Sieci neuronowe

Wstęp

Celem laboratorium jest zapoznanie się z podstawami sieci neuronowych oraz uczeniem głębokim (*deep learning*). Zapoznasz się na nim z następującymi tematami:

- treningiem prostych sieci neuronowych, w szczególności z:
  - regresją liniową w sieciach neuronowych
  - optymalizacją funkcji kosztu
  - algorytmem spadku wzdłuż gradientu
  - siecią typu Multilayer Perceptron (MLP)
- frameworkiem PyTorch, w szczególności z:
  - ładowaniem danych
  - preprocessingiem danych
  - pisaniem pętli treningowej i walidacyjnej
  - walidacją modeli
- architekturą i hiperaprametrami sieci MLP, w szczególności z:
  - warstwami gęstymi (w pełni połączonymi)
  - funkcjami aktywacji
  - regularyzacją: L2, dropout

Wykorzystywane biblioteki

Zaczniemy od pisania ręcznie prostych sieci w bibliotece Numpy, służącej do obliczeń numerycznych na CPU. Później przejdziemy do wykorzystywania frameworka PyTorch, służącego do obliczeń numerycznych na CPU, GPU oraz automatycznego różniczkowania, wykorzystywanego głównie do treningu sieci neuronowych.

Wykorzystamy PyTorcha ze względu na popularność, łatwość instalacji i użycia, oraz dużą kontrolę nad niskopoziomowymi aspektami budowy i treningu sieci neuronowych. Framework ten został stworzony do zastosowań badawczych i naukowych, ale ze względu na wygodę użycia stał się bardzo popularny także w przemyśle. W szczególności całkowicie zdominował przetwarzanie języka naturalnego (NLP) oraz uczenie na grafach.

Pierwszy duży framework do deep learningu, oraz obecnie najpopularniejszy, to TensorFlow, wraz z wysokopoziomową nakładką Keras. Są jednak szanse, że Google (autorzy) będzie go powoli porzucać na rzecz ich nowego frameworka JAX (dyskusja, artykuł Business Insidera), który jest bardzo świeżym, ale ciekawym narzędziem.

Trzecia, ale znacznie mniej popularna od powyższych opcja to Apache MXNet.

Konfiguracja własnego komputera

Jeżeli korzystasz z własnego komputera, to musisz zainstalować trochę więcej bibliotek (Google Colab ma je już zainstalowane).

Jeżeli nie masz GPU lub nie chcesz z niego korzystać, to wystarczy znaleźć odpowiednią komendę CPU na stronie PyTorcha. Dla Anacondy odpowiednia komenda została podana poniżej, dla pip'a znajdź ją na stronie.

Jeżeli chcesz korzystać ze wsparcia GPU (na tym laboratorium nie będzie potrzebne, na kolejnych może przyspieszyć nieco obliczenia), to musi być to odpowiednio nowa karta NVidii, mająca CUDA compatibility (lista). Poza PyTorchem będzie potrzebne narzędzie NVidia CUDA w wersji 11.6 lub 11.7. Instalacja na Windowsie jest bardzo prosta (wystarczy ściągnąć plik EXE i zainstalować jak każdy inny program). Instalacja na Linuxie jest trudna i można względnie łatwo zepsuć sobie system, ale jeżeli chcesz spróbować, to ten tutorial jest bardzo dobry.

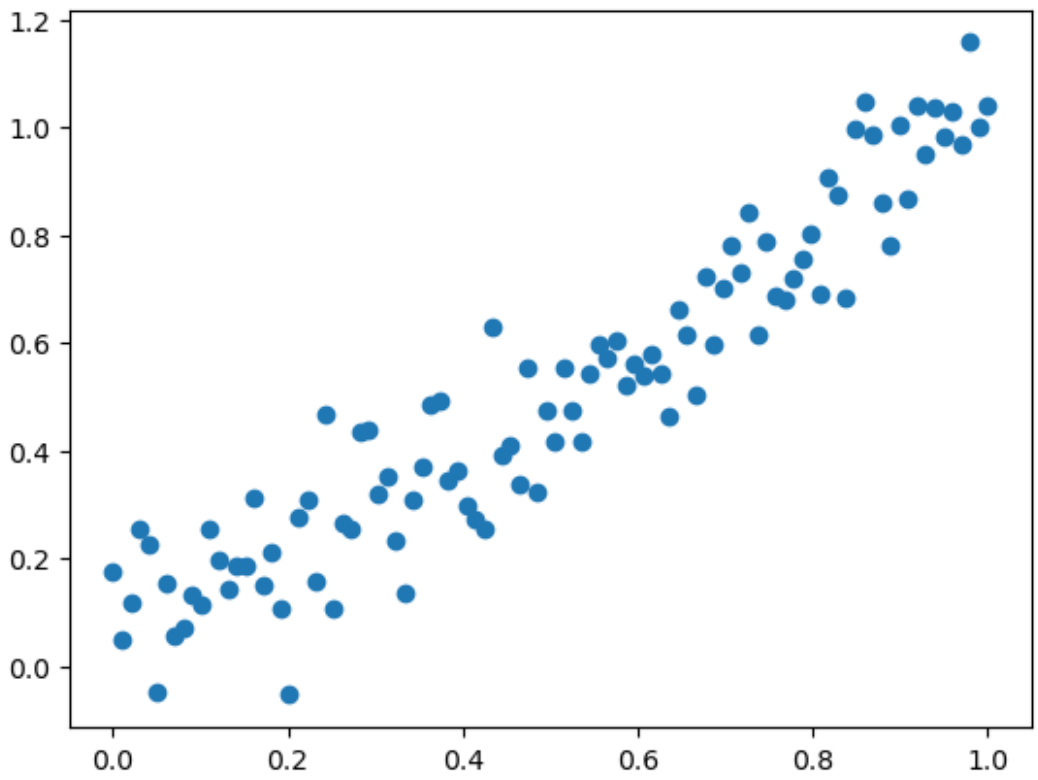
```
# for conda users
# !conda install -y matplotlib pandas pytorch torchvision torchaudio -
c pytorch -c conda-forge
```

Wprowadzenie

Zanim zaczniemy naszą przygodę z sieciami neuronowymi, przyjrzyjmy się prostemu przykładowi regresji liniowej na syntetycznych danych:

```
from typing import Tuple, Dict

import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)

x = np.linspace(0, 1, 100)
y = x + np.random.normal(scale=0.1, size=x.shape)

plt.scatter(x, y)
```

```
<matplotlib.collections.PathCollection at 0x2b8f28db490>
```



W przeciwieństwie do laboratorium 1, tym razem będziemy chcieli rozwiązać ten problem własnoręcznie, bez użycia wysokopoziomowego interfejsu Scikit-learn'a. W tym celu musimy sobie przypomnieć sformułowanie naszego **problemu optymalizacyjnego (optimization problem)**.

