SPRAWOZDANIE

Zajęcia: Uczenie Maszynowe
Prowadzący: prof. dr hab. Vasyl Martsenyuk

Laboratorium Nr 4 Data 11.01.2025 Temat: Implementacja algorytmów optymalizacji gradientowej do trenowania modeli, Projektowanie i trening prostych sieci neuronowych w TensorFlow lub PyTorch,Zastosowanie konwolucyjnych sieci neuronowych (CNN) do analizy obrazu Wariant drugi (2)	Bartosz Bieniek Informatyka II stopień, stacjonarne, 1 semestr, gr.A

1. Polecenie:

1. Zrealizuj w Pythonie optymalizację funkcji metodą spadku gradientu wraz z wizualizacją. Wariant drugi, funkcja: f(x) = |x| + x^2 metodą spadku gradientu i wizualizacja procesu.

2. Zrealizuj w Pythonie najprostszą sieć neuronową wraz z ewaluacją i prognozowaniem. Wariant drugi, temat: sieć neuronowa do klasyfikacji binarnej.

3. Zrealizuj projektowanie, trenowanie i testowanie sieci konwolucyjnej na podstawie jednego z dostępnych w Pythonie podstawowych zbiorów danych. Wariant drugi, temat: zaprojektuj, wytrenuj i przetestuj sieć konwolucyjną na zbiorze CIFAR-10.

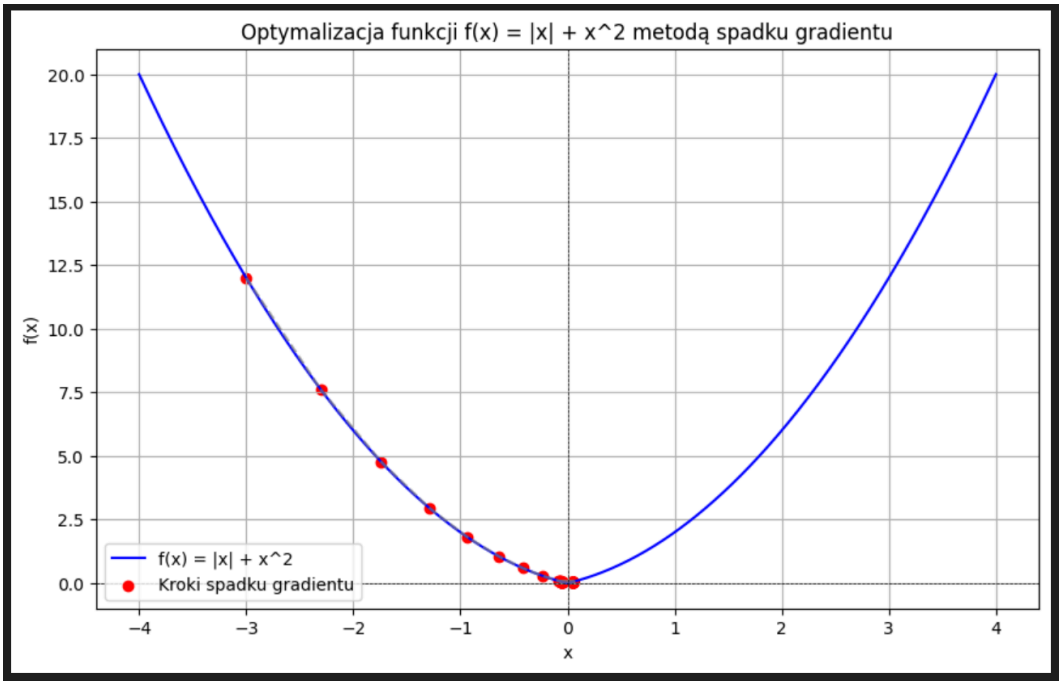
2. Opis programu opracowanego [Repozytorium Github]

