WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Jakub Jucha

Rozpoznawanie cyfr zbioru danych MNIST za pomocą sieci głębokiej

Projekt z przedmiotu Sztuczna Inteligencja

Rzeszów, 2024

Spis treści

1. Wstęp

1.1. Cel projektu

Głównym celem projektu jest wykorzystanie technik głębokiego uczenia do efektywnej klasyfikacji cyfr pisanych odręcznie. Poprzez wykorzystanie języka programowania „Python” oraz biblioteki „PyTorch”, projekt ten stara się stworzyć model, który potrafi rozpoznawać cyfry z dużą dokładnością, nawet w przypadku różnorodnych stylów pisania i odchyleniach od idealnych kształtów cyfr. W ramach projektu zbadano wpływ poniższych parametrów głębokiej sieci na efekt uczenia:

- Architektura sieci neuronowej, w tym rozmiar warstw konwulacyjnych oraz liniowych
- Regularyzacja (dropout) - Parametr dropout kontroluje stopień regularyzacji, czyli losowego wyłączania neuronów w trakcie treningu
- Rozmiar jądra konwulacyjnego – Parametr ten określa rozmiar jądra używanego w warstwach konwulacyjnych

Dzięki powyższym parametrom projekt będzie testował efektywność klasyfikacji cyfr pisanych odręcznie za pomocą głębokiej sieci neuronowej.

1.2. Opis danych uczących

Do treningu głębokiej sieci wykorzystany został zbiór danych MNIST. Jest to zbiór 70.000 obrazów (60.000 treningowych i 10.000 testowych) w skali szarości, każdy o rozdzielczości 28x28 pikseli (daje to łącznie 784 piksele) przedstawiające różne wersje ręcznie pisanych cyfr od 0 do 9. Każdy z obrazów powiązany jest z wartością liczbową będącą równą cyfrze, którą obraz przedstawia.

Rysunek 1. Przykładowe obrazy[1]

Do wczytania danych została wykorzystana biblioteka „PyTorch", a dane zostały poddane normalizacji.

```
#odchylenie standardowe
std = (0.3081)

#wartość średnia
mu = (0.1307,)

#używam dataloader
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST( root: '/filse/', train=True, download=True, transform=torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mu,std)
    ])),
    batch_size = batch_size, shuffle = True
)
```

Rysunek 2. Wczytanie danych wraz z normalizacją

Do uczenia zostały wykorzystane 64 obrazy (batch_size = 64).

[1] https://en.wikipedia.org/wiki/MNIST_database (dostęp: 28.04.2024)

2. Zagadnienia teoretyczne

2.1. Jak działa sieć CNN?

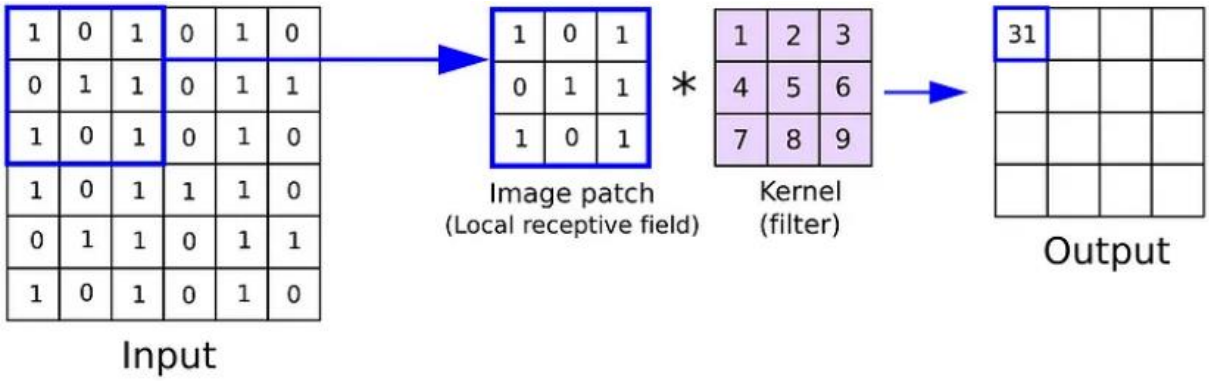
Architektura CNN odzwierciedla ludzki system wizualny, z warstwami przeznaczonymi do wydobywania cech niskiego poziomu, takich jak krawędzie i tekstury, stopniowo przechodząc do reprezentacji wysokiego poziomu, umożliwiając sieci rozpoznawanie skomplikowanych wzorów i obiektów na obrazie.

Pierwsza warstwa to warstwa splotu. Ta warstwa pozwala na znalezienie określonej cechy obrazu poprzez zastosowanie różnych filtrów, zwanymi jądrami.

Filtry są początkowo inicjalizowane losowo. Są to macierze o wymiarach zazwyczaj mniejszych niż wejście obrazu. Początkowe wartości filtrów mogą być również generowane za pomocą różnych metod inicjalizacji jak inicjalizacja Gaussa czy Xaviera.

Aby obliczyć rozmiar wyjścia można zastosować poniższy wzór:

$$\text{Output} = (\text{Input size} - \text{Kernel size}) + 1$$

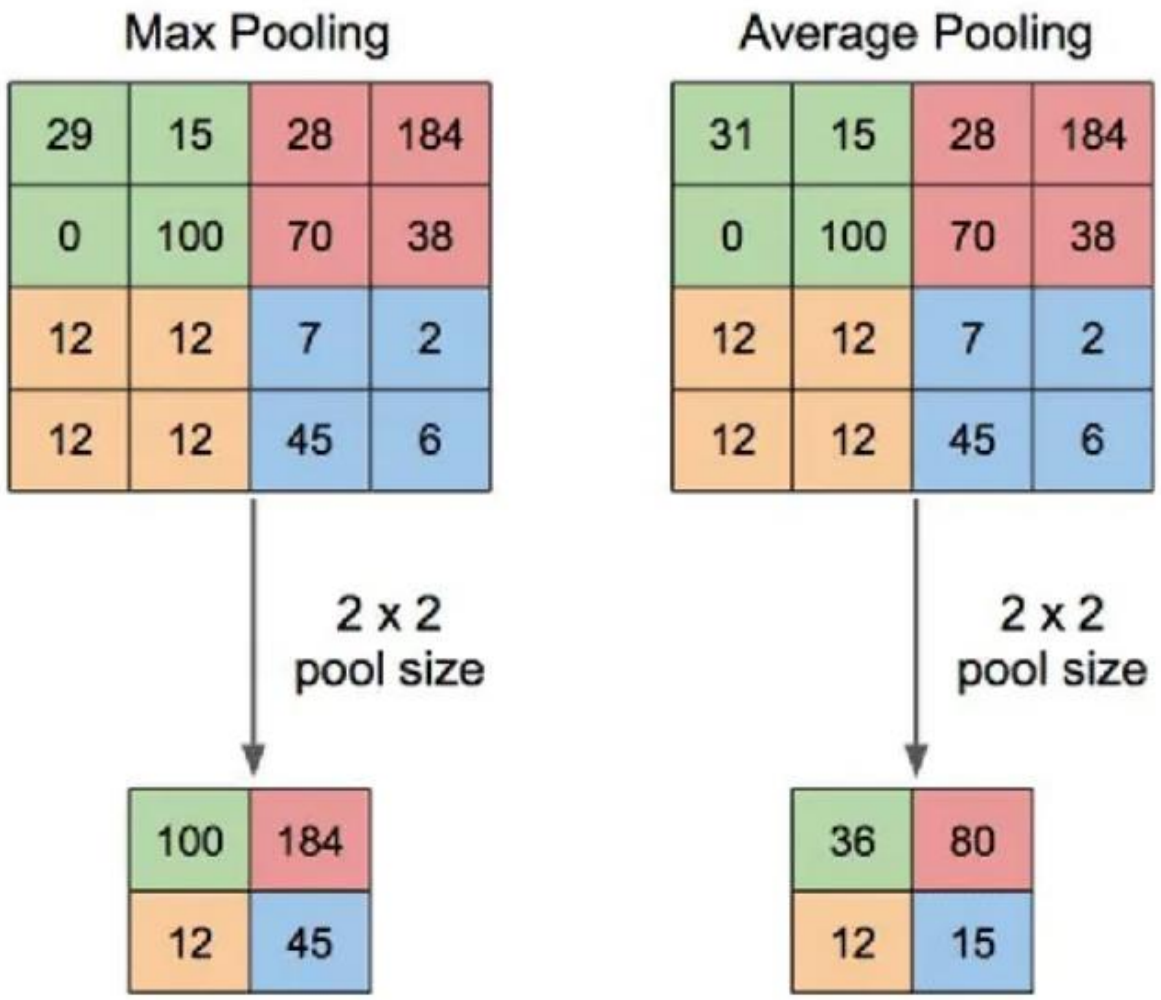


Rysunek 3. Nakładanie filtrów na obraz [2]

Ten hierarchiczny proces ekstrakcji umożliwia sieci rozpoznawanie coraz bardziej złożonych wzorów w miarę przechodzenia informacji przez kolejne warstwy splotowe, co jest niezbędne w zadaniach takich jak klasyfikacja obrazów.

[2] https://sciagaprogramisty.blogspot.com/2018/01/konwolucja-wstep-do-neuronowych-sieci.html (dostęp: 29.04.2024)

Po każdym pojedynczym splocie stosowana jest warstwa łączenia, która grupuje pobliskie wartości w jedną wartość. Pomaga to zmniejszyć wymiary przestrzenne objętości wejściowej, zmniejszając w ten sposób złożoność obliczeniową.



Rysunek 4 Operacja łączenia[3]

Wynik warstwy łączenia są następnie wprowadzane do klasycznej sieci neuronowej, takiej jak na przykład perceptron wielowarstwowy, gdzie są klasyfikowane.

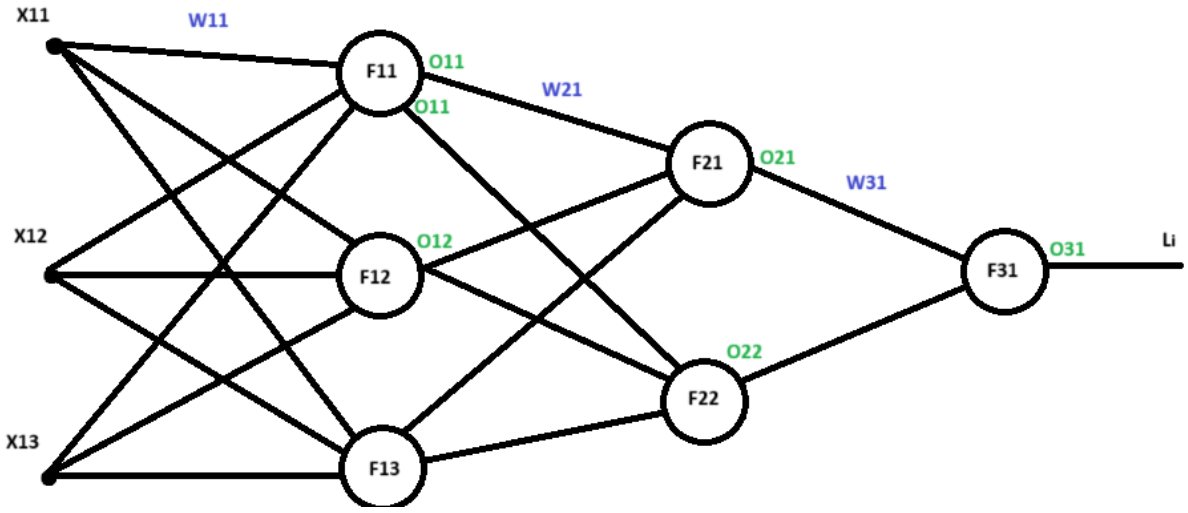
[3] https://bfirst.tech/konwolucyjne-sieci-neuronowe/ (dostęp: 29.04.2024)

2.2. Problem zanikającego gradientu

Problem zanikającego gradientu w sieciach konwulacyjnych jest jednym z głównych wyzwań w uczeniu głębokim, szczególnie gdy sieci te stają się coraz głębsze. Polega on na tym, że w trakcie propagacji wstecznej gradienty błędu maleją wraz z głębokością sieci, co prowadzi do bardzo powolnej lub wręcz zanikającej aktualizacji wag w warstwach początkowych.