W przypadku prostej regresji liniowej (1 zmienna) mamy model postaci $\hat{y} = \alpha x + \beta$, z dwoma parametrami, których będziemy się uczyć. Miarą niedopasowania modelu o danych parametrach jest **funkcja kosztu (cost function)**, nazywana też funkcją celu. Najczęściej używa się **błędu średniokwadratowego (mean squared error, MSE)**:

$$\large MSE = \frac{1}{N} \sum_{i}^{N} (y - \hat{y})^2 $$

Od jakich α i β zacząć? W najprostszym wypadku wystarczy po prostu je wylosować jako niewielkie liczby zmiennoprzecinkowe.

Zadanie 1 (0.5 punkt)

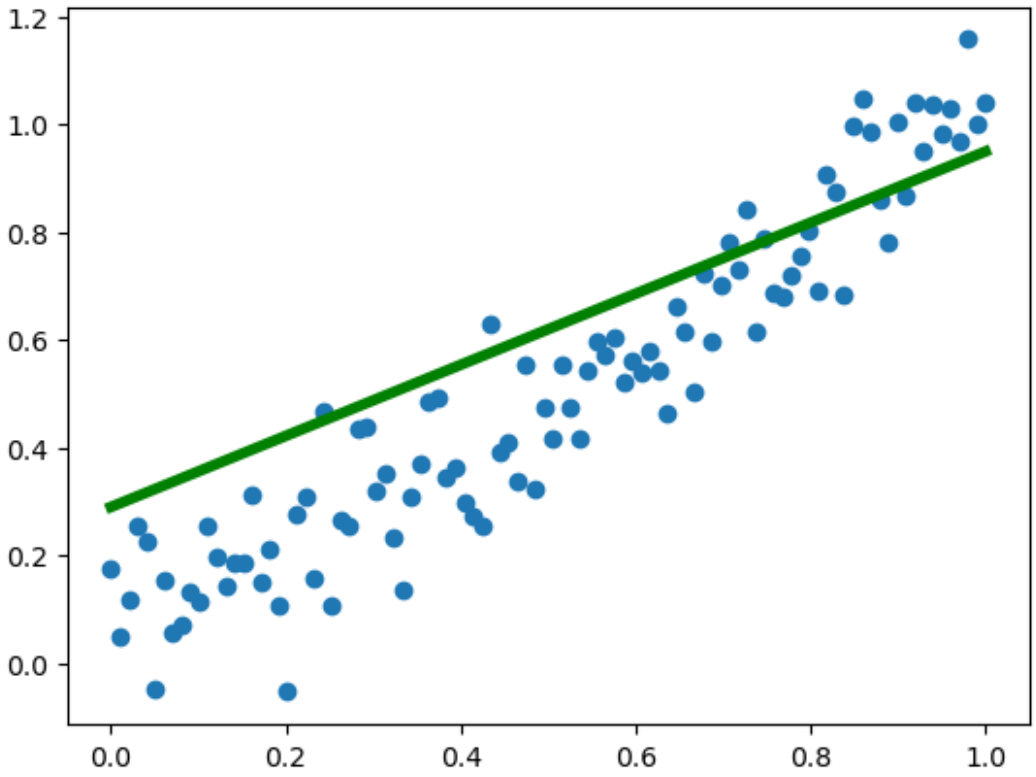
Uzupełnij kod funkcji `mse`, obliczającej błąd średniokwadratowy. Wykorzystaj Numpy'a w celu wektoryzacji obliczeń dla wydajności.

```
def mse(y: np.ndarray, y_hat: np.ndarray) -> float:
    n=len(y)
    return np.sum([np.square(y[i]-y_hat[i]) for i in range(n)])/n

a = np.random.rand()
b = np.random.rand()
```

```
print(f"MSE: {mse(y, a * x + b):.3f}")

plt.scatter(x, y)
plt.plot(x, a * x + b, color="g", linewidth=4)
```

```
MSE: 0.031
```

```
[<matplotlib.lines.Line2D at 0x2b8f2b146d0>]
```



Losowe parametry radzą sobie nie najlepiej. Jak lepiej dopasować naszą prostą do danych? Zawsze możemy starać się wyprowadzić rozwiązanie analitycznie, i w tym wypadku nawet nam się uda. Jest to jednak szczególny i dość rzadki przypadek, a w szczególności nie będzie to możliwe w większych sieciach neuronowych.

Potrzebna nam będzie **metoda optymalizacji (optimization method)**, dającą wartości parametrów minimalizujące dowolną różniczkowalną funkcję kosztu. Zdecydowanie najpopularniejszy jest tutaj **spadek wzdłuż gradientu (gradient descent)**.

Metoda ta wywodzi się z prostych obserwacji, które tutaj przedstawimy. Bardziej szczegółowe rozwinięcie dla zainteresowanych: sekcja 4.3 "Deep Learning Book", ten praktyczny kurs, analiza oryginalnej publikacji Cauchy'ego (oryginał w języku francuskim).

Pochodna jest dokładnie równa granicy funkcji. Dla małego ϵ można ją przybliżyć jako:

$$\large \frac{f(x)}{dx} \approx \frac{f(x) - f(x+\epsilon)}{\epsilon} $$

Przyglądając się temu równaniu widzimy, że:

- dla funkcji rosnącej ($f(x+\epsilon)>f(x)$) wyrażenie $\frac{f(x)}{dx}$ będzie miało znak ujemny
- dla funkcji malejącej ($f(x+\epsilon)<f(x)$) wyrażenie $\frac{f(x)}{dx}$ będzie miało znak dodatni

Widzimy więc, że potrafimy wskazać kierunek zmniejszenia wartości funkcji, patrząc na znak pochodnej. Zaobserwowano także, że amplituda wartości w $\frac{f(x)}{dx}$ jest tym większa, im dalej jesteśmy od minimum (maximum). Pochodna wyznacza więc, w jakim kierunku funkcja najszybciej rośnie, więc kierunek o przeciwnym zwrocie to kierunek, w którym funkcja najszybciej spada.

