Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Sieci neuronowe

Raport ćw. 6

Autor

Szymon Sawczuk

nr albumu: **260287**

kierunek: **Informatyka Stosowana**

8 grudnia 2023

Spis treści

1 Ćwiczenie 6 - Sieć z warstwą CNN

Celem ćwiczenia jest zbudowanie oraz wykorzystanie sieci do klasyfikacji obrazów z użyciem warstwy konwolucyjnej oraz operatora max pooling. W tym ćwiczeniu wykorzystuję bibliotekę pytorch oraz zbiór danych FashionMNIST (taki sam jak w poprzednim ćwiczeniu). Do rozwiązania wykorzystuję również bibliotekę torchvision do pobrania zbioru danych, matplotlib oraz numpy, a także torchmetrics do obliczania miar jakości modelu.

1.1 Sieć z warstwą CNN

W pierwszej warstwie sieci wykorzystuje warstwę CNN, za pomocą conv2d (*Pytorch CONV2D*, n.d.), podając wejście (w tym zadaniu będzie wynosić 1), liczbę kanałów wyjściowych oraz rozmiar filtru. Inicjalizuję wagi oraz bias tej warstwy. Po niej stosowana jest funkcja aktywacji Leaky ReLU. Następnie za pomocą maxpool2d (*Pytorch MAXPOOL2D*, n.d.) wykonuje operację poolingu o rozmiarze pool_size. A następnie spłaszczam dane za pomocą Flatten(). Kolejne warstwy są podobne jak dla warstw ukrytych sieci MLP, z różnicą do poprzedniego zadania, że zamiast warstwy Linear wykorzystuję LazyLinear, dzięki czemu nie muszę podawać liczby danych wejściowych do warstwy.

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

```
class MultilayerNetworkCNN(torch.nn.Module):
    def __init__(self, hidden_layers_sizes: tuple, input_size: int, output_size: int,
        out_channels: int, kernel_size: int, pool_size: int):
        super(MultilayerNetworkCNN, self).__init__()

        self._layers = torch.nn.ModuleList()
        curr_size = input_size

        # Init conv2d layer with maxpool2d
        cnn_layer = torch.nn.Conv2d(curr_size, out_channels, kernel_size=kernel_size)

        self._layers.append(cnn_layer)
        self._layers.append(torch.nn.LeakyReLU())
        self._layers.append(torch.nn.MaxPool2d(kernel_size=pool_size))
        self._layers.append(torch.nn.Flatten())

        for hidden_layer_size in hidden_layers_sizes:
            curr_size = hidden_layer_size
            layer = torch.nn.LazyLinear(curr_size)
            self._layers.append(layer)
            self._layers.append(torch.nn.LeakyReLU())
        output_layer = torch.nn.LazyLinear(output_size)
        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 = layer(x)
        return x
```

Do trenowania modelu utworzyłem funkcję podobną do tej z poprzedniego zadania. 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.

```
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.view(batch_size, 1, 28, 28))
        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
```

Do inicjalizacji dla warstw Lazy potrzebna jest metoda okreslajaca sposob ich inicjalizacji:

```
def init_weights_and_bias(model):
    if isinstance(model, torch.nn.Conv2d) or isinstance(model, torch.nn.Linear):
        torch.nn.init.normal_(model.weight)
        torch.nn.init.normal_(model.bias)
```

1.2 Eksperymenty na różnych parametrach modelu

Funkcja do wykonywania eksperymentów:

```
# options [0] -> out_channels, [1] -> train_set, [2] -> test_set, [3] -> kernel_size, [4]
    -> pool_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])
        model = MultilayerNetworkCNN(hiddenLayer_20_10, input_size, output_size, option[0],
            option[3], option[4])
        dummy = torch.rand(1, 1, 28, 28)
        model(dummy)
        model.apply(init_weights_and_bias)
        result = train_model(model, option[1], option[2], batch_size, optimizer, loss,
            max_iter, learning_rate, output_size, verb=verbose)
        plot_learning(result[0], result[1], options_title[option_index], axs[option_index])
```

W pierwszym kroku przed uczeniem, wykonuje jedne przejście po sieci z fikcyjnymi danymi o wymiarach zdjecia, aby zainicjalizować wagi i bias'y.