Istnieje kilka czynników, które przyczyniają się do zanikania gradientu w CNN. Jednym z głównych jest stosowanie funkcji aktywacji, takiej jak sigmoidalna funkcja logiczna, której pochodna ma małe wartości dla dużych lub małych wartości wejściowych, co prowadzi do malejących gradientów w głębokich warstwach. Innym czynnikiem jest inicjalizacja wag sieci, która może również wpłynąć na to, jak szybko gradienty maleją w trakcie propagacji wstecznej.

Aby lepiej przybliżyć problem zanikającego gradientu rozważmy poniższy rysunek



Gdzie F to funkcja aktywacji, O to wyjście neuronu, a W to waga. W trakcie propagacji wstecznej celem aktualizacji wagi np. W_{11} stosujemy poniższy wzór:

$$W_{11(new)} = W_{11(old)} - \eta \frac{\partial L_i}{\partial W_{11(old)}}$$

- $W_{11(old)}$ – to waga która podlega procesowi uczenia (waga którą aktualizujemy)
- η - to współczynnik uczenia (learning rate)
- $\frac{\partial L_i}{\partial W_{11(old)}}$ – to pochodna cząstkowa funkcji straty względem wagi

Rozważmy równanie $\frac{\partial L_i}{\partial W_{11(old)}}$. Możemy je przekształcić i zapisać w następujący sposób:

$$\frac{\partial L_i}{\partial W_{11}} = \frac{\partial L_i}{\partial O_{31}} * \frac{\partial O_{31}}{\partial W_{11}}$$

Dzięki takiemu zabiegowi jesteśmy w stanie zapisać pochodną wyjścia neuronu 31 względem Wagi 11 tj. $\frac{\partial O_{31}}{\partial W_{11}}$ w taki sposób aby uwzględniać wszystkie ścieżki od warstwy wyjściowej do wagi którą aktualizujemy:

$$\frac{\partial O_{31}}{\partial W_{11}} = \frac{\partial O_{31}}{\partial O_{21}} * \frac{\partial O_{21}}{\partial O_{11}} * \frac{\partial O_{11}}{\partial W_{11}} + \frac{\partial O_{31}}{\partial O_{22}} * \frac{\partial O_{22}}{\partial O_{11}} * \frac{\partial O_{11}}{\partial W_{11}}$$

Ważną kwestią na którą teraz należy zwrócić uwagę jest to, że w powyższym równaniu bierzemy pod uwagę pochodne wyjść neuronów. Co ważne, stosunki pochodnych wyjść neuronów są niczym innym jak pochodną funkcji aktywacji. Wynika to z faktu, że wynik jest wynikiem zastosowania sum funkcji aktywacji.

W związku z tym rozważaniem $\frac{\partial O_{31}}{\partial O_{21}}$ pochodna ta jest niczym innym jak pochodną funkcji aktywacji. Jak zostało to wspomniane wcześniej, najpopularniejszą funkcją aktywacji jest funkcja sigmoidalna, której pochodna osiąga maksymalną wartość 0,25. Przyjmując przykładowe wartości załóżmy, że:

$$\frac{\partial O_{31}}{\partial O_{21}} = 0.2 \qquad \frac{\partial O_{21}}{\partial O_{11}} = 0.1 \qquad \frac{\partial O_{11}}{\partial W_{11}} = 0.05 \qquad \frac{\partial O_{31}}{\partial O_{22}} = 0.1 \qquad \frac{\partial O_{22}}{\partial O_{11}} = 0.05,$$

To $\frac{\partial O_{31}}{\partial W_{11}}$ przyjmie wartość równą 0.00125. Jeżeli teraz wrócimy do wzoru na aktualizację wagi i przyjmiemy że w pierwszym etapie szkolenia sieci waga 11 została przyjęta na wartość 1.5 a learning rate na 0.01 to:

$$W_{11(new)} = W_{11(old)} - \eta\frac{\partial L_i}{\partial W_{11(old)}} = 1.5 - 0.01*0.00125 = 1.4999875$$

Jak można zauważyć nowa waga względem starej wagi praktycznie nie uległa zmianie. Takie zjawisko nazywamy zjawiskiem zanikającego gradientu.

2.3. Zapobieganie problemowi zanikającego gradientu

Aby radzić sobie z problemem zanikającego gradientu, stosuje się inne techniki takie jak:

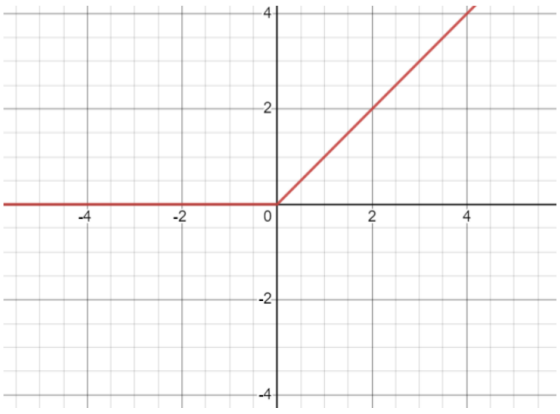
- Użycie funkcji aktywacyjnej ReLU,
- Inicjalizacja wag,
- Normalizacja danych,
- Dodanie połączeń pomijających.

2.3.1. Funkcja aktywacji ReLU

Funkcja aktywacji ReLU jest jedną z najczęściej stosowanych funkcji aktywacji w sieciach neuronowych. Jest to prosta funkcja nieliniowa, która zwraca zero dla wartości ujemnych i równa się wartości wejściowej dla wartości nieujemnej. Matematycznie można ją zdefiniować jako:

$f(x) = \max(0,x)$

Wartości ujemne są przycinane do zera, co sprawia, że funkcja jest niewrażliwa na dużą ilość szumu lub wartości ujemne, co może pomóc w radzeniu sobie z zanikającym gradientem.