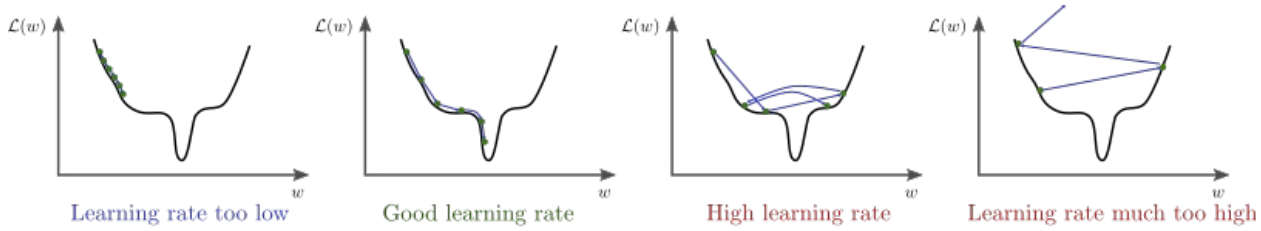
Stosując powyższe do optymalizacji, mamy:

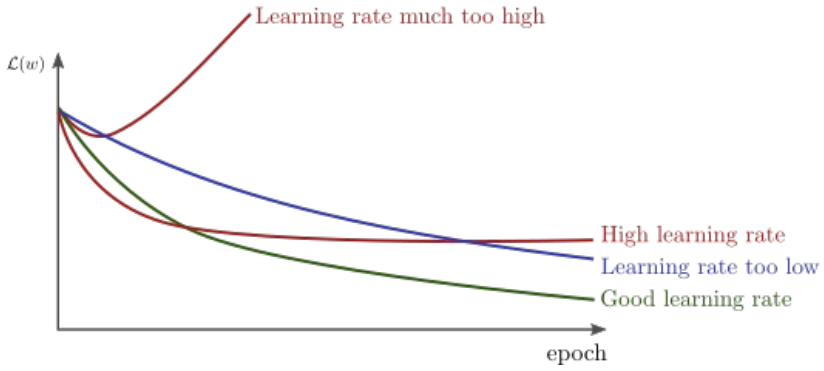
$$\large x_{t+1} = x_{t} - \alpha * \frac{f(x)}{dx} $$

α to niewielka wartość (rzędu zwykle 10^{-5} - 10^{-2}), wprowadzona, aby trzymać się założenia o małej zmianie parametrów (ϵ). Nazywa się ją **stałą uczącą (learning rate)** i jest zwykle najważniejszym hiperparametrem podczas nauki sieci.

Metoda ta zakłada, że używamy całego zbioru danych do aktualizacji parametrów w każdym kroku, co nazywa się po prostu GD (od *gradient descent*) albo *full batch GD*. Wtedy każdy krok optymalizacji nazywa się **epoką (epoch)**.

Im większa stała ucząca, tym większe nasze kroki podczas minimalizacji. Możemy więc uczyć szybciej, ale istnieje ryzyko, że będziemy "przeskakiwać" minima. Mniejsza stała ucząca to wolniejszy trening, ale dokładniejszy. Można także zmieniać ją podczas treningu, co nazywa się **learning rate scheduling (LR scheduling)**. Obrazowo:





Policzmy więc pochodną dla naszej funkcji kosztu MSE. Pochodną liczymy po predykcjach naszego modelu, czyli de facto po jego parametrach, bo to od nich zależą predykcje.

$$\large \frac{\text{d} MSE}{\text{d} \hat{y}} = -2 \cdot \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i) = -2 \cdot \frac{1}{N} \sum_{i=1}^{N} (y_i - (ax + b)) $$

Musimy jeszcze się dowiedzieć, jak zaktualizować każdy z naszych parametrów. Możemy wykorzystać tutaj regułę łańcuchową (*chain rule*) i policzyć ponownie pochodną, tylko że po naszych parametrach. Dzięki temu dostajemy informację, jak każdy z parametrów wpływa na funkcję kosztu i jak zmodyfikować każdy z nich w kolejnym kroku.

$$\large \frac{\text{d} \hat{y}}{\text{d} a} = x $$

$$\large \frac{\text{d} \hat{y}}{\text{d} b} = 1 $$

Pełna aktualizacja to zatem:

$$\large a' = a + \alpha * \left(\frac{-2}{N} \sum_{i=1}^N (y_i - \hat{y}_i) * (-x) \right) $$

$$\large b' = b + \alpha * \left(\frac{-2}{N} \sum_{i=1}^N (y_i - \hat{y}_i) * (-1) \right) $$

Liczymy więc pochodną funkcji kosztu, a potem za pomocą reguły łańcuchowej "cofamy się", dochodząc do tego, jak każdy z parametrów wpływa na błąd i w jaki sposób powinniśmy go zmienić. Nazywa się to **propagacją wsteczną (backpropagation)** i jest podstawowym mechanizmem umożliwiającym naukę sieci neuronowych za pomocą spadku wzdłuż gradientu. Więcej możesz o tym przeczytać tutaj.

Obliczenie pochodnych cząstkowych ze względu na każdy

Zadanie 2 (1.5 punkty)

Zaimplementuj funkcję realizującą jedną epokę treningową. Oblicz predykcję przy aktualnych parametrach oraz zaktualizuj je zgodnie z powyższymi wzorami.

