

Sieci Neuronowe

Sprawozdanie z ćwiczenia 5

Technologia

Python 3.11, Jupyter Notebook, Torchvision

Realizacja ćwiczenia

Cel ćwiczenia

W ćwiczeniu 5 wykorzystamy sieć wielowarstwową do klasyfikacji obrazów. Korzystamy z biblioteki PyTorch oraz skupiamy się na module torchvision.

Architektura modeli

Z racji tego, możemy już w pełni korzystać z bibliotek, to możemy zdefiniować naszą sieć w taki elegancki sposób, korzystając z `nn.Sequential`, który pozwala na łatwe łączenie warstw sieci w sekwencję.

```
oneLayerModel = nn.Sequential(  
    nn.Linear(784, 16),  
    nn.LeakyReLU(),  
    nn.Linear(16, 10),  
    nn.LogSoftmax(dim = 1))
```

```
twoLayerModel = nn.Sequential(  
    nn.Linear(784, 16),  
    nn.LeakyReLU(),  
    nn.Linear(16, 8),  
    nn.LeakyReLU(),  
    nn.Linear(8, 10),  
    nn.LogSoftmax(dim = 1))
```

`nn.Linear(784,16)` to jest warstwa liniowa (fully connected) z 784 wejściami (co odpowiada spłaszczonym obrazom 28x28 pikseli) i 16 neuronami w warstwie ukrytej.

`nn.LeakyRelu()` to jest funkcja aktywacji, która jest stosowana po warstwie liniowej. LeakyReLU to wariant funkcji ReLU, który pozwala na przepływ małego gradientu, gdy wartość wejściowa jest mniejsza od zera.

`nn.Linear(16,10)` to jest druga warstwa liniowa, która przekształca wyjście z poprzedniej warstwy (16 neuronów) do 10 neuronów na wyjściu. Te 10 neuronów na wyjściu odpowiada prawdopodobieństwom 10 klas, do których obrazek może należeć.

`nn.LogSoftmax(dim = 1)` to jest funkcja softmax, która jest stosowana na końcu sieci, aby przekształcić wyjścia sieci na prawdopodobieństwa. LogSoftmax to wariant funkcji softmax, który dodatkowo oblicza logarytm prawdopodobieństw.

W przypadku drugiej sieci dwuwarstwowej dodaliśmy warstwę z liczbą neuronów odpowiadającą połowie liczby neuronów z warstwy poprzedniej.



Pętla uczenia modelu:

```
def train(model, criterion, data_loader, test_loader, epochs):
    #reset model
    model.apply(init_normal)
    optimizer = optim.Adam(model.parameters())
    torch.manual_seed(seed)

    start_timestamp = time.time()
    training_loss = []
    test_loss_list = []
    for epoch in range(epochs):
        running_loss = 0
        test_loss = 0
        with torch.no_grad():
            for images, labels in test_loader:
                images = images.view(images.shape[0], -1)
                logits = model(images)
                loss_test = criterion(logits, labels)
                test_loss += loss_test.item()

        for images, labels in data_loader:
            images = images.view(images.shape[0], -1)
            optimizer.zero_grad()
            logits = model(images)
            loss = criterion(logits, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        running_loss = running_loss/len(data_loader)
        test_loss = test_loss/len(test_loader)
        training_loss.append(running_loss)
        test_loss_list.append(test_loss)

        if (epoch) % 10 == 0:
            print(f"Epoch {epoch}/{epochs} - Train Loss: {running_loss:.4f}, Test Loss: {test_loss:.4f}")

    print(f"\nTraining Time (in seconds) = {(time.time()-start_timestamp):.2f}")
    return training_loss, test_loss_list
```

Inicjalizacja wag:

```
def init_normal(m):
    if isinstance(m, nn.Linear):
        init.xavier_normal_(m.weight)
        init.constant_(m.bias, 0)
```

Transformacja danych:

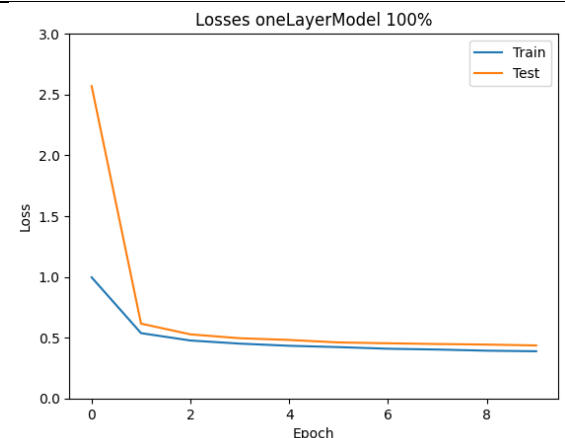
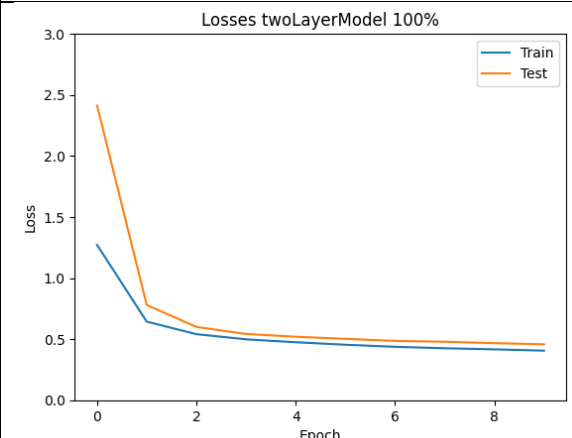
```
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])
```

Dodawanie szumu gaussowskiego:

```
transform_blur = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)),
    GaussianBlur(kernel_size=9, sigma=(1.5, 5.0))
])
```

Wyniki domyślnej konfiguracji

epok	10
batch	500
zbiór	100%

1 (784-16-10)	2 (784-16-8-10)
Training Time (in seconds) = 178.51 Accuracy: 0.8440 Precision: 0.8422 F1 Score: 0.8427	Training Time (in seconds) = 180.58 Accuracy: 0.8414 Precision: 0.8410 F1 Score: 0.8398
 <p>Losses oneLayerModel 100%</p> <p>The graph shows training and test losses for a one-layer model over 10 epochs. The training loss (blue line) starts at approximately 1.0 and decreases to about 0.4. The test loss (orange line) starts at approximately 2.6 and decreases to about 0.5. Both losses stabilize after about 4 epochs.</p>	 <p>Losses twoLayerModel 100%</p> <p>The graph shows training and test losses for a two-layer model over 10 epochs. The training loss (blue line) starts at approximately 1.3 and decreases to about 0.4. The test loss (orange line) starts at approximately 2.4 and decreases to about 0.5. Both losses stabilize after about 4 epochs.</p>