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

```
train_set_gauss = torchvision.datasets.FashionMNIST('path', download = True, train = True,
    transform = transforms.Compose([transforms.ToTensor(),
    transforms.GaussianBlur(kernel_size=(7, 7))]))
test_set_gauss = torchvision.datasets.FashionMNIST('path', download = True, train = False,
    transform = transforms.Compose([transforms.ToTensor(),
    transforms.GaussianBlur(kernel_size=(7, 7))]))
```

Parametry, które są wspólne dla wszystkich poniższych testów to:

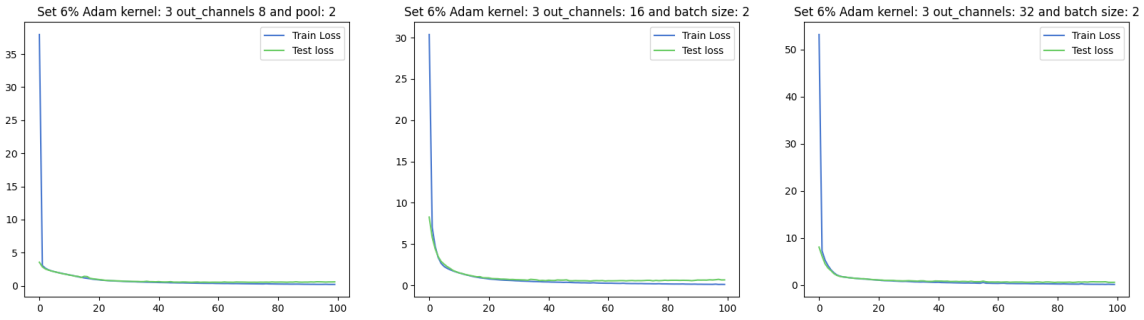
- Optimizer - Adam
- Funkcja kosztu - Entropia krzyżowa
- Liczba iteracji - 100
- Learning rate - 0.001
- Liczba wejść - 1
- Liczba wyjść - 10
- Dane 6% zbioru FashionMNIST
- Rozmiar warstw ukrytych - (20, 10)
- Rozmiar batch'a - 16

1.3 Wyniki eksperymentów

1.3.1 Liczba kanałów wyjściowych warstwy konwolucyjnej

Dla rozmiaru filtra (3,3) oraz rozmiaru pool'a (2,2) otrzymujemy:

	8	16	32
Dokładność	0.8190	0.8354	0.8388
Precyzja	0.8192	0.8395	0.8422
F score	0.8172	0.8285	0.8393
Recall	0.8190	0.8354	0.8388



Rysunek 1: Liczba kanałów wyjściowych warstwy konwolucyjnej

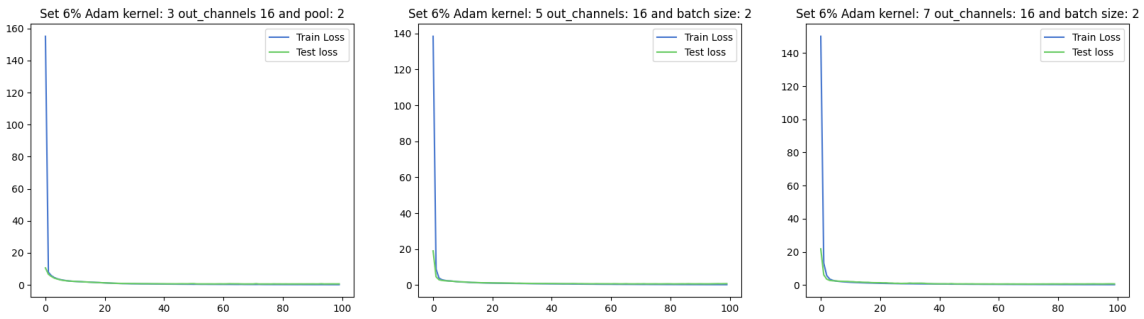
Wnioski:

- Zwiększenie liczby kanałów wyjściowych pozwoliło na zwiększenie wyników miar sieci
- Możemy zauważyć słabsze uogólnianie swojej wiedzy przez sieć o 16 kanałach, natomiast kolejne zwiększenie ich do liczby 32 pozwoliło na lepszą generalizację wiedzy, zatem można byłoby zwiększyć ilość iteracji uczenia, w celu uzyskania potencjalnie lepszych wyników miar

1.3.2 Rozmiar filtra warstwy konwolucyjnej

Dla liczby kanałów wyjściowych 16 oraz rozmiaru pool'a (2,2) otrzymujemy:

	(3,3)	(5,5)	(7,7)
Dokładność	0.8460	0.8159	0.8069
Precyzja	0.8472	0.8182	0.8107
F score	0.8455	0.8138	0.8071
Recall	0.8460	0.8159	0.8069



