Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Sieci neuronowe

Raport ćw. 5

Autor

Szymon Sawczuk

nr albumu: **260287**

kierunek: **Informatyka Stosowana**

23 listopada 2023

Spis treści

1 Ćwiczenie 5 - Rozpoznawanie obrazów

Celem ćwiczenia jest zbudowanie oraz wykorzystanie sieci wielowarstwowej do klasyfikacji obrazów. Dla nauczenia się innej biblioteki wykorzystywanej do tworzenia sieci neuronowych, w tym ćwiczeniu wykorzystuję bibliotekę pytorch (zamiast tensorflow w odrębie keras). Do rozwiązania wykorzystuję również bibliotekę torchvision do pobrania zbioru danych, matplotlib oraz numpy, a także torchmetrics do obliczania miar jakości modelu.

1.1 Zbiór danych

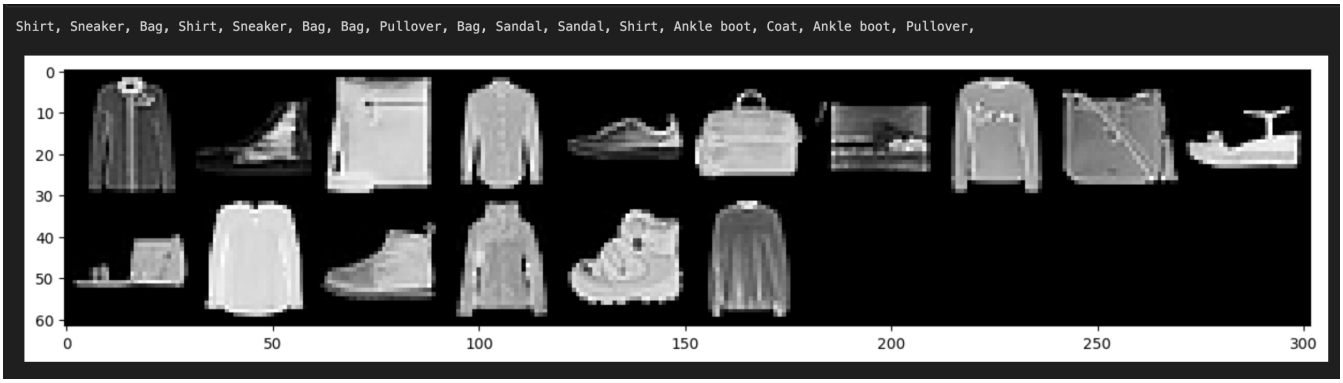
W ćwiczeniu wykorzystuję zbiór danych FashionMNIST (*FashionMNIST*, 2022). Składa się on ze zbioru treningowego o rozmiarze 60 000 przykładów oraz zbioru testowego o rozmiarze 10 000 przykładów. Każdy przykład to zdjęcie w wymiarze 28x28 pikseli w skali szarości, przedstawiające ubranie, do którego przypisana jest jedna z 10 klas ubrania (T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot - indeksowane numerem od 0 do 9).

Pobranie danych przebiega za pomocą biblioteki torchvision. Za pomocą transforms.Compose([transforms.ToTensor()])) przekształcam dane na tensory.

```
train_set = torchvision.datasets.FashionMNIST('path', download = True, train = True,
    transform = transforms.Compose([transforms.ToTensor()]))
test_set = torchvision.datasets.FashionMNIST('path', download = True, train = False,
    transform = transforms.Compose([transforms.ToTensor()]))
```

Następnie za pomocą DataLoader'a załadowuję batch danych o rozmiarze podanym w batch_size.

```
data_loader_train = torch.utils.data.DataLoader(train_set, batch_size=16, shuffle=True)
data_loader_test = torch.utils.data.DataLoader(test_set, batch_size=16, shuffle=True)
```