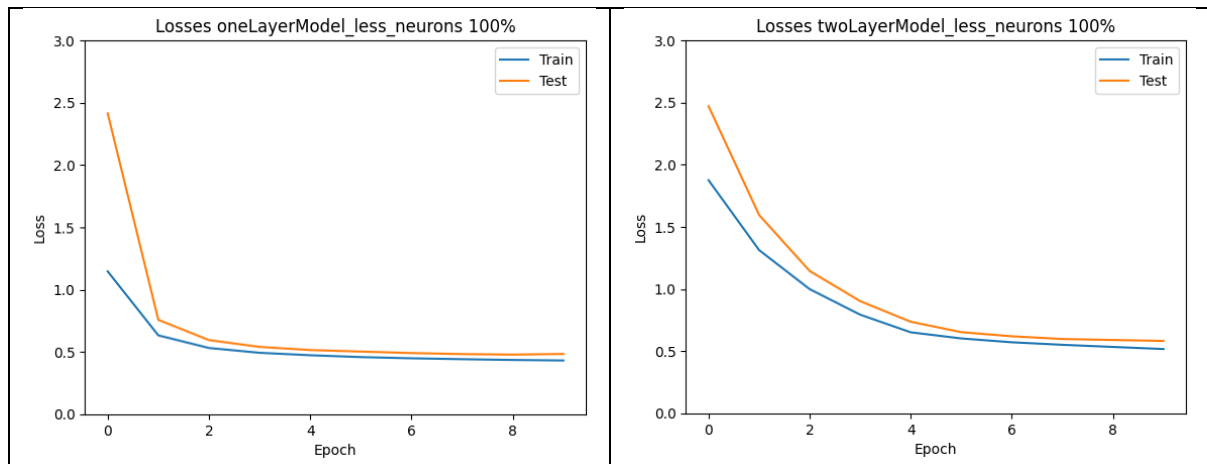
Dużo danych oraz proporcjonalnie duży batch sprawiło, że koszt szybko zmaleł, W przypadku modelu dwuwarstwowego nastąpiło to co typowe dla większej ilości warstw, co też daje więcej neuronów, uczenie zostało spowolnione. Już po czterech epokach modele byłyby na podobnym poziomie wyuczenia. Widać, że model funkcja kosztu szybciej spada dla modelu jednowarstwowego.

Porównanie Liczby neuronów

epok	10
batch	500
zbiór	100%

1 (784-8-10)	2 (784-8-4-10)
<pre>oneLayerModel_less_neurons = nn.Sequential(nn.Linear(784, 8), nn.LeakyReLU(), nn.Linear(8, 10), nn.LogSoftmax(dim = 1))</pre>	<pre>twoLayerModel_less_neurons = nn.Sequential(nn.Linear(784, 8), nn.LeakyReLU(), nn.Linear(8, 4), nn.LeakyReLU(), nn.Linear(4, 10), nn.LogSoftmax(dim = 1))</pre>

1 (784-8-10)	2 (784-8-4-10)
Training Time (in seconds) = 177.99 Accuracy: 0.8328 Precision: 0.8309 F1 Score: 0.8313	Training Time (in seconds) = 180.13 Accuracy: 0.8154 Precision: 0.8107 F1 Score: 0.8110

