test na zaszumionych
print("Scenario 2: Train Noisy, Test Noisy")
model_noisy = NeuralNetwork(hidden_size=128, num_hidden_layers=1).to(device)
optimizer = optim.Adam(model_noisy.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# Trenujemy na train_loader_noisy
train_model(model_noisy, train_loader_noisy, criterion, optimizer, epochs=5)
acc_noisy_noisy = evaluate_model(model_noisy, test_loader_noisy) # Testujemy
results_exp4['Train Noisy / Test Noisy'] = acc_noisy_noisy
```

```

print(f"Train Noisy, Test Noisy Accuracy: {acc_noisy_noisy:.4f}")
results_exp4['Train Noisy / Test Noisy'] = acc_noisy_noisy

# Scenariusz 3: Trening na zaszumionych danych, test na czystych
print("Scenario 3: Train Noisy, Test Clean")
acc_noisy_clean = evaluate_model(model_noisy, test_loader) # Testujemy na czystych
print(f"Train Noisy, Test Clean Accuracy: {acc_noisy_clean:.4f}")
results_exp4['Train Noisy / Test Clean'] = acc_noisy_clean

# Baseline: Train Clean, Test Clean (dla odniesienia)
acc_clean_clean = evaluate_model(model_clean, test_loader)
results_exp4['Train Clean / Test Clean'] = acc_clean_clean

# Wizualizacja
plt.figure(figsize=(10, 5))
plt.bar(results_exp4.keys(), results_exp4.values())
plt.title(f"Noise Robustness (std={noise_std})")
plt.ylabel("Accuracy")
plt.xticks(rotation=15)
plt.show()

```

--- Eksperyment 4: Odporność na szum ---

Scenario 1: Train Clean, Test Noisy

Epoch 1/5, Loss: 0.4809

Epoch 1/5, Loss: 0.4809

Epoch 2/5, Loss: 0.3760

Epoch 2/5, Loss: 0.3760

Epoch 3/5, Loss: 0.3402

Epoch 3/5, Loss: 0.3402

Epoch 4/5, Loss: 0.3189

Epoch 4/5, Loss: 0.3189

Epoch 5/5, Loss: 0.3025

Epoch 5/5, Loss: 0.3025

Train Clean, Test Noisy Accuracy: 0.8412

Scenario 2: Train Noisy, Test Noisy

Train Clean, Test Noisy Accuracy: 0.8412

Scenario 2: Train Noisy, Test Noisy

Epoch 1/5, Loss: 0.5431

Epoch 1/5, Loss: 0.5431

Epoch 2/5, Loss: 0.4402

Epoch 2/5, Loss: 0.4402

Epoch 3/5, Loss: 0.4062

Epoch 3/5, Loss: 0.4062

Epoch 4/5, Loss: 0.3867

Epoch 4/5, Loss: 0.3867

Epoch 5/5, Loss: 0.3708

Epoch 5/5, Loss: 0.3708

Train Noisy, Test Noisy Accuracy: 0.8432

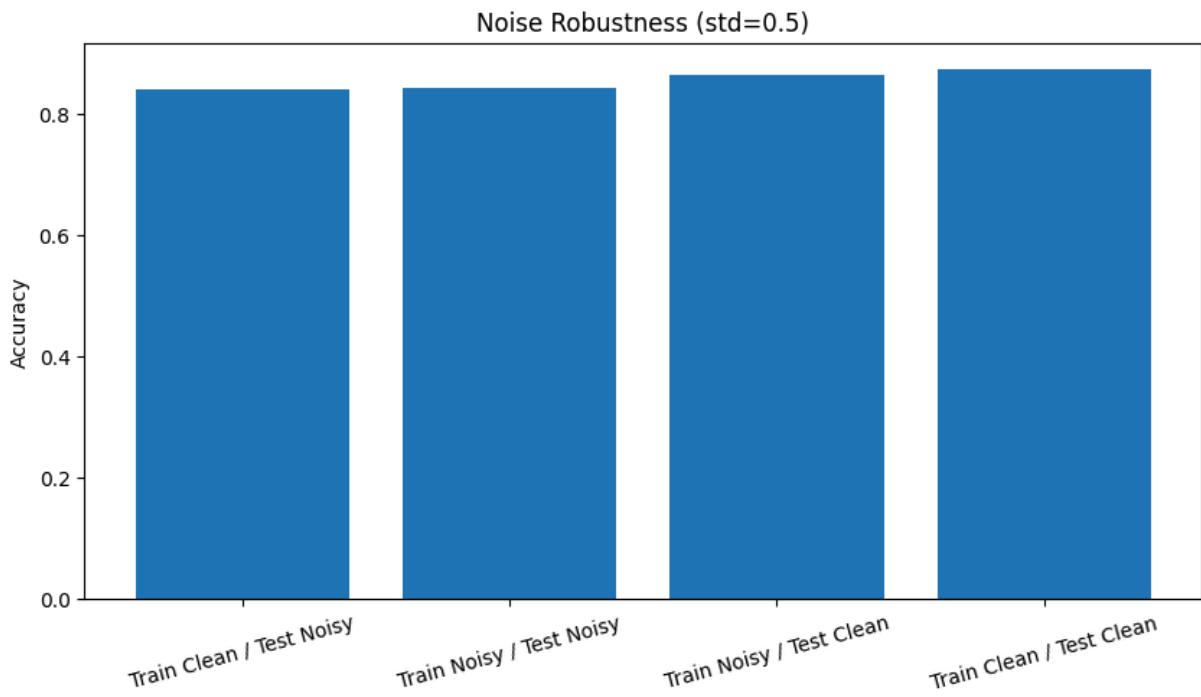
Scenario 3: Train Noisy, Test Clean

Train Noisy, Test Noisy Accuracy: 0.8432

Scenario 3: Train Noisy, Test Clean

Train Noisy, Test Clean Accuracy: 0.8632

Train Noisy, Test Clean Accuracy: 0.8632



## Wnioski - Eksperyment 4

- **Train Clean / Test Noisy:** Model nauczony na czystych danych kompletnie gubi się przy zaszumionych danych wejściowych. To pokazuje brak odporności na zmiany w rozkładzie danych (domain shift).
- **Train Noisy / Test Noisy:** Trenowanie na zaszumionych danych pozwala modelowi "zrozumieć" szum i ignorować go, co drastycznie poprawia wynik na zaszumionym zbiorze testowym.
- **Train Noisy / Test Clean:** Model uczony na szumie często radzi sobie całkiem nieźle na czystych danych (czasem szum działa jak regularyzacja), choć może być nieco gorszy od modelu trenowanego specyficznie na czystych danych.