lab06-sieci-neuronowe

May 14, 2025

1 Sieci neuronowe

1.1 Perceptron

1.1.1 Schemat uczenia perceptronu

Dopóki nie zostanie spełniony warunek stopu (np. wykonana zostanie pewna ustalona z góry liczba rund) dla każdego przykładu (x, y) wykonaj:

$$w = w + a(y - f(w \cdot x))x$$

1.1.2 Objaśnienia do algorytmu

x – wektor wejściowy

y – oczekiwane wyjście dla wektora x

w – wektor wag

a – stała ucząca

f – funkcja aktywacji

1.2 Sieci wielowarstwowe

1.2.1 Algorytm propagacji wstecznej

Dopóki nie zostanie spełniony warunek stopu dla każdego przykładu (x, y) wykonaj:

1. dla każdego wierzchołka j w warstwie wejściowej $a[j] = x[j]$
2. dla każdej warstwy l od 2 do $liczba\_warstw$
    1. dla każdego wierzchołka i w warstwie l
        1. dla każdej krawędzi z j do i
           $sum[i] += w[j, i] * a[j]$
        2. $a[i] = g(sum[i])$
3. dla każdego wierzchołka i warstwy wyjściowej

   $d[i] = g'(sum[i]) * (y[i] - a[i])$
4. dla każdej warstwy l od $liczba\_warstw - 1$ do 1
    1. dla każdego wierzchołka j w warstwie l

1. $sum = 0$
2. dla każdej krawędzi z j do i
   $sum+ = w[j,i] * d[i]$
3. $d[j] = g'(sum[j]) * sum$
4. dla każdego wierzchołka i w warstwie $l+1$
   $w[j,i]+ = a * a[j] * d[i]$

1.2.2 Objaśnienia do algorytmu

$w[i,j]$ – macierz wag

$a[i]$ – wartości funkcji aktywacji (poza warstwą wejściową, gdzie a[i]=x[i])

$d[i]$ – „delta" (propagowany błąd)

a – stała ucząca

g – funkcja aktywacji (np. $\frac{1}{1+e^{-x}}$)

g' – pochodna g

2 Przykład

Zastosujemy algorytm propagacji wstecznej (jego wariant Stochastic Gradient Descent) do wytrenowania sieci neuronowej, której zadaniem będzie rozpoznawanie odręcznie zapisanych cyfr. Do uczenia i testowania naszej sieci wykorzystamy bazę danych MNIST, która zawiera zdjęcia (w skali szarości) odręcznie zapisanych cyfr.

W procesie uczenia skorzystamy z biblioteki PyTorch.

[1]:
```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import Compose, Lambda, ToTensor
import matplotlib.pyplot as plt
```

Na początek ustalmy ziarno dla generatora liczb pseudolosowych, żeby wyniki były w pełni odtwarzalne.

[2]:
```
torch.manual_seed(42);
```

Jeżeli dysponują Państwo kartą graficzną kompatybilną z biblioteką CUDA, to warto przeprowadzić uczenie z wykorzystaniem procesora karty graficznej. Czas uczenia będzie wtedy o wiele krótszy.

[3]:
```
device = torch.device('cuda') if torch.cuda.is_available() else torch.
↪device('cpu')
```

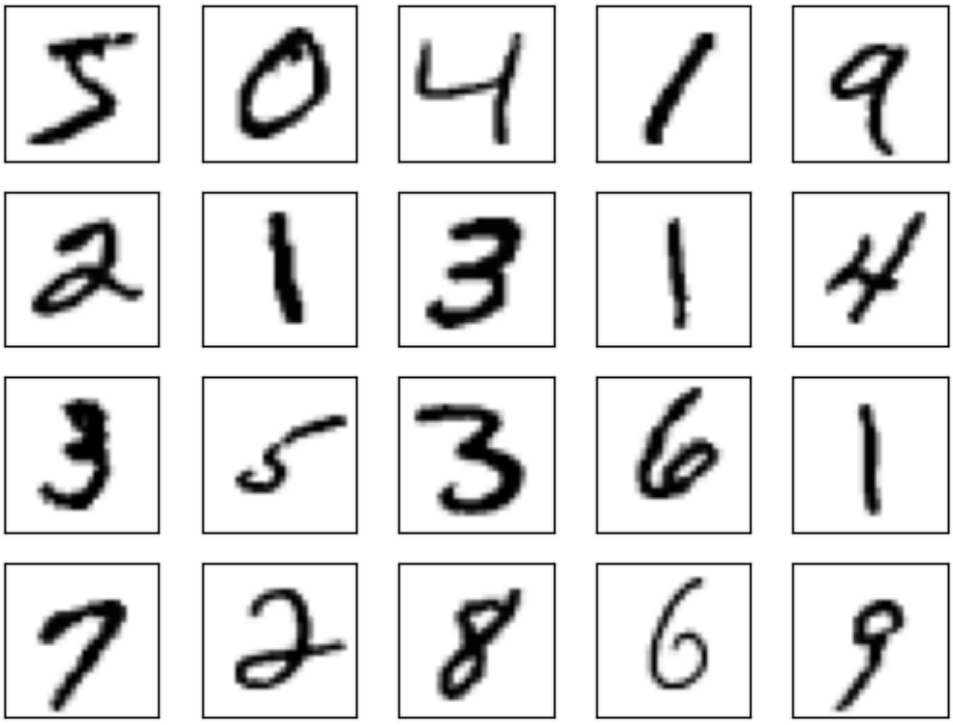
Wczytajmy zbiory danych (uczący i testowy), korzystając z następujących wywołań

[4]:
```
trainset = datasets.MNIST('data', train=True, download=True,
                          transform=Compose([ToTensor(),
```

```
                                                Lambda(lambda x: x.flatten())]))
testset = datasets.MNIST('data', train=False, download=True,
                         transform=Compose([ToTensor(),
                                            Lambda(lambda x: x.flatten())]))
```

Celem ilustracji wyświetlmy zdjęcia znajdujące się na początku zbioru uczącego.