Optymalizacja Funkcji

```python
# 1. Zrealizuj w Pythonie optymalizację funkcji metodą spadku gradientu wraz z wizualizacją.
# Wariant drugi, funkcja: f(x) = |x| + x^2 metoda  spadku gradientu i wizualizacja procesu.

import numpy as np
import matplotlib.pyplot as plt

def f(x): # Funkcja celu
    return np.abs(x) + x**2

def gradient(x): # Gradient funkcji celu
    return 1 + 2*x if x > 0 else -1 + 2*x

def gradient_descent(start_x, learning_rate, tolerance, max_iters): # Spadek gradientu
    x = start_x
    history = [x]
    for _ in range(max_iters):
        grad = gradient(x)
        new_x = x - learning_rate * grad
        history.append(new_x)
        if abs(new_x - x) < tolerance: # Sprawdzanie warunku stopu
            break
        x = new_x
    return x, history

# Parametry algorytmu
start_x = -3.0          # Punkt początkowy
learning_rate = 0.1     # Krok uczenia
tolerance = 1e-6        # Tolerancja
max_iters = 100         # Maksymalna liczba iteracji

optimal_x, history = gradient_descent(start_x, learning_rate, tolerance, max_iters) # Optymalizacja

# Wizualizacja
x_vals = np.linspace(-4, 4, 500)
y_vals = f(x_vals)
plt.figure(figsize=(10, 6))
plt.plot(x_vals, y_vals, label='f(x) = |x| + x^2', color='blue')
plt.scatter(history, [f(x) for x in history], color='red', label='Kroki spadku gradientu')
plt.plot(history, [f(x) for x in history], linestyle='--', color='gray', alpha=0.7)
plt.title('Optymalizacja funkcji f(x) = |x| + x^2 metodą spadku gradientu')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.axhline(0, color='black', linewidth=0.5, linestyle='--')
plt.axvline(0, color='black', linewidth=0.5, linestyle='--')
plt.legend()
plt.grid()
plt.show()

print(f"Wartość optymalna x: {optimal_x}")
print(f"Wartość funkcji w minimum f(x): {f(optimal_x)}")
```

Rys. 1. Program zadania pierwszego, optymalizacja funkcji $f(x)=|x|+x^2$.

Przedstawiony na rysunku pierwszym program implementuje metodę spadku gradientu, wykorzystując gradient funkcji $f(x)=|x|+x^2$. Następnie rysuje wykres funkcji i pokazuje kroki optymalizacji jako punkty na wykresie. Pozwala on dostosować parametry, takie jak punkt początkowy, szybkość uczenia, maksymalną liczbę iteracji lub tolerancję.



Rys. 2. Wynik działania programu.

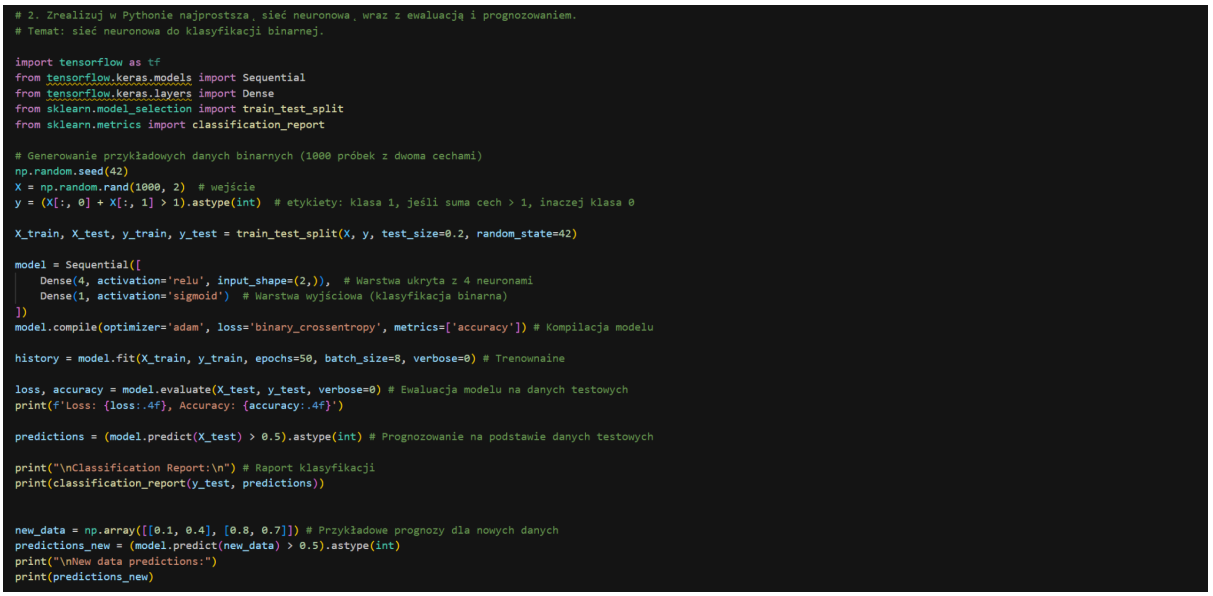
Wartość optymalna x: -0.055555555593980344
Wartość funkcji w minimum f(x): 0.05864197535133618

W analizie uzyskanych wyników zauważono, że proces optymalizacji funkcji $f(x)=|x|+x^2$ metodą spadku gradientu, skutkował stopniowym zmniejszaniem się wartości funkcji w kolejnych iteracjach. Punkty oznaczone na wykresie jako kroki spadku gradientu ukazywały zmniejszającą się odległość między kolejnymi iteracjami, co wskazywało na zbliżanie się do minimum.