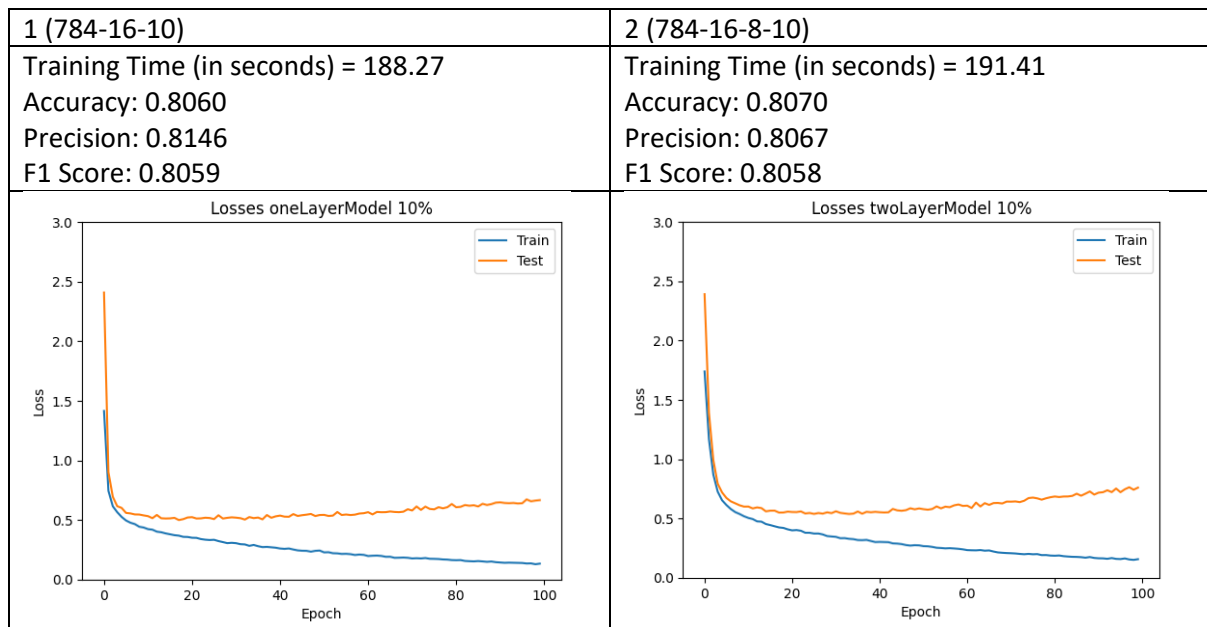


Można zauważyć, że w przypadku modelu jednowarstwowego tempo uczenia się minimalnie zmalało. W modelu dwuwarstwowym tempo znacznie zmalało co widać powyższym wykresie. Trzeba zauważyć, że używamy optymalizatora Adam, gdzie nie definiujemy współczynnika uczenia się przy inicjalizacji zatem dostosowuje on współczynnik uczenia w trakcie treningu.

Porównanie rozmiaru batcha

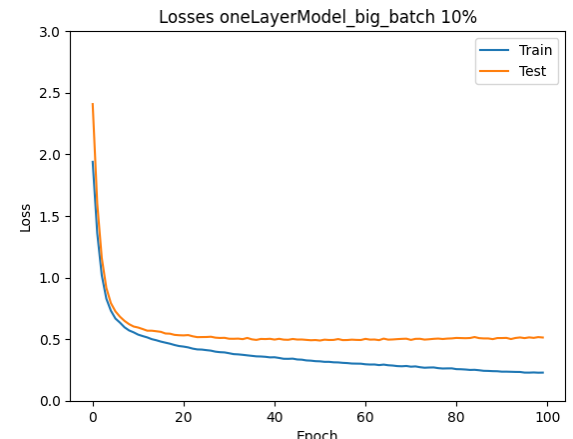
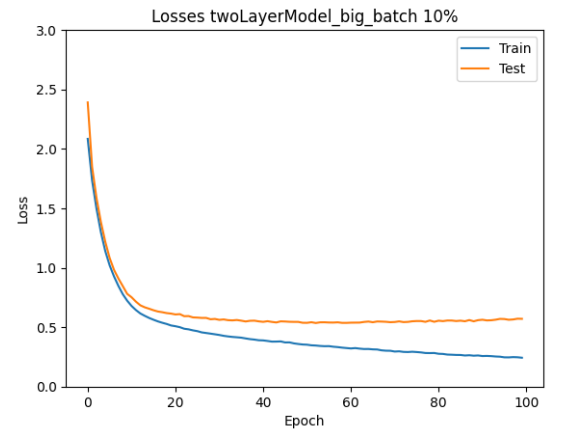
Porównamy dla danych będących 10% podzbiorem całego zbioru.