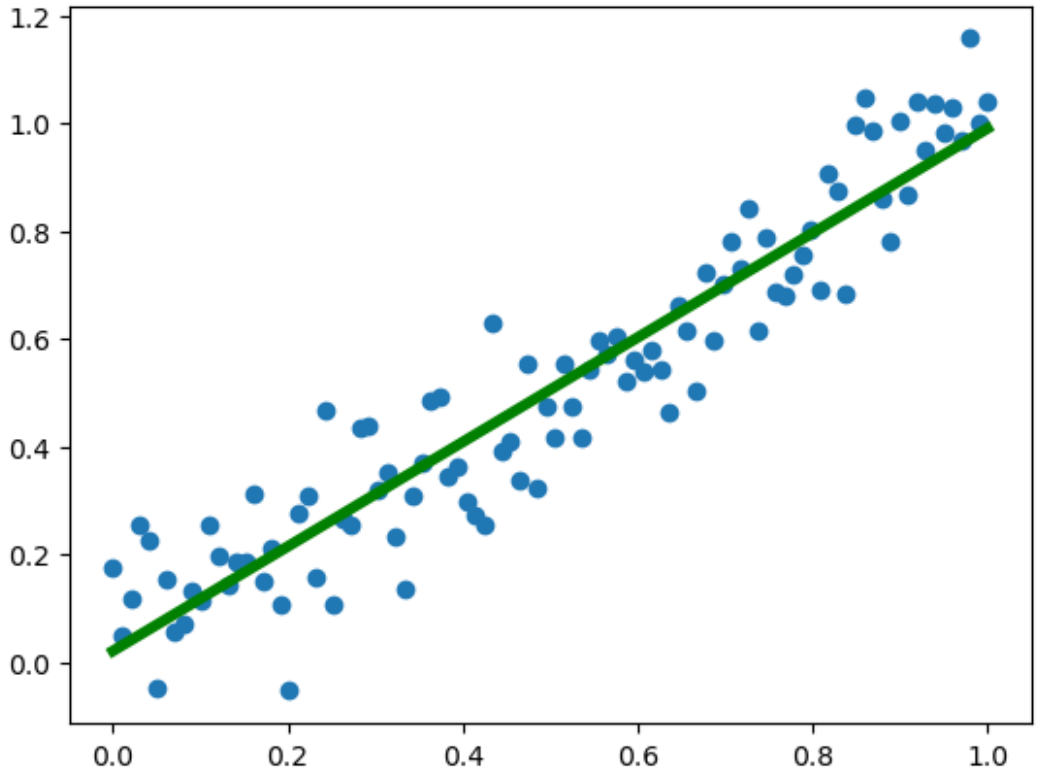
```
def optimize(
    x: np.ndarray, y: np.ndarray, a: float, b: float, learning_rate:
```

```
float = 0.1
):
    n=len(y)
    y_hat = a * x + b
    errors = y - y_hat

    d = lambda arg: (-2 * np.sum(errors * arg))/n

    new_a =  a + learning_rate  * d(-x)
    new_b = b + learning_rate * d(-1)

    return new_a, new_b

for i in range(1000):
    loss = mse(y, a * x + b)
    a, b = optimize(x, y, a, b)
    if i % 100 == 0:
        print(f"step {i} loss: ", loss)

print("final loss:", loss)
```

```
step 0 loss:  0.03129605876737945
step 100 loss:  0.010805769233407923
step 200 loss:  0.010131668825057637
step 300 loss:  0.010086371617631045
step 400 loss:  0.010083327802532835
step 500 loss:  0.010083123268724275
step 600 loss:  0.010083109524762341
step 700 loss:  0.010083108601215808
step 800 loss:  0.010083108539156688
step 900 loss:  0.01008310853498653
final loss: 0.010083108534706862
```

```
plt.scatter(x, y)
plt.plot(x, a * x + b, color="g", linewidth=4)
```

```
[<matplotlib.lines.Line2D at 0x2b8f2b5b940>]
```



Udało ci się wytrenować swoją pierwszą sieć neuronową. Czemu? Otóż neuron to po prostu wektor parametrów, a zwykle robimy iloczyn skalarny tych parametrów z wejściem. Dodatkowo na wyjście nakłada się **funkcję aktywacji (activation function)**, która przekształca wyjście. Tutaj takiej nie było, a właściwie była to po prostu funkcja identyczności.

Oczywiście w praktyce korzystamy z odpowiedniego frameworka, który w szczególności:

- ułatwia budowanie sieci, np. ma gotowe klasy dla warstw neuronów
- ma zaimplementowane funkcje kosztu oraz ich pochodne
- sam różniczkuje ze względu na odpowiednie parametry i aktualizuje je odpowiednio podczas treningu

Wprowadzenie do PyTorcha

PyTorch to w gruncie rzeczy narzędzie do algebry liniowej z automatycznym rożniczkowaniem, z możliwością przyspieszenia obliczeń z pomocą GPU. Na tych fundamentach zbudowany jest pełny framework do uczenia głębokiego. Można spotkać się ze stwierdzenie, że PyTorch to NumPy + GPU + opcjonalne różniczkowanie, co jest całkiem celne. Plus można łatwo debugować printem :)

PyTorch używa dynamicznego grafu obliczeń, który sami definiujemy w kodzie. Takie podejście jest bardzo wygodne, elastyczne i pozwala na łatwe eksperymentowanie. Odbywa się to potencjalnie kosztem wydajności, ponieważ pozostawia kwestię optymalizacji programiście. Więcej na ten temat dla zainteresowanych na końcu laboratorium.

Samo API PyTorcha bardzo przypomina Numpy'a, a podstawowym obiektem jest `Tensor`, klasa reprezentująca tensory dowolnego wymiaru. Dodatkowo niektóre tensory będą miały automatycznie obliczony gradient. Co ważne, tensor jest na pewnym urządzeniu, CPU lub GPU, a przenosić między nimi trzeba explicite.

Najważniejsze moduły:

- `torch` - podstawowe klasy oraz funkcje, np. `Tensor`, `from_numpy()`
- `torch.nn` - klasy związane z sieciami neuronowymi, np. `Linear`, `Sigmoid`
- `torch.optim` - wszystko związane z optymalizacją, głównie spadkiem wzdłuż gradientu

```
import torch
import torch.nn as nn
import torch.optim as optim

ones = torch.ones(10)
noise = torch.ones(10) * torch.rand(10)

# elementwise sum
print(ones + noise)

# elementwise multiplication
print(ones * noise)

# dot product
print(ones @ noise)
```

```
tensor([1.0716, 1.7756, 1.6651, 1.0794, 1.9280, 1.2550, 1.3943,
1.0271, 1.7158,
        1.1696])
tensor([0.0716, 0.7756, 0.6651, 0.0794, 0.9280, 0.2550, 0.3943,
0.0271, 0.7158,
        0.1696])
tensor(4.0816)
```

```
# beware - shares memory with original Numpy array!
# very fast, but modifications are visible to original variable
x = torch.from_numpy(x)
y = torch.from_numpy(y)
```

Jeżeli dla stworzonych przez nas tensorów chcemy śledzić operacje i obliczać gradient, to musimy oznaczyć `requires_grad=True`.

```
a = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)
a, b
```

```
(tensor([0.3545], requires_grad=True), tensor([0.1534],
requires_grad=True))
```

PyTorch zawiera większość powszechnie używanych funkcji kosztu, np. MSE. Mogą być one używane na 2 sposoby, z czego pierwszy jest popularniejszy:

- jako klasy wywoływalne z modułu `torch.nn`
- jako funkcje z modułu `torch.nn.functional`