Funkcja charakteryzuje się punktem minimum w pobliżu x=0, co potwierdziły zarówno wyniki optymalizacji, jak i kształt wykresu. Zastosowany algorytm skutecznie zbiegł do rozwiązania przy wybranych parametrach, takich jak krok uczenia i tolerancja. Podczas optymalizacji zaobserwowano szybkie zmniejszanie wartości funkcji na początku iteracji, a następnie wolniejsze zbliżanie się do optymalnego punktu w pobliżu minimum.

Na podstawie wizualizacji zauważono także, że wybór punktu początkowego po lewej stronie osi (dla $x=-3x$) wpłynął na kierunek optymalizacji, w którym proces stopniowo zmierzał do środka wykresu. Algorytm zbiegał się zgodnie z oczekiwaniami i poprawnie uwzględniał różniczkowalne i nieróżniczkowalne fragmenty funkcji f(x).

Realizacja sieci neuronowej wraz z ewaluacją i prognozowaniem dla klasyfikacji binarnej



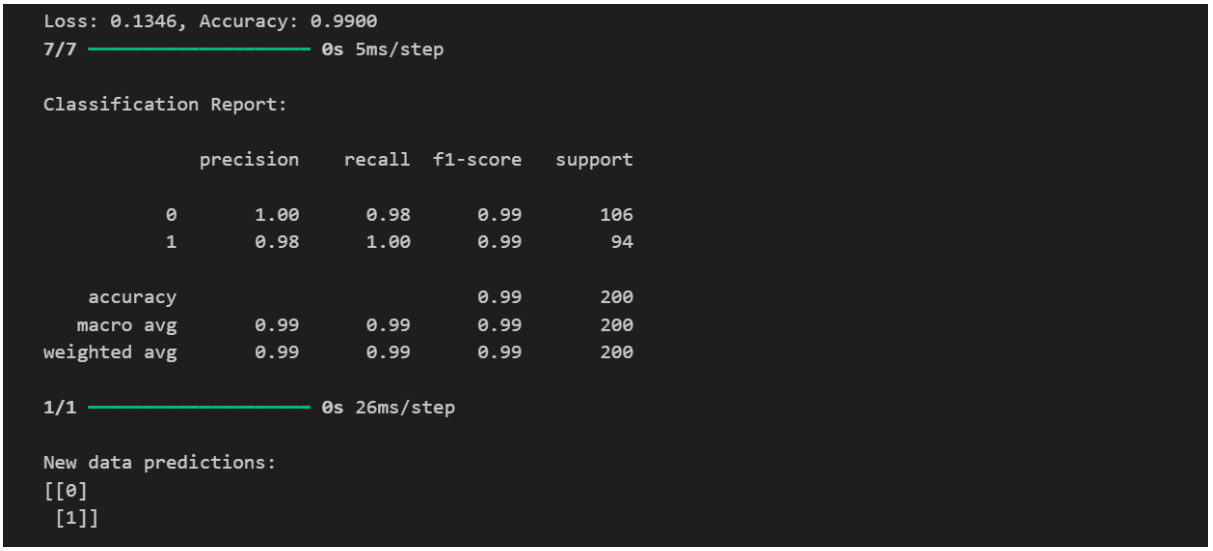
```python
# 2. Zrealizuj w Pythonie najprostszą sieć neuronową wraz z ewaluacją i prognozowaniem.
# Temat: sieć neuronowa do klasyfikacji binarnej.

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# Generowanie przykładowych danych binarnych (1000 próbek z dwoma cechami)
np.random.seed(42)
X = np.random.rand(1000, 2)  # wejście
y = (X[:, 0] + X[:, 1] > 1).astype(int)  # etykiety: klasa 1, jeśli suma cech > 1, inaczej klasa 0

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

model = Sequential([
    Dense(4, activation='relu', input_shape=(2,)),  # Warstwa ukryta z 4 neuronami
    Dense(1, activation='sigmoid')  # Warstwa wyjściowa (klasyfikacja binarna)
])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']) # Kompilacja modelu

history = model.fit(X_train, y_train, epochs=50, batch_size=8, verbose=0) # Trenownaine

loss, accuracy = model.evaluate(X_test, y_test, verbose=0) # Ewaluacja modelu na danych testowych
print(f'Loss: {loss:.4f}, Accuracy: {accuracy:.4f}')

predictions = (model.predict(X_test) > 0.5).astype(int) # Prognozowanie na podstawie danych testowych

print("\nClassification Report:\n") # Raport klasyfikacji
print(classification_report(y_test, predictions))


new_data = np.array([[0.1, 0.4], [0.8, 0.7]]) # Przykładowe prognozy dla nowych danych
predictions_new = (model.predict(new_data) > 0.5).astype(int)
print("\nNew data predictions:")
print(predictions_new)
```