Rysunek 1: Przykładowy batch danych

1.2 Sieć wielowarstwowa

Każda warstwa ukryta sieci składa się z warstwy torch.nn.Linear, która przekształca liniowo dane wejściowe za pomocą wzoru: $y = xA^T + b$. Po niej stosowana jest funkcja aktywacji Leaky ReLU, która pozwala w porównaniu do zwykłego ReLU ustawić niezerowy gradient dla wartości ujemnych (może to zapobiec problemowi nieaktywnego neuronu). Wzór Leaky ReLU:

$$f(x) = \begin{cases} x & \text{jeżeli x} > 0 \\ \alpha * x & \text{jeżeli x} <= 0 \end{cases}$$

Gdzie α to mały współczynnik nachylenia dla wartości ujemnych, na przykład 0.01.

W warstwie wyjściowej po wykonaniu transformacji liniowej stosuję funkcję LogSoftMax, aby przekształcić wyniki wyjściowe na prawdopodobieństwa.

```
class MultilayerNetwork(torch.nn.Module):
    def __init__(self, hidden_layers_sizes: tuple, input_size: int, output_size: int):
        super(MultilayerNetwork, self).__init__()

        self._layers = torch.nn.ModuleList()
        curr_size = input_size
        for hidden_layer_size in hidden_layers_sizes:
            layer = torch.nn.Linear(curr_size, hidden_layer_size)
            torch.nn.init.normal_(layer.weight)
            torch.nn.init.normal_(layer.bias)
            self._layers.append(layer)
            self._layers.append(torch.nn.LeakyReLU())
            curr_size = hidden_layer_size
        output_layer = torch.nn.Linear(curr_size, output_size)
        torch.nn.init.normal_(output_layer.weight)
        torch.nn.init.normal_(output_layer.bias)
        self._layers.append(output_layer)
        self._layers.append(torch.nn.LogSoftmax(dim=1))

    def forward(self, inputs):
        x = inputs
        for layer in self._layers:
            x = x.view(x.shape[0], -1)
            x = layer(x)
        return x
```

Do trenowania modelu utworzyłem funkcję podobną do tej z poprzedniego zadania, główną różnicą w pythorch'u do tensorflow'a, jest zerowanie gradientu przy każdej iteracji po batch'ach z powodu akumulacji gradientów, a nie ich nadpisywania. Batch'e pobierane są za pomocą wcześniej wymienionego DataLoader'a. A miary jakości modelu obliczane są za pomocą klas o prefiksie MultiClass z biblioteki torchmetrics (pozwalają one nam porównywać predykcje do wyjść danych wieloklasowo, a nie binarnie jak to było w poprzednich zadaniach).

```
def train_model(model: torch.nn.Module, train_set: torch.Tensor, test_set: torch.Tensor,
    batch_size: int, optimizer, loss, max_iter: int, learning_rate: float, output_size:
    int, verb=False):

    optimizer = optimizer(model.parameters(), lr = learning_rate)

    losses = []
    losses_test = []

    accuracy = []
    precision = []
    f_score = []
```

```
recalls = []

if batch_size > len(train_set):
    batch_size = len(train_set)

data_loader_train = torch.utils.data.DataLoader(train_set, batch_size=batch_size,
    shuffle=True)
data_loader_test = torch.utils.data.DataLoader(test_set, batch_size=batch_size,
    shuffle=True)

metric_accuracy = MulticlassAccuracy(num_classes=output_size)
metric_precision = MulticlassPrecision(num_classes=output_size)
metric_f_score = MulticlassF1Score(num_classes=output_size)
metric_recall = MulticlassRecall(num_classes=output_size)

for index in range(max_iter): #learn for max_iter
    curr_train_loss = 0
    curr_test_loss = 0
    metric_accuracy.reset()
    metric_precision.reset()
    metric_f_score.reset()
    metric_recall.reset()

    # for each batch perform learning
    for x_train_batch, y_train_batch in data_loader_train:
        optimizer.zero_grad()
        y_pred = model(x_train_batch)
        loss_value = loss(y_pred, y_train_batch)
        loss_value.backward()
        optimizer.step()
        curr_train_loss += loss_value.item()

    losses.append(curr_train_loss / len(data_loader_train))

    with torch.no_grad():
        model.eval()
        for x_test_batch, y_test_batch in data_loader_test:
            test_pred = model(x_test_batch)
            loss_test_value = loss(test_pred, y_test_batch)
            curr_test_loss += loss_test_value.item()

            #calculate scores for each batch of iteration
            metric_accuracy.update(test_pred, y_test_batch)
            metric_precision.update(test_pred, y_test_batch)
            metric_f_score.update(test_pred, y_test_batch)
            metric_recall.update(test_pred, y_test_batch)

        losses_test.append(curr_test_loss / len(data_loader_test))
        accuracy.append(metric_accuracy.compute())
        precision.append(metric_precision.compute())
        f_score.append(metric_f_score.compute())
        recalls.append(metric_recall.compute())

    if verb and index % 10 == 0:
        print("------------------------------------------- Interation " + str(index))
        print("Train loss on " + str(index) + " iteration: ", losses[index])
        print("Test loss on " + str(index) + " iteration: ", losses_test[index])
        print("Accuracy on " + str(index) + " iteration: ", accuracy[index])
        print("Precision on " + str(index) + " iteration: ", precision[index])
        print("Recall on " + str(index) + " iteration: ", recalls[index])
        print("Fscore on " + str(index) + " iteration: ", f_score[index])
```

```
            print("-----------------------------------------------\n")

    print("Result of learning process for " + str(max_iter) + " iterations")
    print("-------------------------------------------------------\n")
    print("Train loss: ", losses[-1])
    print("Test loss: ", losses_test[-1])
    print("-----------------------------------------------\n")
    print("Scores")
    print("Accuracy: ", accuracy[-1])
    print("Precision: ", precision[-1])
    print("F_score: ", f_score[-1])
    print("Recall: ", recalls[-1])
    return losses, losses_test, accuracy, f_score, recalls
```