Rysunek 5. Wykres funkcji aktywacji

2.3.2. Inicjalizacja wag

Inicjalizacja wag jest jednym z kluczowych kroków w procesie tworzenia i trenowania sieci neuronowej. Polega ona na nadaniu początkowych wartości wagom w sieci, które będą podlegać procesowi uczenia się w celu minimalizacji funkcji kosztu. Poprawa inicjalizacja wag może znaczenie wpłynąć na skuteczność uczenia się sieci neuronowej i szybkości zbieżności do optymalnych rozwiązań jak i zapobieganiu problemowi zanikającego gradientu.

Jedną z popularniejszych metod inicjalizacji wag jest inicjalizacja losowa, w której wagi są początkowo ustawiane na losowe wartości. Jednakże, inicjalizacja losowa może prowadzić do niestabilności lub wolnego procesu uczenia, zwłaszcza w przypadku głębokich sieci neuronowych. Dlatego istnieją bardziej zaawansowane metody inicjalizacji wag, które są bardziej skuteczne w praktyce.

Jedną z tych metod jest inicjalizacja Xaveira. Ta metoda inicjalizuje wagi losowo, ale zgodnie z rozkładem Gaussa o średniej 0 i wariancji zależnej od liczny neuronów w poprzedniej i następnej warstwie. Wagi są skalowane przez pierwiastek z liczby neuronów w poprzedniej warstwie, co pozwala na zachowanie stabilności rozkładu aktywacji w trakcie propagacji w przód i wstecz.

2.3.3. Normalizacja danych

Normalizacja danych jest powszechną techniką stosowaną w sieciach neuronowych w celu zmniejszenia zakresu wartości wejściowych. Ma to na celu ułatwienie uczenia się modelu poprzez zapewnienie bardziej stabilnego i efektywnego procesu optymalizacji. Najczęściej stosowane metody normalizacji to:

- Normalizacja wsadowa
- Normalizacja warstwowa
- Standardowa normalizacja

2.3.4. Połączenia pomijające (skip connecting)

Połączenia pomijające, zwane również połączeniami przeskakującymi, są kluczowym elementem w architekturze głębokich sieci neuronowych. Idea polega na przekazywaniu pewnych informacji z jednej warstwy neuronowej do innej, omijając część warstw pośrednich. W ten sposób można uniknąć utraty informacji w procesie propagacji wstecznej, zwłaszcza w przypadku sieci głębokich, które mogą być podatne na zanikanie gradientu. Połączenia pomijające umożliwiają również skuteczniejsze uczenie się przez sieć neuronową, zapewniając ścieżki, które umożliwiają przekazywanie istotnych informacji bez zbędnego ich powtarzania. Jest to szczególnie istotne w przypadku zadań wymagających modelowań długoterminowych zależności lub w przypadku, gdy istnieją ścieżki alternatywne, które mogą być bardziej efektywne dla danego zadania.

3. Implementacja sieci neuronowej

3.1. Przygotowanie skryptu

Na potrzeby realizacji sieci neuronowej wykorzystano bibliotekę języka Python o nazwie „PyTorch”, która w znaczącym stopniu ułatwia realizację głębokiej sieci neuronowej. Przed przystąpieniem do części głównej nastąpiła normalizacja danych, która odbywa się podczas wczytywania zbioru danych uczących MNIST. Dane te zostały podzielone na dane do uczenia oraz dane do testowania.