Po wykonaniu poniższego kodu widzimy, że zwraca on nam tensor z dodatkowymi atrybutami. Co ważne, jest to skalar (0-wymiarowy tensor), bo potrzebujemy zwyczajnej liczby do obliczania propagacji wstecznych (pochodnych cząstkowych).

```
mse = nn.MSELoss()
mse(y, a * x + b)
```

```
tensor(0.0731, dtype=torch.float64, grad_fn=<MseLossBackward0>)
```

Atrybutu `grad_fn` nie używamy wprost, bo korzysta z niego w środku PyTorch, ale widać, że tensor jest "świadomy", że liczy się na nim pochodną. Możemy natomiast skorzystać z atrybutu `grad`, który zawiera faktyczny gradient. Zanim go jednak dostaniemy, to trzeba powiedzieć PyTorchowi, żeby policzył gradient. Służy do tego metoda `.backward()`, wywoływana na obiekcie zwracanym przez funkcję kosztu.

```
loss = mse(y, a * x + b)
loss.backward()

print(a.grad)
```

```
tensor([-0.2800])
```

Ważne jest, że PyTorch nie liczy za każdym razem nowego gradientu, tylko dodaje go do istniejącego, czyli go akumuluje. Jest to przydatne w niektórych sieciach neuronowych, ale zazwyczaj trzeba go zerować. Jeżeli tego nie zrobimy, to dostaniemy coraz większe gradienty.

Do zerowania służy metoda `.zero_()`. W PyTorchu wszystkie metody modyfikujące tensor w miejscu mają `_` na końcu nazwy. Jest to dość niskopoziomowa operacja dla pojedynczych tensorów - zobaczymy za chwilę, jak to robić łatwiej dla całej sieci.

```
loss = mse(y, a * x + b)
loss.backward()
a.grad
```

```
tensor([-0.5601])
```

Zobaczmy, jak wyglądałaby regresja liniowa, ale napisana w PyTorchu. Jest to oczywiście bardzo niskopoziomowa implementacja - za chwilę zobaczymy, jak to wygląda w praktyce.

```
learning_rate = 0.1
for i in range(1000):
    loss = mse(y, a * x + b)

    # compute gradients
```

```
    loss.backward()

    # update parameters
    a.data -= learning_rate * a.grad
    b.data -= learning_rate * b.grad

    # zero gradients
    a.grad.data.zero_()
    b.grad.data.zero_()

    if i % 100 == 0:
        print(f"step {i} loss: ", loss)

print("final loss:", loss)
```

```
step 0 loss:  tensor(0.0731, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 100 loss:  tensor(0.0116, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 200 loss:  tensor(0.0102, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 300 loss:  tensor(0.0101, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 400 loss:  tensor(0.0101, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 500 loss:  tensor(0.0101, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 600 loss:  tensor(0.0101, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 700 loss:  tensor(0.0101, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 800 loss:  tensor(0.0101, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 900 loss:  tensor(0.0101, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
final loss: tensor(0.0101, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
```

Trening modeli w PyTorchu jest dosyć schematyczny i najczęściej rozdziela się go na kilka bloków, dających razem **pętlę uczącą (training loop)**, powtarzaną w każdej epoce:

1. Forward pass - obliczenie predykcji sieci
2. Loss calculation
3. Backpropagation - obliczenie pochodnych oraz zerowanie gradientów
4. Optimalization - aktualizacja wag
5. Other - ewaluacja na zbiorze walidacyjnym, logging etc.

```
# initialization
learning_rate = 0.1
a = torch.rand(1, requires_grad=True)
```

```
b = torch.rand(1, requires_grad=True)
optimizer = torch.optim.SGD([a, b], lr=learning_rate)
best_loss = float("inf")

# training loop in each epoch
for i in range(1000):
    # forward pass
    y_hat = a * x + b

    # loss calculation
    loss = mse(y, y_hat)

    # backpropagation
    loss.backward()

    # optimization
    optimizer.step()
    optimizer.zero_grad()  # zeroes all gradients - very convenient!

    if i % 100 == 0:
        if loss < best_loss:
            best_model = (a.clone(), b.clone())
            best_loss = loss
        print(f"step {i} loss: {loss.item():.4f}")

print("final loss:", loss)
```

```
step 0 loss: 0.1219
step 100 loss: 0.0115
step 200 loss: 0.0102
step 300 loss: 0.0101
step 400 loss: 0.0101
step 500 loss: 0.0101
step 600 loss: 0.0101
step 700 loss: 0.0101
step 800 loss: 0.0101
step 900 loss: 0.0101
final loss: tensor(0.0101, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
```

Przejdziemy teraz do budowy sieci neuronowej do klasyfikacji. Typowo implementuje się ją po prostu jako sieć dla regresji, ale zwracającą tyle wyników, ile mamy klas, a potem aplikuje się na tym funkcję sigmoidalną (2 klasy) lub softmax (>2 klasy). W przypadku klasyfikacji binarnej zwraca się czasem tylko 1 wartość, przepuszczaną przez sigmoidę - wtedy wyjście z sieci to prawdopodobieństwo klasy pozytywnej.

Funkcją kosztu zwykle jest **entropia krzyżowa (cross-entropy)**, stosowana też w klasycznej regresji logistycznej. Co ważne, sieci neuronowe, nawet tak proste, uczą się szybciej i stabilniej, gdy dane na wejściu (a przynajmniej zmienne numeryczne) są **ustandaryzowane (standardized)**.

Operacja ta polega na odjęciu średniej i podzieleniu przez odchylenie standardowe (tzw. *Z-score transformation*).

Uwaga - PyTorch wymaga tensora klas będącego liczbami zmiennoprzecinkowymi!