Warstwa wejściowa modelu posiada $28 * 28 = 784$ neurony, a wyjściowa 10 neuronów (10 klas ubrań).

1.3 Eksperymenty na różnych parametrach modelu

Funkcja do wykonywania eksperymentów:

```
# options [0] -> hidden_layers, [1] -> train_set, [2] -> test_set, [3] -> output_size, [4]
    -> batch_size
def run_models(options: list, options_title: list):
    fig, axs = plt.subplots(1, len(options), figsize=(20, 5))
    for option_index, option in enumerate(options):
        print(options_title[option_index])
        input_size = option[1][0][0].shape[1] * option[1][0][0].shape[2] # width times
            height of image
        model = MultilayerNetwork(option[0], input_size, option[3])
        result = train_model(model, option[1], option[2], option[4], optimizer, loss,
            max_iter, learning_rate, option[3], verb=verbose)
        plot_learning(result[0], result[1], options_title[option_index], axs[option_index])
```

W celu zaburzenia danych wykorzystuję transforms.GaussianBlur(), który rozmazuje nam dane zdjęć wejściowych.

Parametry, które są wspólne dla wszystkich poniższych testów to:

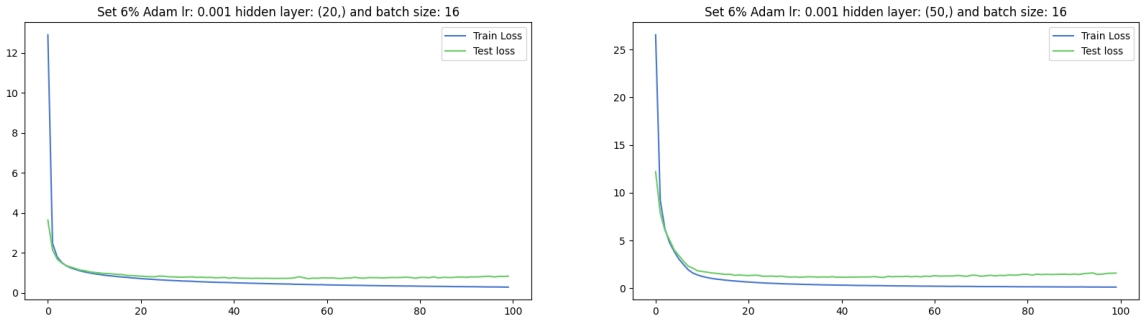
- Optimizer - Adam
- Funkcja kosztu - Entropia krzyżowa
- Liczba iteracji - 100
- Learning rate - 0.001