3.2. Kod źródłowy

```
import torch
import torch.nn as nn
import torchvision
from matplotlib import pyplot as plt
import torch.nn.functional as F



#liczba epok
n_epochs = 1
#rozmiar partii danych
batch_size = 64
#rozmiar partii danych używany do testowania modelu
batch_size_test = 64
#szybkość aktualizacji parametrów modelu
learning_rate = 0.001
#co ile kroków model wyświetli logi
log_interval = 10
#jak sama nazwa wskazuje random seed
random_seed = 123

out_channels3 = [5,10,25,50,100]
out_channels2 = [20,30,50,100,150]

device = torch.device("cuda" if torch.cuda.is_available() else
torch.device('cpu'))
num_of_devices = torch.cuda.device_count()
print(f"Number of devices is {num_of_devices}")
#print(f"Device name is {torch.cuda.get_device_name()}")

# Ładowanie cyfry MINST

#odchylenie standardowe
std = (0.3081)

#wartość średnia
mu = (0.1307,)

#używam dataloader
```

```
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('/filse/', train=True, download=True,
transform=torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mu,std)
    ])),
    batch_size = batch_size, shuffle = True
)

#test loader
test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('/filse/', train=False, download=True,
transform=torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mu,std)
    ])),
    batch_size = batch_size, shuffle = True
)

#sprawdzam dostęp do danych
data, target = next(iter(train_loader))
print(f"Data shape is {data.shape}")
print(f"Shape of target is {target.shape}")
plt.imshow(next(iter(train_loader))[0][6,0,:,:], cmap='gray')
plt.show()

#implementacja klasyfikatora
class Network(nn.Module):
    def __init__(self, out_channels3, out_channels2):
        super(Network, self).__init__()
        # Pierwsza warstwa konwolucyjna: przekształca obraz 1-kanałowy na
40-kanałowy, z kernel_size=9
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=40,
kernel_size=8)

        # Druga warstwa konwolucyjna: przekształca 40-kanałowy tensor na
out_channels2-kanałowy, z kernel_size=9
        self.conv2 = nn.Conv2d(in_channels=40, out_channels=out_channels2,
kernel_size=8)

        # Warstwa dropout: zapobiega przeuczeniu, losowo wyłączając neurony
        self.conv2_drop = nn.Dropout2d()

        # Obliczenie rozmiaru wejścia dla pierwszej warstwy liniowej
        self._calculate_linear_input_size()

        # Pierwsza warstwa liniowa: przekształca tensor o rozmiarze
`linear1_input_size` na tensor o rozmiarze `out_channels3`
        self.linear1 = nn.Linear(self.linear1_input_size, out_channels3)

        # Druga warstwa liniowa: przekształca tensor o rozmiarze
`out_channels3` na tensor o rozmiarze 10 (klasyfikacja na 10 klas)
        self.linear2 = nn.Linear(out_channels3, 10)

    def _calculate_linear_input_size(self):
        output_size = ((28 - 8) // 2 - 8) // 2  # Obliczenie rozmiaru
tensora po dwóch operacjach max-pooling
        self.linear1_input_size = self.conv2.out_channels * output_size *
output_size

    # funkcja przetwarzania danych w przód
```

```
    def forward(self, x):
        # Przepuszczenie danych przez pierwszą warstwę konwolucyjną
        x = self.conv1(x)

        # Max-pooling (zmniejszenie rozmiaru tensora)
        x = F.max_pool2d(x, kernel_size=2)

        # Funkcja aktywacji ReLU
        x = F.relu(x)

        # Przepuszczenie danych przez drugą warstwę konwolucyjną
        x = self.conv2(x)

        # Max-pooling (zmniejszenie rozmiaru tensora)
        x = F.max_pool2d(x, kernel_size=2)

        # Dropout (losowe wyłączenie 50% neuronów)
        x = F.dropout(x, p=0.5, training=self.training)

        # Spłaszczenie tensora przed podaniem go do warstwy liniowej
        #[batch_size, num_channels, height, width] > [batch_size,
num_features]
        x = x.view(-1, self.linear1_input_size)

        # Przepuszczenie danych przez pierwszą warstwę liniową
        x = self.linear1(x)

        # Funkcja aktywacji ReLU
        x = F.relu(x)

        # Dropout (losowe wyłączenie 50% neuronów)
        x = F.dropout(x, p=0.5, training=self.training)

        # Przepuszczenie danych przez drugą warstwę liniową
        x = self.linear2(x)

        # Log-softmax (wyjście w formie logarytmu prawdopodobieństwa)
        return F.log_softmax(x, dim=1)


a = next(iter(train_loader))
print(a[0].shape)
print(type(a[0]))


#do wizualizacji postępów uczenia
train_loss = []
train_counter = []
test_loss = []
test_counter = [i*len(train_loader.dataset) for i in range(n_epochs + 1)]


#trening
def training_single_epoch(epoch, model, optimizer, train_loader, loss_fcn):
    #ustawiam model w tym do uczenia jest to niezbędne aby dropout działał
    model.train()
    for batch_ind, (data, target) in enumerate(train_loader):
        #zeruje gradienty z poprzedniej iteracji
        optimizer.zero_grad()

        # przetwarzanie do przodu, przekazuje dane przez model, generując
```

```
przewidywania
        output = model(data)
        #oblicza stratę między przewidywanymi a rzeczywistymi wartościami.
        loss = F.nll_loss(output, target)

        # propagacja do tyłu, oblicza gradienty błędu względem parametrów
modelu
        loss.backward()

        # uaktualnienie parametrów
        optimizer.step()

        #jeżeli index batcha jest wielokrotnością log_interval to wyświetla
informacje o bieżącej epoce, postępie w danych treningowych oraz wartości
straty.
        if batch_ind % log_interval == 0:
            print("Train Epoch: {}[{}/{} ({:.0f}%)]\tLoss: {:.6f}]".format(
                epoch, batch_ind * len(data), len(train_loader.dataset),
                100. * batch_ind / len(train_loader), loss.item()
            ))


# testowanie
def test():
    #ustawiam model w tryb testowania bo inaczej dropout nie zadziała
    model.eval()
    #inicjalizacja zmiennych do zapisu całkowitej straty testowej i liczby
poprawnych przewidywań
    test_loss = 0
    correct = 0
    test_losses = []

    with torch.no_grad():
        #iteracja przez wszystkie serie danych z test_loader
        for data, target in test_loader:
            #przetwarzanie danych w przód, przekazuje dane przez model
generujac przewidywania
            output = model(data)
            #dodawanie wartości staraty z obecnej serii do całkowitej
straty testoej
            test_loss += F.nll_loss(output, target).item()

            #obliczam poprawne przewidywania olbiczając liczbę poprawnych
przewidywań w bieżącej serii
            pred = output.data.max(dim = 1, keepdim = True)[1]

            # suma poprawnych predykcji czyli przewidywań
            correct += pred.eq(target.data.view_as(pred)).sum()

    test_loss /= len(test_loader.dataset)
    test_losses.append(test_loss)

    print("\nTest set: Acg. loss: {:.4f}, Accuracy: {}/{}
({:.0f}%)\n".format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)
    ))

    # obliczanie dokładności dla całego zbioru testowego
    accuracy = 100. * correct / len(test_loader.dataset)
    # zwracanie dokładności i straty testowej
```

```
    return test_loss, accuracy

# trenujemy nasz model


results = []


# Testowanie modelu dla każdej kombinacji parametrów
#for channels1, channels2 in zip(out_channels1, out_channels2):
# Testowanie modelu dla każdej kombinacji parametrów
for channels3 in out_channels3:
    for channels2 in out_channels2:
        model = Network(channels3, channels2)
        optimizer = torch.optim.Adam(params=model.parameters(),
lr=learning_rate)

        print(f"\nTesting model with out_channels1={channels3} and
out_channels2={channels2}")
        test_accuracy = []
        for epoch in range(1, n_epochs + 1):
            training_single_epoch(epoch, model=model, optimizer=optimizer,
train_loader=train_loader, loss_fcn=F.nll_loss)
            test_loss, accuracy = test()  # pobranie dokładności i straty
testowej
            test_accuracy.append(accuracy)  # zapisanie dokładności w
liście
            print(f"Test Loss: {test_loss}, Accuracy: {accuracy}%")
        results.append((channels3, channels2, test_accuracy))

data, label = next(iter(test_loader))
with torch.no_grad():
    output = model(data)
    fig = plt.figure()
    for i in range(6):
        plt.subplot(2,3,i+1)
        plt.tight_layout()
        plt.imshow(torch.squeeze(data[i][0]), cmap = 'gray', interpolation
= 'none')
        plt.title("prediction: {}".format(
            output.data.max(1, keepdim = True)[1][i].item()
        ))
        plt.xticks([])
        plt.yticks([])
        plt.show()


# Tworzenie wykresu 3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Pobranie wyników treningu
for result in results:
    channels3, channels2, accuracy = result
    # Dodanie punktów do wykresu
    ax.plot([channels3], [channels2], [accuracy[-1]], marker='o',
color='r')

# Ustawienie etykiet osi
ax.set_xlabel('out_channels3')
ax.set_ylabel('out_channels2')
```

```
ax.set_zlabel('Accuracy')

# Zmiana punktów na wykresie na powierzchnię
X = [result[0] for result in results]
Y = [result[1] for result in results]
Z = [result[2][-1] for result in results]
ax.plot_trisurf(X, Y, Z, cmap='viridis', edgecolor='none')

plt.show()
```