Rysunek 2: Rozmiar filtra warstwy konwolucyjnej

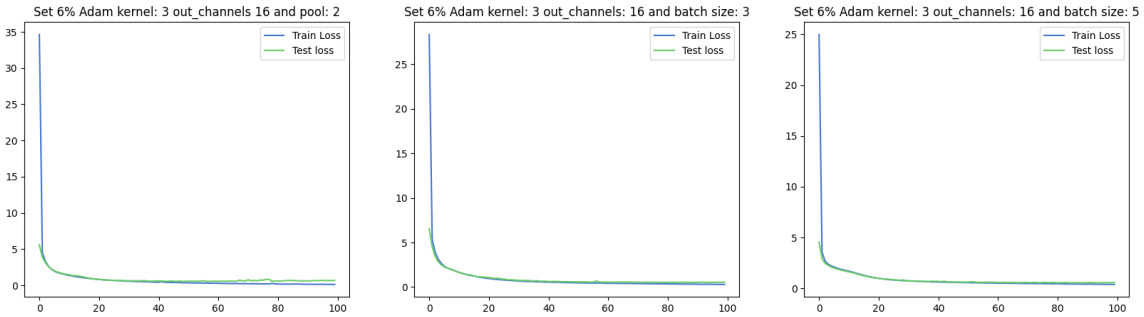
Wnioski:

- Przy zwiększeniu rozmiaru filtra pogorszyły się wyniki sieci, może to wynikać z mniejszej ilości wyciągniętych cech ze zdjęcia
- Natomiast możemy zobaczyć, że rozmiar filtra nie wpłynął na stabilność sieci, a także na pojawienie się zjawiska przeuczenia dla tej ilości iteracji

1.3.3 Rozmiar okna poolingu

Dla liczby kanałów wyjściowych 16 oraz rozmiaru filtra (3,3) otrzymujemy:

	2	3	5
Dokładność	0.8426	0.8208	0.8143
Precyzja	0.8425	0.8251	0.8080
F score	0.8411	0.8186	0.8025
Recall	0.8426	0.8208	0.8143



Rysunek 3: Rozmiar okna poolingu

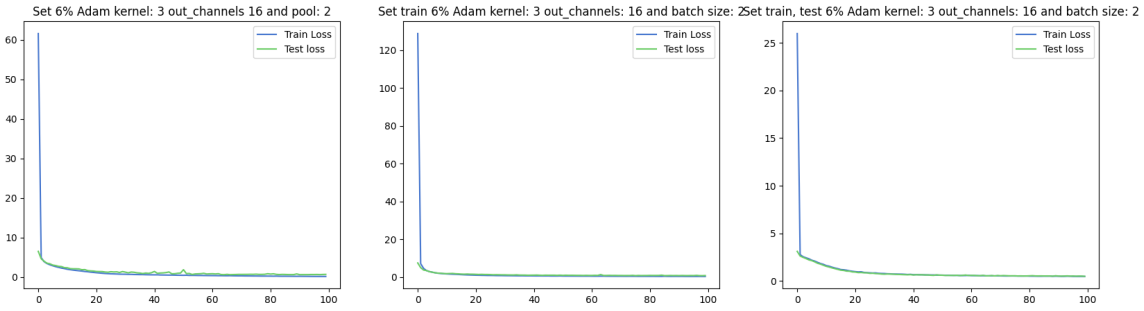
Wnioski:

- Przy zwiększeniu rozmiaru okna zauważamy pogorszenie się wyników sieci, spowodowane jest to zmniejszoną ilością cech przekazywanych po operacji poolingu do dalszych warstw sieci
- Możemy jednak zauważyć lekką niestabilność sieci dla okna (2,2)

1.3.4 Zaburzenie danych

Dla liczby kanałów wyjściowych 16, rozmiaru filtra (3,3) oraz rozmiaru pool'a (2,2) otrzymujemy:

	Brak zaburzenia	Zaburzenie danych testowych	Zaburzenie danych testowych i treningowych
Dokładność	0.8282	0.7055	0.8199
Precyzja	0.8332	0.7631	0.8186
F score	0.8282	0.7201	0.8185
Recall	0.8282	0.7055	0.8199



Rysunek 4: Zaburzenie danych

Wnioski:

- Zaburzenie danych testowych i treningowych pozwoliło zmniejszyć funkcję kosztu dla danych testowych i zmniejszyć przeuczenie modelu, wyniki są podobne, jak dla danych bez zaburzenia
- Zaburzenie te pozwoliło także ustabilizować naukę modelu oraz zapobiec przeuczeniu modelu
- Zaburzenie tylko danych testowych pogorszyło wyniki osiągane przez sieć

Wyniki sieci z warstwą CNN w porównaniu do sieci bez takiej warstwy badanej w poprzednim zadaniu, osiągają lepsze wyniki na poziomie 85% wszystkich miar, gdzie modele w poprzednim zadaniu dla tej samej liczby iteracji nauki osiągały wyniki w okolicach maksymalnie 80%.

Literatura

Pytorch CONV2D. (n.d.). Retrieved from `https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html`

Pytorch MAXPOOL2D. (n.d.). Retrieved from `https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html`