Rys. 3. Realizacja sieci neuronowej w Pythonie.

Zaprezentowany na rysunku trzecim program składa się z czterech podstawowych części: generowania danych, tworzenie modelu, trenowanie oraz ewaluacja i prognozowanie.

Generowane dane tworzymy za pomocą losowych liczb i etykiet “y”, zależnych od sumy cech w każdej próbce. Wartości “y” mają wartość 0 lub 1 (klasyfikacja binarna). Sieć neuronowa zawiera warstwę ukrytą z 4 neutronami i funkcją aktywacji ReLU. Warstwa wyjściowa przechodzi przez funkcję sigmoid, która zwraca wynik w przedziale (0,1). Trenowanie odbywa się na modelu przez 50 epok w oparciu o dane treningowe.

Najbardziej przydatną funkcją jest ewaluacja i prognozowanie, gdzie obliczana jest strata i dokładność danych testowych oraz generowanie prognozy dla nowych danych, wyświetlając wyniki.



Rys. 4. Realizacja sieci neuronowej w Pythonie.

Wyniki ewaluacji modelu na zbiorze testowym wykazały, że wartość straty (loss) wyniosła 0.1346, co sugeruje, że model dobrze dopasował się do danych. Wysoka dokładność (accuracy) na poziomie 0.9900 wskazuje na to, że model poprawnie klasyfikował 99% próbek w zbiorze testowym.

Raport klasyfikacji dostarczył szczegółowych informacji na temat wydajności modelu w kontekście poszczególnych klas. W przypadku klasy 0, precyzja wyniosła 1.00, co oznacza, że wszystkie przewidywania dla tej klasy były poprawne. Wartość recall dla klasy 0 wyniosła 0.98, co oznacza, że model poprawnie zidentyfikował 98% rzeczywistych próbek tej klasy. Dla klasy 1, precyzja ma wartość 0.98, a to oznacza, że 98% przewidywań dla tej klasy było trafnych, podczas gdy recall wyniósł 1.00, czyli model zidentyfikował wszystkie rzeczywiste próbki tej klasy. Wartości f1-score dla obu klas były bliskie 1, co potwierdza wysoką jakość klasyfikacji.

W ostatnim kroku przeprowadzono prognozowanie na podstawie nowych danych. Model przewidział, że dla danych wejściowych [0.1, 0.4] przypisano klasę 0, natomiast dla danych [0.8, 0.7] przypisano klasę 1. Te wyniki wskazują na zdolność modelu do generalizacji i podejmowania decyzji na podstawie wcześniej niewidzianych danych.