Zbiór danych

Na tym laboratorium wykorzystamy zbiór Adult Census. Dotyczy on przewidywania na podstawie danych demograficznych, czy dany człowiek zarabia powyżej 50 tysięcy dolarów miesięcznie, czy też mniej. Jest to cenna informacja np. przy planowaniu kampanii marketingowych. Jak możesz się domyślić, zbiór pochodzi z czasów, kiedy inflacja była dużo niższa :)

Poniżej znajduje się kod do ściągnięcia i preprocessingu zbioru. Nie musisz go dokładnie analizować.

```
!wget https://archive.ics.uci.edu/ml/machine-learning-
databases/adult/adult.data
```

```
'wget' is not recognized as an internal or external command,
operable program or batch file.
```

```
import pandas as pd

columns = [
    "age",
    "workclass",
    "fnlwgt",
    "education",
    "education-num",
    "marital-status",
    "occupation",
    "relationship",
    "race",
    "sex",
    "capital-gain",
    "capital-loss",
    "hours-per-week",
    "native-country",
    "wage"
]

"""
age: continuous.
workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov,
Local-gov, State-gov, Without-pay, Never-worked.
fnlwgt: continuous.
education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-
acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th,
Doctorate, 5th-6th, Preschool.
```

```
education-num: continuous.
marital-status: Married-civ-spouse, Divorced, Never-married,
Separated, Widowed, Married-spouse-absent, Married-AF-spouse.
occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-
managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-
clerical, Farming-fishing, Transport-moving, Priv-house-serv,
Protective-serv, Armed-Forces.
relationship: Wife, Own-child, Husband, Not-in-family, Other-relative,
Unmarried.
race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
sex: Female, Male.
capital-gain: continuous.
capital-loss: continuous.
hours-per-week: continuous.
native-country: United-States, Cambodia, England, Puerto-Rico, Canada,
Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South,
China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica,
Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos,
Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua,
Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru,
Hong, Holand-Netherlands.
"""

df = pd.read_csv("adult.data", header=None, names=columns)
df.wage.unique()
```

```
array([' <=50K', ' >50K'], dtype=object)
```

```
# attribution: https://www.kaggle.com/code/royshih23/topic7-
classification-in-python
df['education'].replace('Preschool', 'dropout',inplace=True)
df['education'].replace('10th', 'dropout',inplace=True)
df['education'].replace('11th', 'dropout',inplace=True)
df['education'].replace('12th', 'dropout',inplace=True)
df['education'].replace('1st-4th', 'dropout',inplace=True)
df['education'].replace('5th-6th', 'dropout',inplace=True)
df['education'].replace('7th-8th', 'dropout',inplace=True)
df['education'].replace('9th', 'dropout',inplace=True)
df['education'].replace('HS-Grad', 'HighGrad',inplace=True)
df['education'].replace('HS-grad', 'HighGrad',inplace=True)
df['education'].replace('Some-college',
'CommunityCollege',inplace=True)
df['education'].replace('Assoc-acdm', 'CommunityCollege',inplace=True)
df['education'].replace('Assoc-voc', 'CommunityCollege',inplace=True)
df['education'].replace('Bachelors', 'Bachelors',inplace=True)
df['education'].replace('Masters', 'Masters',inplace=True)
df['education'].replace('Prof-school', 'Masters',inplace=True)
df['education'].replace('Doctorate', 'Doctorate',inplace=True)

df['marital-status'].replace('Never-married',
```

```
'NotMarried',inplace=True)
df['marital-status'].replace(['Married-AF-spouse'],
'Married',inplace=True)
df['marital-status'].replace(['Married-civ-spouse'],
'Married',inplace=True)
df['marital-status'].replace(['Married-spouse-absent'],
'NotMarried',inplace=True)
df['marital-status'].replace(['Separated'], 'Separated',inplace=True)
df['marital-status'].replace(['Divorced'], 'Separated',inplace=True)
df['marital-status'].replace(['Widowed'], 'Widowed',inplace=True)

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder,
StandardScaler

X = df.copy()
y = (X.pop("wage") == ' >50K').astype(int).values

train_valid_size = 0.2

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=train_valid_size,
    random_state=0,
    shuffle=True,
    stratify=y
)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train, y_train,
    test_size=train_valid_size,
    random_state=0,
    shuffle=True,
    stratify=y_train
)

continuous_cols = ['age', 'fnlwgt', 'education-num', 'capital-gain',
'capital-loss', 'hours-per-week']
continuous_X_train = X_train[continuous_cols]
categorical_X_train = X_train.loc[:,
~X_train.columns.isin(continuous_cols)]

continuous_X_valid = X_valid[continuous_cols]
categorical_X_valid = X_valid.loc[:,
~X_valid.columns.isin(continuous_cols)]

continuous_X_test = X_test[continuous_cols]
categorical_X_test = X_test.loc[:,
~X_test.columns.isin(continuous_cols)]

categorical_encoder = OneHotEncoder(sparse=False,
```

```
handle_unknown='ignore')
continuous_scaler = StandardScaler() #MinMaxScaler(feature_range=(-1,
1))

categorical_encoder.fit(categorical_X_train)
continuous_scaler.fit(continuous_X_train)

continuous_X_train = continuous_scaler.transform(continuous_X_train)
continuous_X_valid = continuous_scaler.transform(continuous_X_valid)
continuous_X_test = continuous_scaler.transform(continuous_X_test)

categorical_X_train =
categorical_encoder.transform(categorical_X_train)
categorical_X_valid =
categorical_encoder.transform(categorical_X_valid)
categorical_X_test = categorical_encoder.transform(categorical_X_test)

X_train = np.concatenate([continuous_X_train, categorical_X_train],
axis=1)
X_valid = np.concatenate([continuous_X_valid, categorical_X_valid],
axis=1)
X_test = np.concatenate([continuous_X_test, categorical_X_test],
axis=1)

X_train.shape, y_train.shape
```

```
C:\Users\Szczepan\anaconda3\envs\PSI\lib\site-packages\sklearn\
preprocessing\_encoders.py:972: FutureWarning: `sparse` was renamed to
`sparse_output` in version 1.2 and will be removed in 1.4.
`sparse_output` is ignored unless you leave `sparse` to its default
value.
  warnings.warn(

((20838, 108), (20838,))
```