4. Eksperymenty

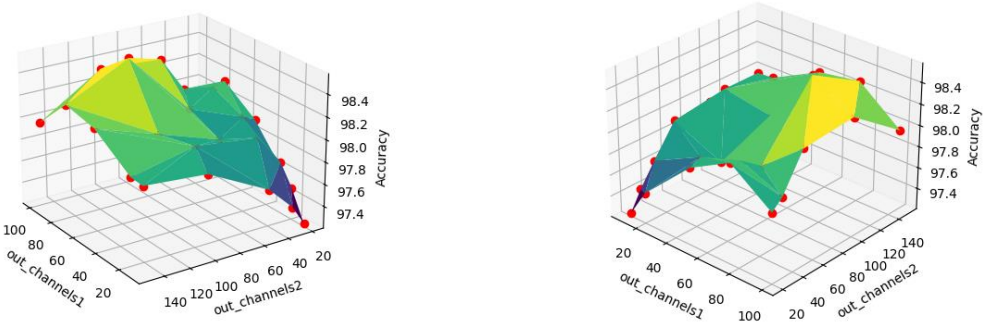
Celem eksperymentów było znalezienie odpowiedniej konfiguracji parametrów uczenia, dla których sieć będzie osiągała najlepsze wyniki. Eksperymenty zostały podzielone na dwa etapy.

4.1. Etap I

Celem pierwszego etapu było znalezienie odpowiednich parametrów uczenia takich jak dropout, liczba neuronów na wyjściu pierwszej warstwy sieci konwulacyjnej oraz liczba neuronów na wyjściu drugiej warstwy sieci konwulacyjnej.

4.1.1 Eksperyment I.I

W pierwszym eksperymencie pierwszego etapu została przetestowana ilość neuronów w każdej z warstw sieci konwulacyjnych. Na potrzeby tego eksperymentu dropout został ustalony na 50%, zaś learning rate na 0.001. Parament kernel size został ustawiony na 5, a liczba epok wynosiła 10. Liczba testowanych neuronów na wyjściu pierwszej warstwy sieci konwulacyjnej wynosiła kolejno 10,20,50,75,100, a na wyjściu drugiej warstwy sieci konwulacyjnej wynosiła 20,30,50,100,150. Dodatkowo w eksperymencie ustawiono stałą wartość wyjścia w pierwszej warstwie liniowej i wynosi ona dokładnie 50 neuronów.

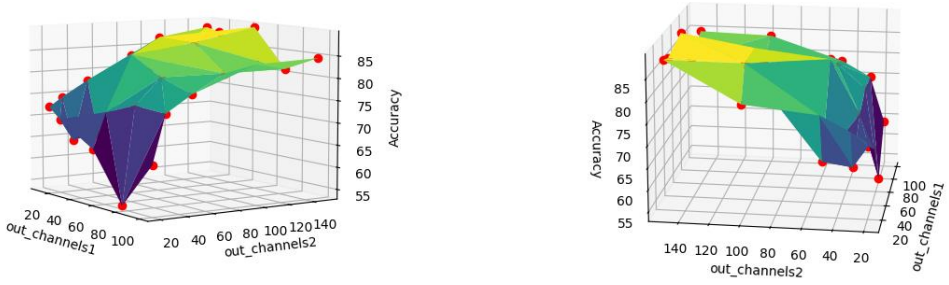


Rysunek 6. Wykresy poprawności klasyfikacji dla eksperymentu I.I.

Analizując otrzymane wyniki, można zauważyć wysoką poprawność klasyfikacji dla każdej konfiguracji parametrów. Przedział poprawności wyników wahał się pomiędzy 97.5% a 98%. Nawet dla małej ilości neuronów w pierwszej i drugiej warstwie konwulacyjnej wynik był bardzo satysfakcjonującym wynikiem.

4.1.2 Eksperyment I.II

W drugim eksperymencie zostały wykorzystane te same wartości liczb neuronów w każdej warstwie co w eksperymencie I.I, jednakże został zmieniony dropout na 90% oraz kernel size na 8. Learning rate oraz liczba neuronów w drugiej warstwie nie uległa zmianie.

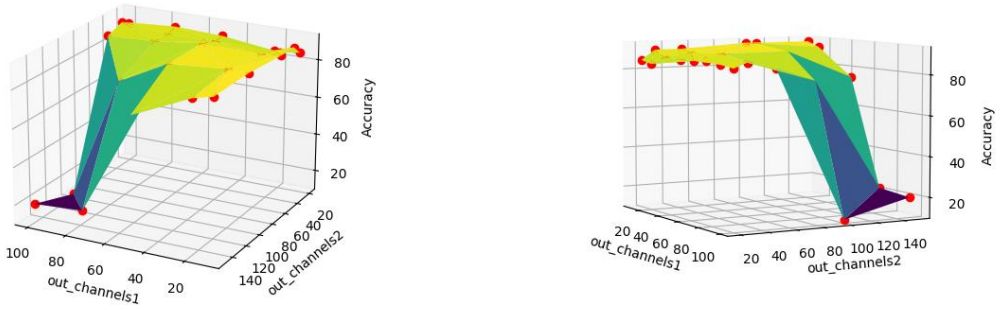


Rysunek 7. Wykres poprawności klasyfikacji dla eksperymentu I.II.