epok	100
batch	128
zbiór	10%



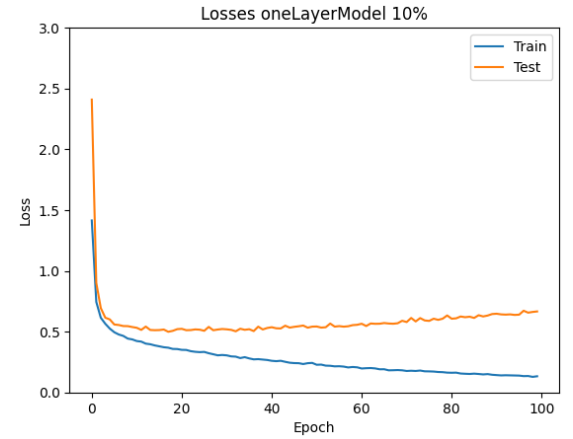
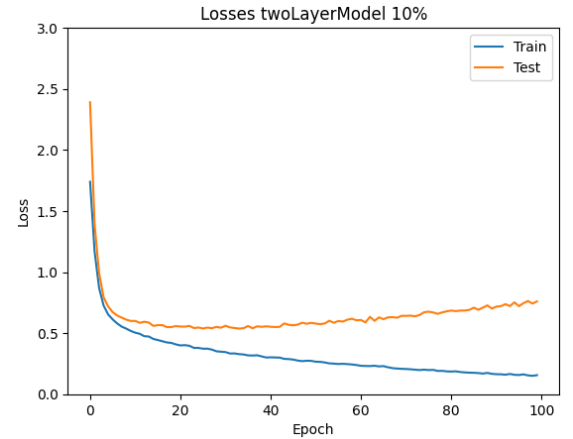
Można zauważyć, że dla mniejszego batcha oraz mniejszego zbioru danych modele zachowują się podobnie. Lekka tendencja do przeuczenia oraz podobne wyniki końcowe, natomiast model jednowarstwowy można uznać za bardziej „stromy”.

epok	100
batch	512
zbiór	10%

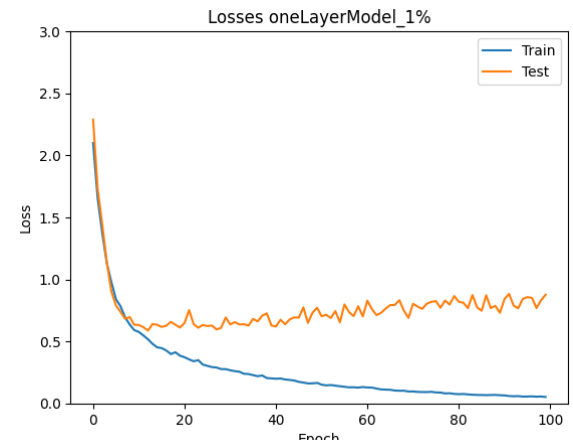
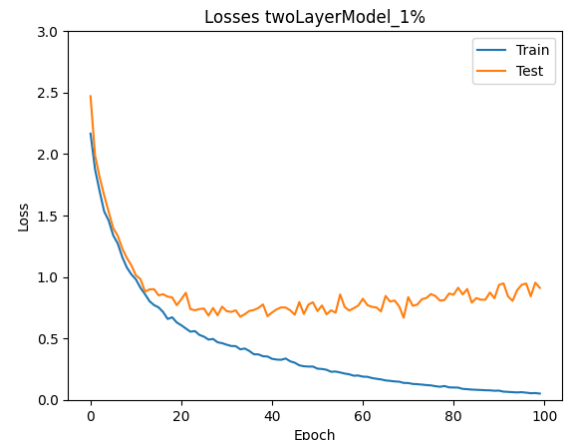
1 (784-16-10)	2 (784-16-8-10)
Training Time (in seconds) = 179.21 Accuracy: 0.8170 Precision: 0.8190 F1 Score: 0.8174	Training Time (in seconds) = 181.82 Accuracy: 0.8290 Precision: 0.8287 F1 Score: 0.8282
 <p>Losses oneLayerModel_big_batch 10%</p>	 <p>Losses twoLayerModel_big_batch 10%</p>

Większy batch zmniejsza tendencję do przeuczenia, ponieważ do liczenia gradientów wchodzi większa paczka danych przez co zmiany są bardziej uśredniane, co prowadzi do spowolnienia uczenia, większych błędów początkowych, natomiast lepsze wyniki końcowe. Model dwuwarstwowy znowu jest bardziej łagodny, ale można zauważyć minimalnie lepszą dokładność.