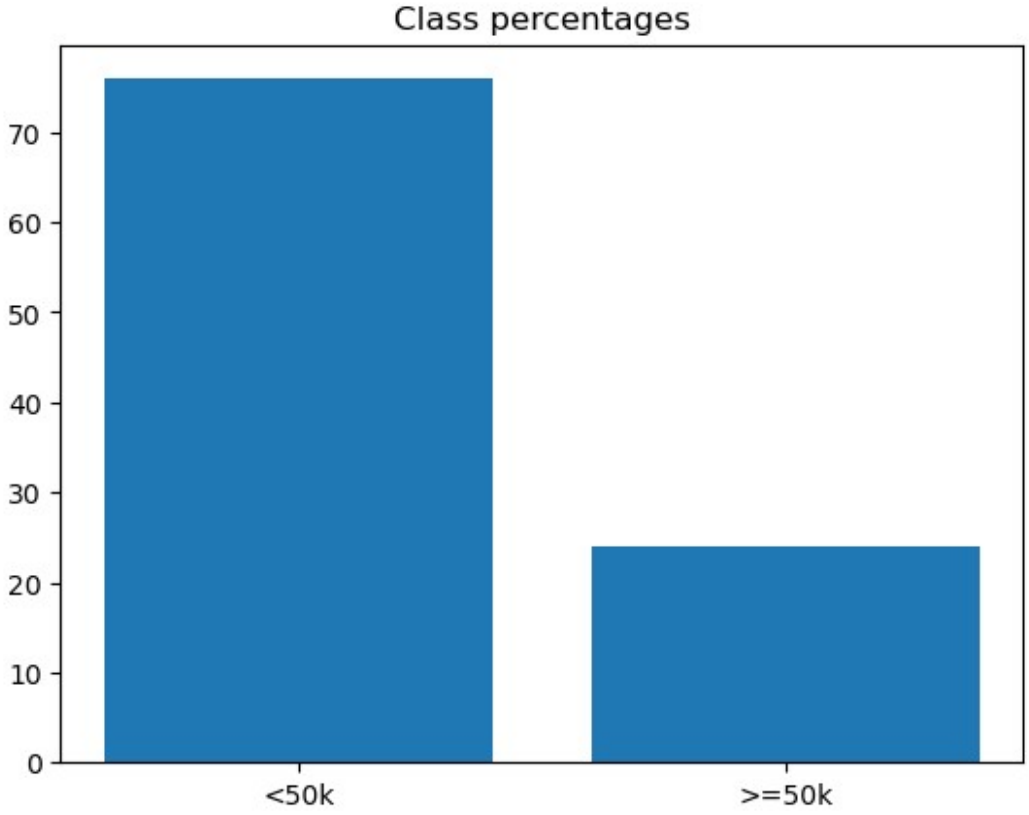
Uwaga co do typów - PyTorchu wszystko w sieci neuronowej musi być typu `float32`. W szczególności trzeba uważać na konwersje z Numpy'a, który używa domyślnie typu `float64`. Może ci się przydać metoda `.float()`.

Uwaga co do kształtów wyjścia - wejścia do `nn.BCELoss` muszą być tego samego kształtu. Może ci się przydać metoda `.squeeze()` lub `.unsqueeze()`.

```
X_train = torch.from_numpy(X_train).float()
y_train = torch.from_numpy(y_train).float().unsqueeze(-1)

X_valid = torch.from_numpy(X_valid).float()
y_valid = torch.from_numpy(y_valid).float().unsqueeze(-1)

X_test = torch.from_numpy(X_test).float()
y_test = torch.from_numpy(y_test).float().unsqueeze(-1)
```

Podobnie jak w laboratorium 2, mamy tu do czynienia z klasyfikacją niezbalansowaną:

```
import matplotlib.pyplot as plt

y_pos_perc = 100 * y_train.sum().item() / len(y_train)
y_neg_perc = 100 - y_pos_perc

plt.title("Class percentages")
plt.bar(["<50k", ">=50k"], [y_neg_perc, y_pos_perc])
plt.show()
```



W związku z powyższym będziemy używać odpowiednich metryk, czyli AUROC, precyzji i czułości.

Zadanie 3 (1 punkt)

Zaimplementuj regresję logistyczną dla tego zbioru danych, używając PyTorcha. Dane wejściowe zostały dla ciebie przygotowane w komórkach poniżej.

Sama sieć składa się z 2 elementów:

- warstwa liniowa `nn.Linear`, przekształcająca wektor wejściowy na 1 wyjście - logit
- aktywacja sigmoidalna `nn.Sigmoid`, przekształcająca logit na prawdopodobieństwo klasy pozytywnej

Użyj binarnej entropii krzyżowej `nn.BCELoss` jako funkcji kosztu. Użyj optymalizatora SGD ze stałą uczącą `1e-3`. Trenuj przez 3000 epok. Pamiętaj, aby przekazać do optymalizatora `torch.optim.SGD` parametry sieci (metoda `.parameters()`).

```
learning_rate = 1e-3

model = nn.Linear(X_train.shape[1], 1)
activation = nn.Sigmoid()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = nn.BCELoss()

for i in range(3000):
    y_pred = activation(model(X_train))
    loss = loss_fn(y_pred, y_train)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    if i % 1000 == 0:
        print(f"step {i} loss: {loss.item():.4f}")



print(f"final loss: {loss.item():.4f}")
```

```
step 0 loss: 0.7501
step 1000 loss: 0.5367
step 2000 loss: 0.4736
final loss: 0.4414
```

Teraz trzeba sprawdzić, jak poszło naszej sieci. W PyTorchu sieć pracuje zawsze w jednym z dwóch trybów: treningowym lub ewaluacyjnym (predykcyjnym). Ten drugi wyłącza niektóre mechanizmy, które są używane tylko podczas treningu, w szczególności regularyzację dropout. Do przełączania służą metody modelu `.train()` i `.eval()`.

Dodatkowo podczas liczenia predykcji dobrze jest wyłączyć liczenie gradientów, bo nie będą potrzebne, a oszczędza to czas i pamięć. Używa się do tego menadżera kontekstu `with torch.no_grad():`.