Jak widać na wykresie, dla takich paramentów uczenia, klasyfikator poradził sobie gorzej i osiągnął najwyższą skuteczność około 88% zaś najniższa skuteczność wynosiła około 55%. Sieć poradziła sobie najlepiej z klasyfikacją obrazu dla 50 neuronów na wyjściu w pierwszej warstwie konwulacyjnej oraz 150 neuronów na wyjściu w drugiej warstwie.

4.1.3 Eksperyment I.III

Celem tego eksperymentu było sprawdzenie, czy parametr kernel size ma duży wpływ na poprawność klasyfikacji. W tym celu parametry pierwszej i drugiej warstwy konwulacyjnej nieuległy zmianie. Modyfikacji nie uległy również parametry takie jak dropout, który nadal wynosił 90% oraz learning rate. Natomiast parametr kernel size został ustawiony na taką samą wartość jak w eksperymencie I.I, czyli 2.



Rysunek 8 Wykres poprawności klasyfikacji w eksperymencie I.III

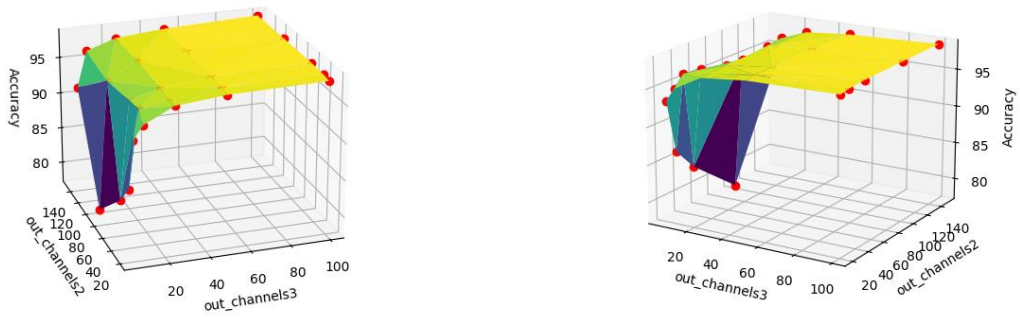
Analizując wykresy przeprowadzonego eksperymentu można zauważyć, że duża wartość parametru dropout oraz mała wartość parametru kernel size, powoduje bardzo dużą rozbieżność w wynikach. Dodatkowo można również zauważyć, że wyższe ilości neuronów w warstwach konwulacyjnych niekoniecznie wpływają na skuteczność klasyfikacji, gdyż najwyższe wyniki zostały osiągnięte dla najmniejszych ilość neuronów. Powodem tak niskiej skuteczności może być zbyt mała ilość epok na jakich model został przeszkolony.

4.2. Etap II

W tym etapie celem było odnalezienie odpowiednich parametrów uczenia dla liczby neuronów na wyjściu w pierwszej warstwie liniowej oraz innych parametrów takich jak kernel size, czy dropout. Do celów testowych liczbę neuronów na wyjściu pierwszej warstwy konwulacyjnej ustawiono na 40.

4.2.1 Eksperyment II.I

W pierwszym eksperymencie drugiego etapu learning rate został ustawiony na 0.001, droput na domyślne 50%, zaś kernel size na 5. Liczba epok na jakich sieć się uczyła wynosiła 10. Liczba neuronów na wyjściu drugiej warstwy konwulacyjnej wynosiła kolejno 20,30,50,100,150, zaś liczba neuronów na wyjściu pierwszej warstwy liniowej wynosiła 5,10,25,50,100.

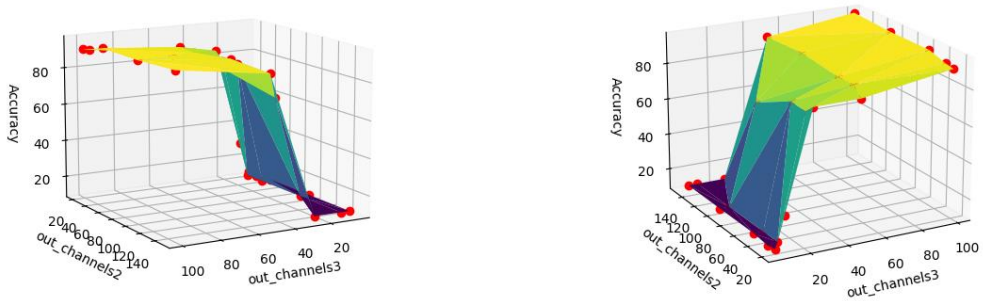


Rysunek 9 Wykres poprawności klasyfikacji w eksperymencie II.I

Dzięki wykresom można zauważyć, sieć osiągnęła skuteczność poniżej 90% dla wartości mniejszych niż 40 neuronów na wyjściu pierwszej warstwy liniowej. Dla każdej wyższej wartości, niezależnie od wartości liczby neuronów na wyjściu drugiej warstwy konwulacyjnej, sieć osiąga satysfakcjonujące wyniki powyżej 90%. Może to być spowodowaną zbyt małą liczbą epok.

4.2.2 Eksperyment II.II

Kolejny eksperyment miał na celu testowanie sieci dla zwiększonego dropoutu. Wartość tego parametru została ustawiona na 90%. Żaden inny parametr nie uległ zmianie.