Porównanie liczby przykładów zbioru uczącego

epok	100
batch	128
zbiór	10% (7000)
1 (784-16-10)	2 (784-16-8-10)
Training Time (in seconds) = 188.27 Accuracy: 0.8060 Precision: 0.8146 F1 Score: 0.8059	Training Time (in seconds) = 191.41 Accuracy: 0.8070 Precision: 0.8067 F1 Score: 0.8058
 <p>Losses oneLayerModel 10%</p>	 <p>Losses twoLayerModel 10%</p>

epok	100
batch	64
zbiór	1%

1 (784-16-10)	2 (784-16-8-10)
Training Time (in seconds) = 20.10 Accuracy: 0.7800 Precision: 0.8217 F1 Score: 0.7885	Training Time (in seconds) = 21.00 Accuracy: 0.7700 Precision: 0.8070 F1 Score: 0.7792
 <p>Losses oneLayerModel_1%</p> <p>The graph shows training and test losses for a one-layer model over 100 epochs. The training loss (blue line) decreases steadily from approximately 2.2 to 0.1. The test loss (orange line) starts at 2.2, drops to about 0.7 by epoch 10, and then fluctuates between 0.6 and 0.9 for the remainder of the training process.</p>	 <p>Losses twoLayerModel_1%</p> <p>The graph shows training and test losses for a two-layer model over 100 epochs. The training loss (blue line) decreases steadily from approximately 2.2 to 0.1. The test loss (orange line) starts at 2.2, drops to about 0.8 by epoch 10, and then fluctuates between 0.7 and 1.0 for the remainder of the training process.</p>

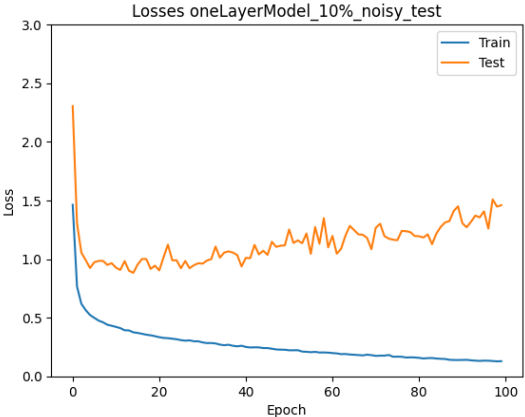
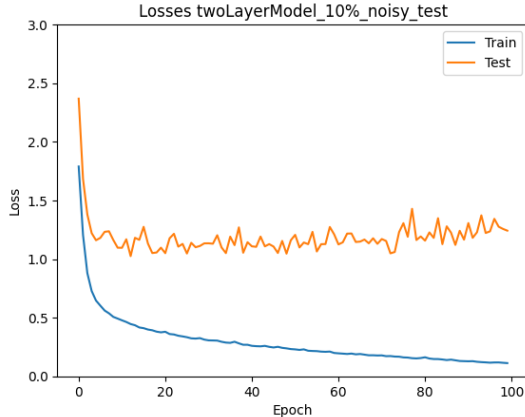
Widać, że znowu zmiana proporcjonalnie paczki przy zmianie wielkości zbioru danych ma wpływ na wyniki modeli. Po pierwsze, ogólnie gorsze wyniki, które wynikają przez mniejszą liczbę danych. Ogólnie większe poziomy błędów. Mniejszy zbiór spowolnił uczenie w obu przypadkach oraz wprowadził gorszą stabilność funkcji kosztu. W modelu warstwowym krzywa funkcji jest bardziej łagodna i nie zesła poziomem błędów danych testowych tak nisko jak w przypadku modelu jednowarstwowego.

Porównanie dodania szumu

Porównamy zachowanie modeli przy zaburzeniu danych szumem gaussowskim przy danych testowych oraz przy zaburzeniu danych treningowych oraz testowych.