[5]:
```
for i in range(1, 21):
    plt.subplot(4, 5, i)
    plt.xticks([], [])
    plt.yticks([], [])
    plt.imshow(trainset.data[i - 1], cmap='gray_r')
```



Oczekiwane odpowiedzi możemy odczytać z pola `targets` zbioru uczącego.

[6]:
```
trainset.targets[:20]
```

[6]:
```
tensor([5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5, 3, 6, 1, 7, 2, 8, 6, 9])
```

Do uczenia sieci wykorzystamy następującą funkcję

[7]:
```
def train(model, dataset, n_iter=100, batch_size=256):
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
    criterion = nn.NLLLoss()
    dl = DataLoader(dataset, batch_size=batch_size)
```

```
    model.train()

    for epoch in range(n_iter):
        for images, targets in dl:
            optimizer.zero_grad()
            out = model(images.to(device))
            loss = criterion(out, targets.to(device))
            loss.backward()
            optimizer.step()

        if epoch % 10 == 0:
            print('epoch: %3d loss: %.4f' % (epoch, loss))
```

Dokładność sieci będziemy mierzyć następującą funkcją

[8]:
```
def accuracy(model, dataset):
    model.eval()
    correct = sum([(model(images.to(device)).argmax(dim=1) == targets.
 ↪to(device)).sum()
                   for images, targets in DataLoader(dataset, batch_size=256)])
    return correct.float() / len(dataset)
```

Na początek zbudujmy i wyuczmy sieć złożoną z pojedynczej warstwy neuronów.

[9]:
```
model = nn.Sequential(
    nn.Linear(28 * 28, 10),
    nn.LogSoftmax(dim=-1)
).to(device)

train(model, trainset)
```

```
epoch:   0 loss: 1.2577
epoch:  10 loss: 0.5092
epoch:  20 loss: 0.4251
epoch:  30 loss: 0.3898
epoch:  40 loss: 0.3697
epoch:  50 loss: 0.3564
epoch:  60 loss: 0.3467
epoch:  70 loss: 0.3391
epoch:  80 loss: 0.3329
epoch:  90 loss: 0.3277
```

Sprawdźmy jaka jest skuteczność wytrenowanej sieci na zbiorze testowym.

[10]:
```
accuracy(model, testset)
```

[10]:
```
tensor(0.9172)
```

Wprowadźmy w naszej sieci warstwę ukrytą złożoną z 300 neuronów, tworząc w ten sposób perceptron wielowarstwowy (ang. MLP – Multilayer Perceptron).

[11]:
```
hidden_size = 300

model = nn.Sequential(
    nn.Linear(28 * 28, hidden_size),
    nn.ReLU(),
    nn.Linear(hidden_size, 10),
    nn.LogSoftmax(dim=-1)
).to(device)

train(model, trainset)
```

```
epoch:   0 loss: 1.7508
epoch:  10 loss: 0.4128
epoch:  20 loss: 0.3393
epoch:  30 loss: 0.3045
epoch:  40 loss: 0.2791
epoch:  50 loss: 0.2593
epoch:  60 loss: 0.2433
epoch:  70 loss: 0.2299
epoch:  80 loss: 0.2186
epoch:  90 loss: 0.2089
```

Sprawdźmy skuteczność perceptronu wielowarstwowego na zbiorze testowym.

[12]:
```
accuracy(model, testset)
```

[12]:
```
tensor(0.9543)
```

Całkiem nieźle, ale sporo jeszcze można poprawić (por. wyniki ze strony MNIST).

3 Zadanie

1. Przeanalizować jak wielkość zbioru uczącego wpływa na skuteczność klasyfikatora na zbiorze testowym. Wykorzystać odpowiednio 10%, 25%, 50% i 100% przykładów ze zbioru uczącego.
2. Sprawdzić jak na skuteczność klasyfikatora wpływa wielkość warstwy ukrytej (zmienna `hidden_size`).

3. Sprawdzić jak na skuteczność klasyfikatora oraz czas trwania procesu uczenia wpłynie wprowadzenie kolejnych warstw ukrytych.
4. Przeanalizować wpływ liczby iteracji (parametr `n_iter` funkcji `train`) na skuteczność sieci.
5. Sprawdzić jak na skuteczność perceptronu wielowarstwowego wpłynie pominięcie funkcji aktywacji (`nn.ReLU`) w definicji modelu.
6. Powtórzyć powyższe eksperymenty dla wybranego (MNIST-like, ale nie MNIST) zbioru danych ze strony: https://www.simonwenkel.com/lists/datasets/list-of-mnist-like-datasets.html
7. Zrealizować kurs e-learningowy dotyczący budowania klasyfikatorów (gdy zostanie udostępniony na wykładzie).