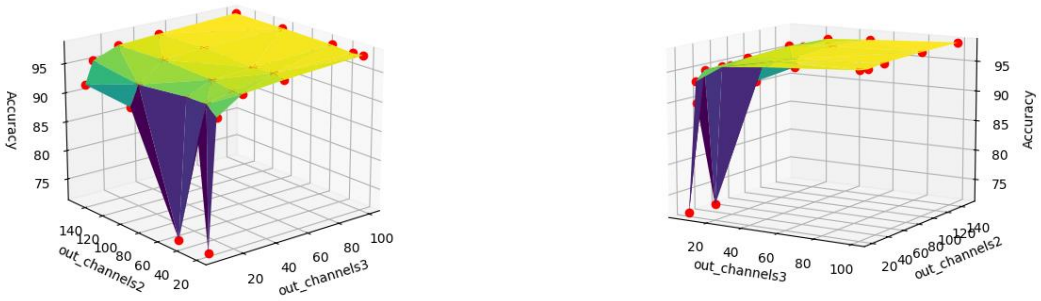


Rysunek 10 Wykres poprawności klasyfikacji w eksperymencie II.II

Otrzymane wykresy są zbliżonego kształtu to wykresów z poprzedniego eksperymentu. Wyniki jednak znacząco się różnią. Jak można zauważyć, dla liczby neuronów na wyjściu pierwszej warstwy liniowej mniejszej od 20, skuteczność wynosi około 11%. Sytuacja jednak ulega znaczącej zmianie dla większych wartości neuronów w tej warstwie. Skuteczność osiąga wtedy wartości rzędu 90%.

4.2.3 Eksperyment II.III

Celem ostatniego eksperymentu było przetestowanie parametru kernel size. W tym celu wartość dropoutu została zmieniona na 50%, gdyż przy ten wartości sieć osiągnęła najlepsze efekty szkolenia. Wartość kernel size została ustawiona na 2.

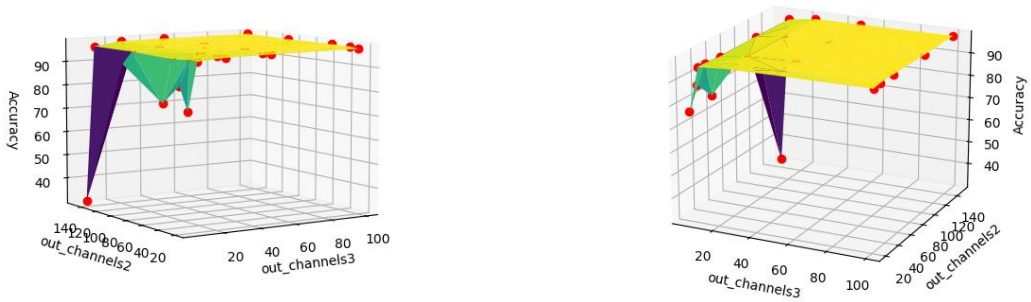


Rysunek 11 Wykres poprawności klasyfikacji w eksperymencie II.III

Analizując otrzymane wykresy, można zauważyć że sieć osiąga znacznie lepsze wyniki dla mniejszych wartości liczby neuronów na wyjściu pierwszej warstwy liniowej niż dla eksperymentu II.I gdzie kernel size wynosił 5. Dla wyższych wartości liczb neuronów w warstwach parametr ten jednak niema większego znaczenia, gdyż wyniki wychodzą podone.

4.2.3 Eksperyment II.IV

W celu dokładniejszej analizy parametry kernel size wykonano dodatkowy eksperyment w którym ustawiono ten parament na wartość 9. Wszystkie inne parametry nie uległy zmianie.



Rysunek 12 Wykres poprawności klasyfikacji w eksperymencie II.III

W odróżnieniu od eksperymentu I.III oraz I.I w tym przypadku osiągnęto najgorsze wyniki dla niskiej wartości liczby neuronów na pierwszej warstwie liniowej i dużej liczby neuronów na wyjściu drugiej warstwy konwulacyjnej. Dla mniejszej ilości neuronów na wyjściu drugiej warstwy konwulacyjnej i na wyjściu pierwszej warstwy liniowej wyniki osiągają znacznie lepsze efekty. Natomiast w pozostałych przypadkach efektywność szkolenia sieci jest satysfakcjonująca na poziomie przekraczającym 95%.

5. Wnioski

Projekt polegał na realizacji głębokiej sieci neuronowej której zadaniem było rozpoznawanie cyfr pisanych od ręcznie za pomocą zbioru danych MNIST. Skrypt został napisany w języku programowania Python z wykorzystaniem biblioteki PyTorch. Program był wielokrotnie modyfikowany na potrzeby przeprowadzonych eksperymentów.

Na podstawie przeprowadzonych eksperymentów można było dostrzec wiele różnych zachowań sieci w zależności od dobranych parametrów. Zauważono, że liczba neuronów w poszczególnych ma wpływ na szybkość uczenia sieci neuronowej. Im więcej jest neuronów w każdej z warstw, tym sieć uczyła się wolniej. W związku z tym faktem zrezygnowano z dużej ilość epok oraz z dużej ilości neuronów w każdej warstwie.

Znaczny wpływ na poprawność klasyfikacji miały także parametry takiej jak kernel size, czy dropout. W przypadku mniejszej liczby neuronów w warstwach, parametr dropoutu ustawiony na wyższe wartości, powodował słabe wyniki sieci.

Analizując wyniki testów przeprowadzonych w II etapie, można zauważyć, że niezależnie od dobranych parametrów uczenia oraz liczby neuronów wyjściu drugiej warstwy konwulacyjnej, najlepsza skuteczność uczenia jest osiągana, gdy liczba neuronów na wyjściu pierwszej warstwy liniowej przyjmuje wartości większe od 60 neuronów. Może to być jednak spowodowane zbyt małą liczbą epok na jakiej model był trenowany.

Na podstawie otrzymanych wyników klasyfikacji w każdym z eksperymentów można postawić wniosek, że zbiór danych MNIST jest bardzo dobrym zbiorem do testowania głębokiej sieci neuronowej. W bardzo łatwy sposób można było przetestować większość parametrów uczenia oraz uzyskać bardzo satysfakcjonujące wyniki rzędu nawet 98,5%.

Źródła

https://pdf.helion.pl/delet2/delet2.pdf

https://www.analyticsvidhya.com/blog/2021/08/all-you-need-to-know-about-skip-connections/

http://krzysztof.halawa.staff.iiar.pwr.wroc.pl/SieciNeuronowe2.pdf

https://datascience.eu/pl/uczenie-maszynowe/relu-funkcja-aktywujaca/

https://en.wikipedia.org/wiki/Batch_normalization

https://medium.com/@a01642207/convolutional-neural-networks-for-image-classification-461306f4e7f9