1.4 Model jednowarstwowy

1.4.1 Liczba neuronów w warstwie ukrytej

Dla 6% danych ze zbioru danych oraz batch'a 16 uzyskujemy wyniki:

	(20,)	(50,)
Dokładność	0.7938	0.7804
Precyzja	0.7999	0.7825
F score	0.7951	0.7789
Recall	0.7938	0.7804



Rysunek 2: Liczba neuronów w warstwie ukrytej

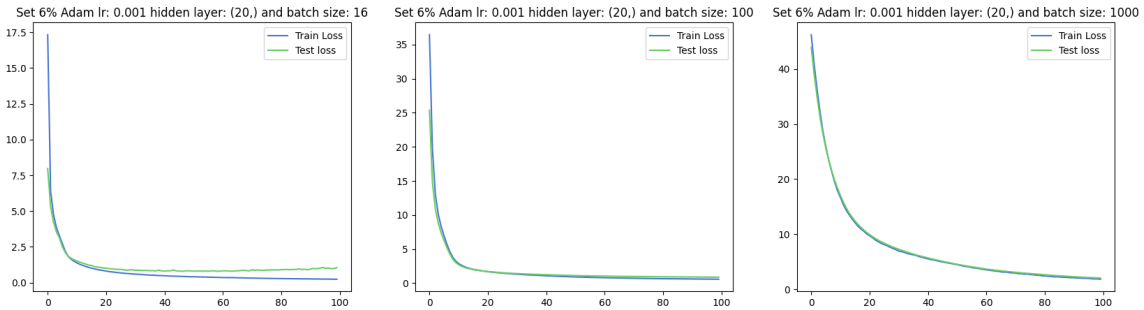
Wnioski:

- Przy zwiększeniu liczby neuronów w warstwie ukrytej naszej sieci możemy zauważyć pogorszenie się miar jakości, co może wynikać ze zbyt bardzo rozbudowanej warstwy neuronów względem dobranych danych.
- Możemy także zauważyć słabsze uogólnianie swojej wiedzy przez sieć z 50 neuronami w warstwie ukrytej

1.4.2 Rozmiar batch'a

Dla 6% danych ze zbioru danych oraz warstwy ukrytej 20 neuronów uzyskujemy wyniki:

	16	100	1000
Dokładność	0.7749	0.6965	0.5569
Precyzja	0.7854	0.7091	0.5647
F score	0.7792	0.6928	0.5561
Recall	0.7749	0.6965	0.5569



Rysunek 3: Rozmiar batch'a

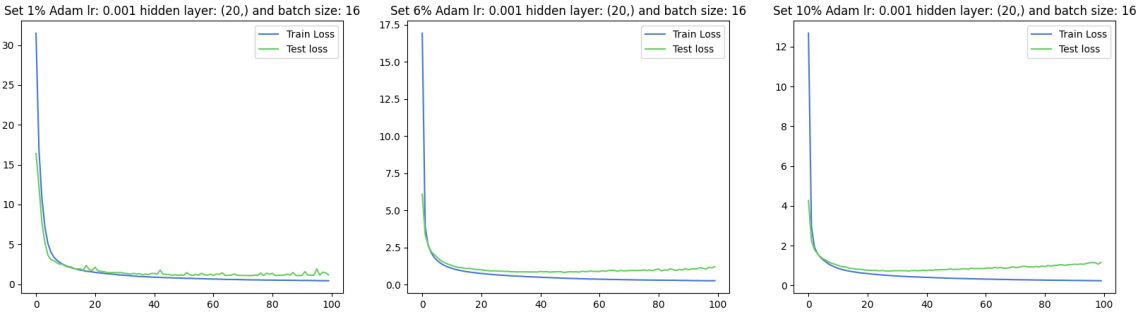
Wnioski:

- Przy zwiększeniu rozmiaru batch'a zauważamy coraz większe niedouczenie modelu, może to być spowodowane mniejszą ilością aktualizacji wag w iteracji
- Możemy też zauważyć podczas zwiększania rozmiaru, większą stabilizacje nauki modelu, możliwe że przy większej liczbie iteracji model o większym batch'u mógł osiągnąć lepsze wyniki od modelu z mniejszym batch'em

1.4.3 Liczba przykładów uczących

Dla warstwy ukrytej 20 neuronów oraz rozmiaru batch'a 16 uzyskujemy wyniki:

	1%	6%	10%
Dokładność	0.7296	0.7648	0.7985
Precyzja	0.7419	0.7698	0.7991
F score	0.7282	0.7652	0.7981
Recall	0.7296	0.7648	0.7985



Rysunek 4: Liczba przykładów uczących

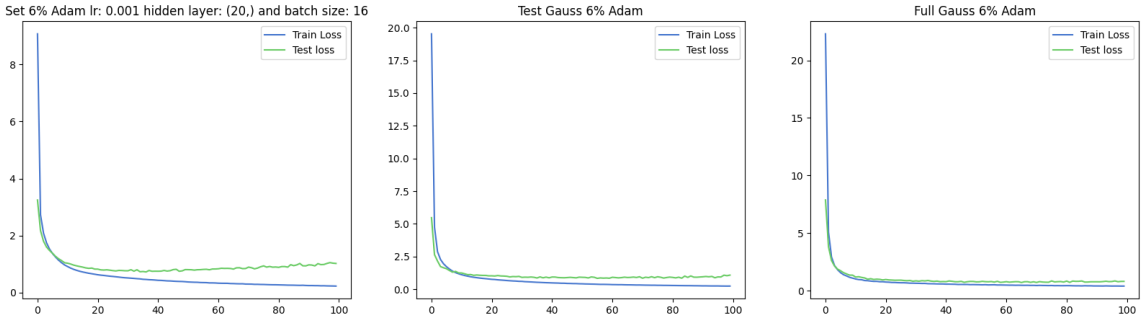
Wnioski:

- Przy zwiększeniu ilości danych uczących i testowych zauważamy lepsze wyniki miar jakości sieci
- Można także zauważyć szybciej pojawiający się problem przeuczenia.

1.4.4 Zaburzenie danych

Dla 6% danych ze zbioru danych, warstwy ukrytej 20 neuronów oraz rozmiaru batch'a 16 uzyskujemy wyniki:

	Brak zaburzenia	Zaburzenie danych testowych	Zaburzenie danych testowych i treningowych
Dokładność	0.7837	0.7221	0.8028
Precyzja	0.7820	0.7312	0.8004
F score	0.7823	0.7163	0.7974
Recall	0.7837	0.7221	0.8028



Rysunek 5: Zaburzenie danych

Wnioski:

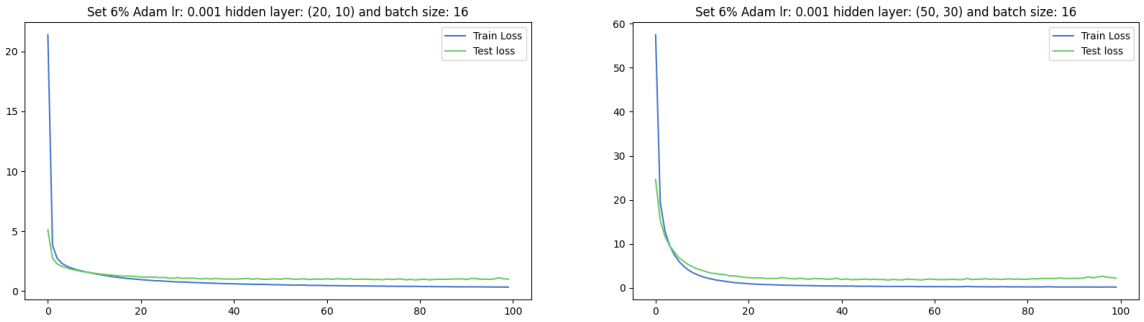
- Zaburzenie danych testowych pozwoliło zmniejszyć funkcję kosztu dla danych testowych i zmniejszyć przeuczenie modelu, wyniki jednak są gorsze od danych bez zaburzenia
- Zaburzenie natomiast danych treningowych i testowych pozwoliło ustabilizować naukę modelu, zapobiec przeuczeniu modelu, a także zwiększyć wyniki miar jakości. Może to wynikać z zwiększenia uogólnienia obrazków za pomocą blur'a.

1.5 Model dwuwarstwowy

1.5.1 Liczba neuronów w warstwie ukrytej

Dla 6% danych ze zbioru danych oraz batch'a 16 uzyskujemy wyniki:

	(20, 10)	(50, 30)
Dokładność	0.7706	0.7517
Precyzja	0.7741	0.7593
F score	0.7694	0.7542
Recall	0.7706	0.7517



Rysunek 6: Liczba neuronów w warstwie ukrytej

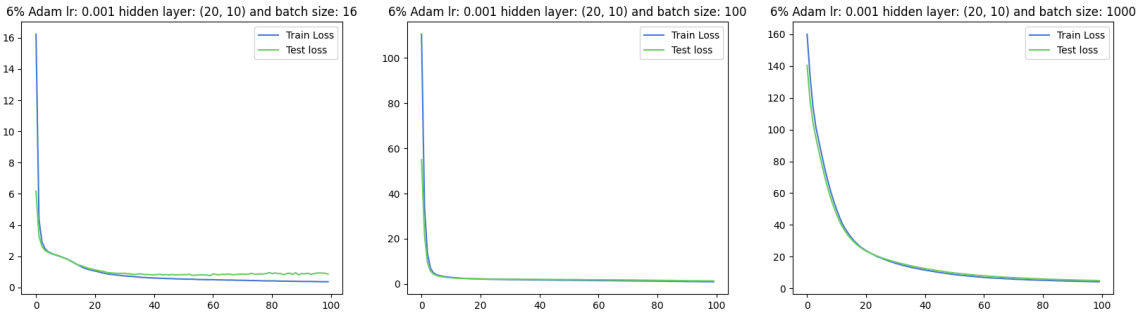
Wnioski:

- Możemy zauważyć podobne zależności jak dla sieci jednowarstwowej
- Wyniki miar dla warstw (20, 10) są gorsze niż dla jednej warstwy 20 neuronów.

1.5.2 Rozmiar batch'a

Dla 6% danych ze zbioru danych oraz warstwy ukrytej (20, 10) neuronów uzyskujemy wyniki:

	16	100	1000
Dokładność	0.7989	0.6528	0.2638
Precyzja	0.8058	0.6644	0.2236
F score	0.8017	0.6516	0.2101
Recall	0.7989	0.6528	0.2638



Rysunek 7: Rozmiar batch'a

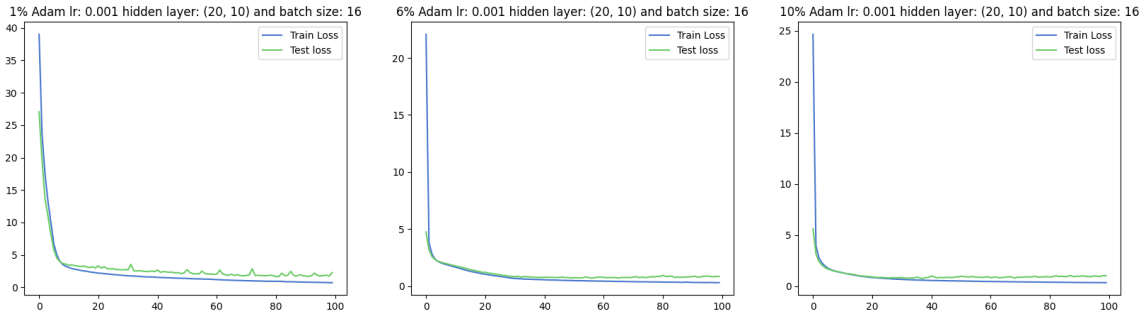
Wnioski:

- Możemy zauważyć, że model o warstwach (20, 10) i batch'u 16 nie zawsze może mieć gorsze wyniki miar od jednej warstwy (20,) i same modele są dość zbliżone do siebie.
- Widzimy za to jeszcze większe ustabilizowanie się nauki modelu dla większego batch'a, a także jeszcze większe niedouczenie dla batch'a rozmiaru 1000.

1.5.3 Liczba przykładów uczących

Dla warstwy ukrytej (20, 10) neuronów oraz rozmiaru batch'a 16 uzyskujemy wyniki:

	1%	6%	10%
Dokładność	0.6188	0.7938	0.8002
Precyzja	0.6331	0.7987	0.8047
F score	0.6174	0.7894	0.8009
Recall	0.6188	0.7938	0.8002



Rysunek 8: Liczba przykładów uczących

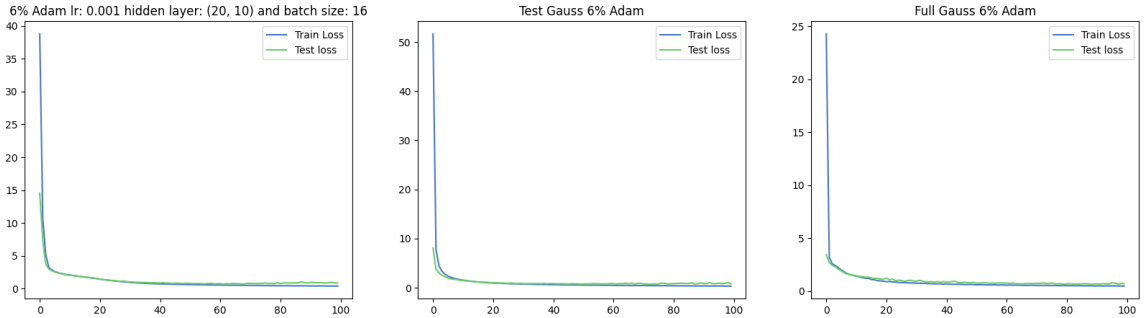
Wnioski:

- Podobne wnioski jak dla modelu jednowarstwowego
- Możemy jednak zauważyć, że w porównaniu do jednej warstwy model uczy się stabilniej (mniejszy problem przeuczenia), co przejawia się w lepszym wyniku dla 10% danych, a także większym niedouczeniem dla 1% danych

1.5.4 Zaburzenie danych

Dla 6% danych ze zbioru danych, warstwy ukrytej 20 neuronów oraz rozmiaru batch'a 16 uzyskujemy wyniki:

	Brak zaburzenia	Zaburzenie danych testowych	Zaburzenie danych testowych i treningowych
Dokładność	0.7781	0.7527	0.7956
Precyzja	0.7813	0.7489	0.7886
F score	0.7779	0.7425	0.7819
Recall	0.7781	0.7527	0.7956



Rysunek 9: Zaburzenie danych

Wnioski:

- Wnioski podobne, jak dla sieci z jedną warstwą

- Widzimy jednak większe ustabilizowanie, już nawet modelu bez zaburzeń danych, mimo gorszych wyników miar jakości, przy większej liczbie iteracji model z dwoma warstwami mógłby osiągać lepsze wyniki niż dla jednej warstwy

Literatura

FashionMNIST. (2022, marzec). Retrieved from `https://github.com/zalandoresearch/fashion-mnist`