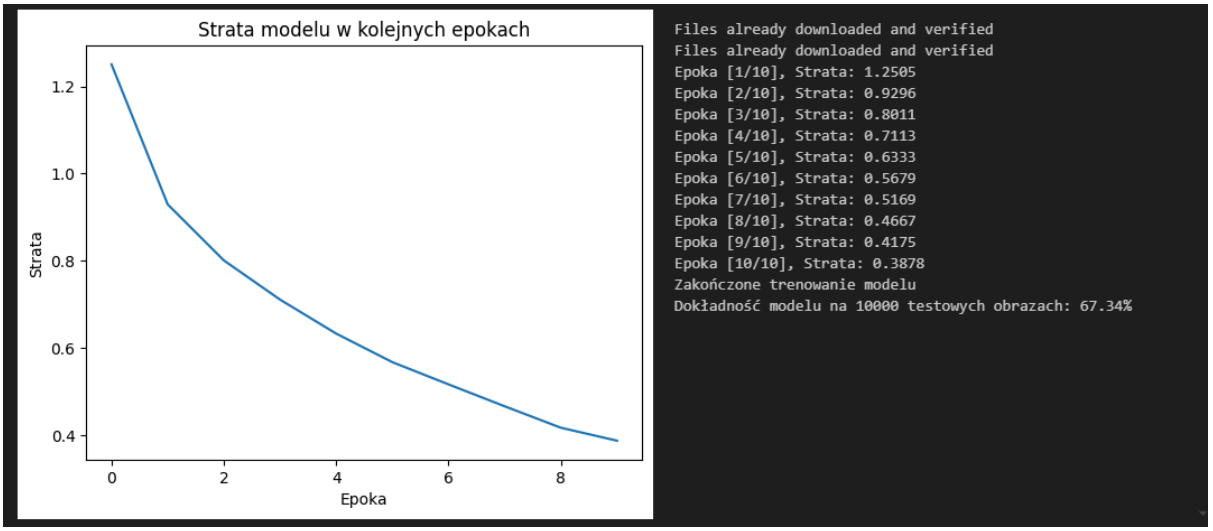
Projektowanie tworzenie i testowanie sieci konwolucyjnej

```python
# 3. Zrealizuj projektowanie, trenowanie i testowanie sieci konwolucyjnej na podstawie jednego z dostępnych w Pyt
# Wariant drugi: Zaprojektuj, wytrenuj i przetestuj sieć konwolucyjną na zbiorze CIFAR-10.

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F  # Dodany import

# Ustawienia urządzenia
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Wczytanie zbioru danych CIFAR-10
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=False)

# Definiowanie architektury sieci konwolucyjnej
class CNN(nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Inicjalizacja modelu, funkcji straty i optymalizatora
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Pętla treningowa
num_epochs = 10
train_losses = []
for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    avg_loss = running_loss / len(trainloader)
    train_losses.append(avg_loss)
    print(f'Epoka [{epoch + 1}/{num_epochs}], Strata: {avg_loss:.4f}')

print('Zakończone trenowanie modelu')

# Testowanie modelu
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Dokładność modelu na 10000 testowych obrazach: {100 * correct / total:.2f}%')

# Wizualizacja strat
plt.plot(train_losses)
plt.title('Strata modelu w kolejnych epokach')
plt.xlabel('Epoka')
plt.ylabel('Strata')
plt.show()
```

Rys. 5. Program sieci konwolucyjnej

W kodzie zrealizowano projektowanie, trenowanie i testowanie sieci konwolucyjnej na zbiorze danych CIFAR-10. Najpierw zaimportowano niezbędne biblioteki, a następnie wczytano dane, stosując odpowiednie transformacje, takie jak normalizacja.

W kolejnym kroku zdefiniowano architekturę sieci konwolucyjnej, składającą się z dwóch warstw konwolucyjnych, warstw poolingowych oraz dwóch warstw w pełni połączonych. Model został wytrenowany przez dziesięć epok, a podczas treningu obliczano stratę, która była monitorowana w każdej epoce. Po zakończeniu treningu przeprowadzono testowanie modelu na zbiorze testowym, obliczając dokładność klasyfikacji. Na koniec wizualizowano straty modelu w kolejnych epokach, co pozwoliło na ocenę procesu uczenia.



Rys. 6. Wynik wykonywania operacji.

Z wykresu przedstawionego na rysunku szóstym można wywnioskować, że wraz z każdą epoką strata była coraz niższa. Dokładność modelu na 10000 obrazach wynosiła 67.34%.

3. Wnioski

Przeprowadzony w drugim zadaniu eksperyment wykazał, że zbudowana sieć neuronowa skutecznie klasyfikuje dane binarne, osiągając wysoką dokładność oraz doskonałe wyniki w raportach klasyfikacyjnych. Model wykazał się również zdolnością do prognozowania na podstawie nowych danych, co potwierdza możliwość użycia go w praktycznych zastosowaniach.

Zbiór danych CIFAR zajmuje 170MB miejsca na dysku, warto zadbać o jego usunięcie po wykonaniu ćwiczenia w celu zwolnienia miejsca. Czas trenowania modelu z zadania trzeciego wynosił 10 minut i stanowił najdłuższą pod względem czasu wykonywania operacją. Proces ten może zostać przyśpieszony po pobraniu środowiska programistycznego języka Cuda ze strony producenta (Nvidii). Po zainstalowaniu zajmującego 3.5GB miejsca na dysku środowiska, czas skraca się w zależności od karty graficznej do kilku sekund/milisekund.