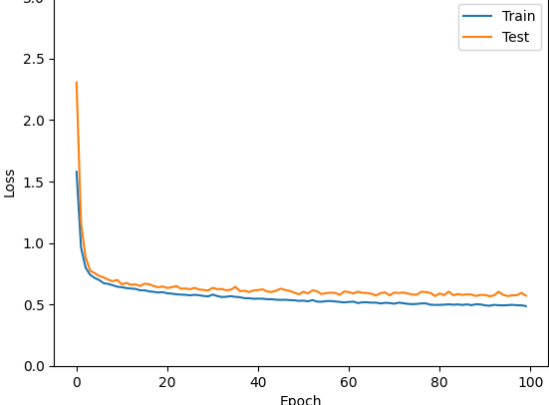
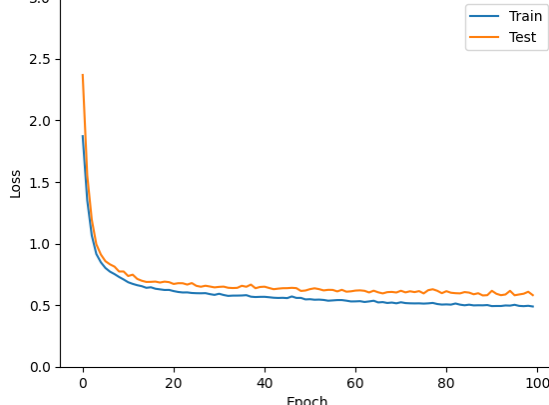
Zaszumienie danych testowych:

Epok	100
Batch	128
Zbiór	10%

1 (784-16-10)	2 (784-16-8-10)
Training Time (in seconds) = 147.60	Training Time (in seconds) = 146.74
Accuracy: 0.6130	Accuracy: 0.6440
Precision: 0.6472	Precision: 0.6611
F1 Score: 0.5572	F1 Score: 0.6209
 <p>Losses oneLayerModel_10%_noisy_test</p> <p>The graph shows training loss (blue line) decreasing from approximately 1.5 to 0.1 over 100 epochs. The test loss (orange line) starts at approximately 2.3, drops to about 1.0, and then fluctuates between 1.0 and 1.5 for the remainder of the training process.</p>	 <p>Losses twoLayerModel_10%_noisy_test</p> <p>The graph shows training loss (blue line) decreasing from approximately 1.8 to 0.1 over 100 epochs. The test loss (orange line) starts at approximately 2.3, drops to about 1.2, and then fluctuates between 1.0 and 1.5 for the remainder of the training process.</p>

Widać, że dodanie szumu tylko do danych testowych nie jest dobrym pomysłem. Krzywe funkcji kosztu danych treningowych są identyczne co do adekwatnego porównania przy tym samym zbiorze powyżej, ale koszty dla danych testowych mają bardzo wysoki poziom. Trzeba zauważyć, że model dwuwarstwowy przez powolniejsze uczenie się miał końcowo lepsze wyniki.

Zaszumienie danych treningowych i testowych:

<p>Training Time (in seconds) = 320.09</p> <p>Accuracy: 0.7900</p> <p>Precision: 0.7835</p> <p>F1 Score: 0.7836</p>	<p>Training Time (in seconds) = 323.62</p> <p>Accuracy: 0.7840</p> <p>Precision: 0.7807</p> <p>F1 Score: 0.7789</p>
<p>Losses oneLayerModel_10%_noisy_test_and_train</p> 	<p>Losses twoLayerModel_10%_noisy_test_and_train</p> 

W tym porównaniu widać, że powinniśmy dodawać szum również dla danych treningowych. Koszty nadal mają wyższe poziomy, ale można rzec, że są bardziej zbieżne. Intuicja podpowiada, że powinniśmy dłużej uczyć modele „zaszumione” ponieważ, nie widać zjawiska przeuczenia, a koszt danych testowych zbiega wraz z kosztem danych treningowych. W tym przypadku minimalnie lepszy okazał się model jednowarstwowy.