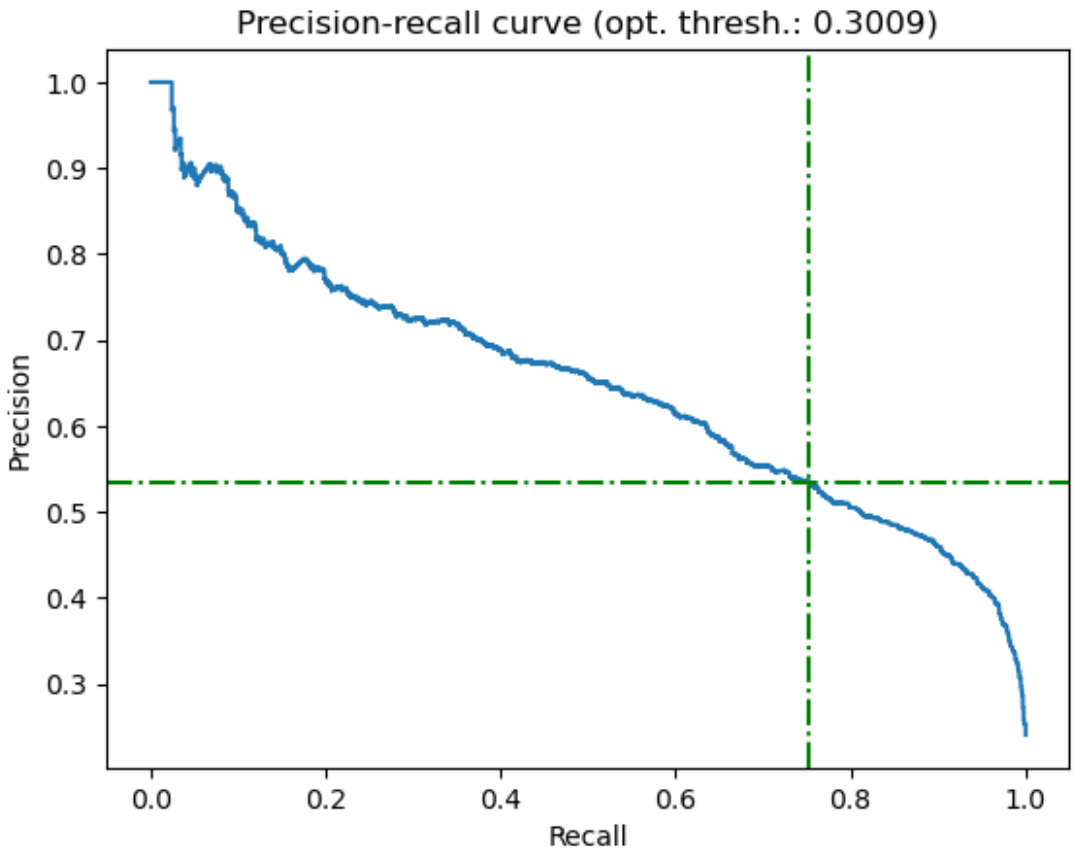
```
from sklearn.metrics import precision_recall_curve,
precision_recall_fscore_support, roc_auc_score

model.eval()
with torch.no_grad():
    y_score = activation(model(X_test))

auroc = roc_auc_score(y_test, y_score)
print(f"AUROC: {100 * auroc:.2f}%")
```

```
AUROC: 85.92%
```

Jest to całkiem dobry wynik, a może być jeszcze lepszy. Sprawdźmy dla pewności jeszcze inne metryki: precyzję, recall oraz F1-score. Dodatkowo narysujemy krzywą precision-recall, czyli jak zmieniają się te metryki w zależności od przyjętego progu (threshold) prawdopodobieństwa, powyżej którego przyjmujemy klasę pozytywną. Taką krzywą należy rysować na zbiorze walidacyjnym, bo później chcemy wykorzystać tę informację do doboru progu, a nie chcemy mieć wycieku danych testowych (data leakage).

Poniżej zaimplementowano także funkcję `get_optimal_threshold()`, która sprawdza, dla którego progu uzyskujemy maksymalny F1-score, i zwraca indeks oraz wartość optymalnego progu. Przyda ci się ona w dalszej części laboratorium.

```
from sklearn.metrics import PrecisionRecallDisplay


def get_optimal_threshold(
    precisions: np.array,
    recalls: np.array,
    thresholds: np.array
) -> Tuple[int, float]:
    f1_scores = 2 * precisions * recalls / (precisions + recalls)

    optimal_idx = np.nanargmax(f1_scores)
    optimal_threshold = thresholds[optimal_idx]

    return optimal_idx, optimal_threshold


def plot_precision_recall_curve(y_true, y_pred_score) -> None:
    precisions, recalls, thresholds = precision_recall_curve(y_true,
y_pred_score)
    optimal_idx, optimal_threshold = get_optimal_threshold(precisions,
recalls, thresholds)

    disp = PrecisionRecallDisplay(precisions, recalls)
    disp.plot()
    plt.title(f"Precision-recall curve (opt. thresh.:
{optimal_threshold:.4f})")
    plt.axvline(recalls[optimal_idx], color="green", linestyle="-.")
    plt.axhline(precisions[optimal_idx], color="green",
linestyle="-.")
    plt.show()

model.eval()
with torch.no_grad():
    y_pred_valid_score = activation(model(X_valid))

plot_precision_recall_curve(y_valid, y_pred_valid_score)
```



Jak widać, chociaż AUROC jest wysokie, to dla optymalnego F1-score recall nie jest zbyt wysoki, a precyzja jest już dość niska. Być może wynik uda się poprawić, używając modelu o większej pojemności - pełnej, głębokiej sieci neuronowej.

Sieci neuronowe

Wszystko zaczęło się od inspirowanych biologią sztucznych neuronów, których próbowano użyć do symulacji mózgu. Naukowcy szybko odeszli od tego podejścia (sam problem modelowania okazał się też znacznie trudniejszy, niż sądzono), zamiast tego używając neuronów jako jednostek reprezentującą dowolną funkcję parametryczną $f(x, \Theta)$. Każdy neuron jest zatem bardzo elastyczny, bo jedyne wymagania to funkcja różniczkowalna, a mamy do tego wektor parametrów Θ.

W praktyce najczęściej można spotkać się z kilkoma rodzinami sieci neuronowych:

1. Perceptrony wielowarstwowe (*MultiLayer Perceptron*, MLP) - najbardziej podobne do powyższego opisu, niezbędne do klasyfikacji i regresji
2. Konwolucyjne (*Convolutional Neural Networks*, CNNs) - do przetwarzania danych z zależnościami przestrzennymi, np. obrazów czy dźwięku
3. Rekurencyjne (*Recurrent Neural Networks*, RNNs) - do przetwarzania danych z zależnościami sekwencyjnymi, np. szeregi czasowe, oraz kiedyś do języka naturalnego

4. Transformacyjne (*Transformers*), oparte o mechanizm atencji (*attention*) - do przetwarzania języka naturalnego (NLP), z którego wyparły RNNs, a coraz częściej także do wszelkich innych danych, np. obrazów, dźwięku
5. Grafowe (*Graph Neural Networks*, GNNS) - do przetwarzania grafów

Na tym laboratorium skupimy się na najprostszej architekturze, czyli MLP. Jest ona powszechnie łączona z wszelkimi innymi architekturami, bo pozwala dokonywać klasyfikacji i regresji. Przykładowo, klasyfikacja obrazów to zwykle CNN + MLP, klasyfikacja tekstów to transformer + MLP, a regresja na grafach to GNN + MLP.

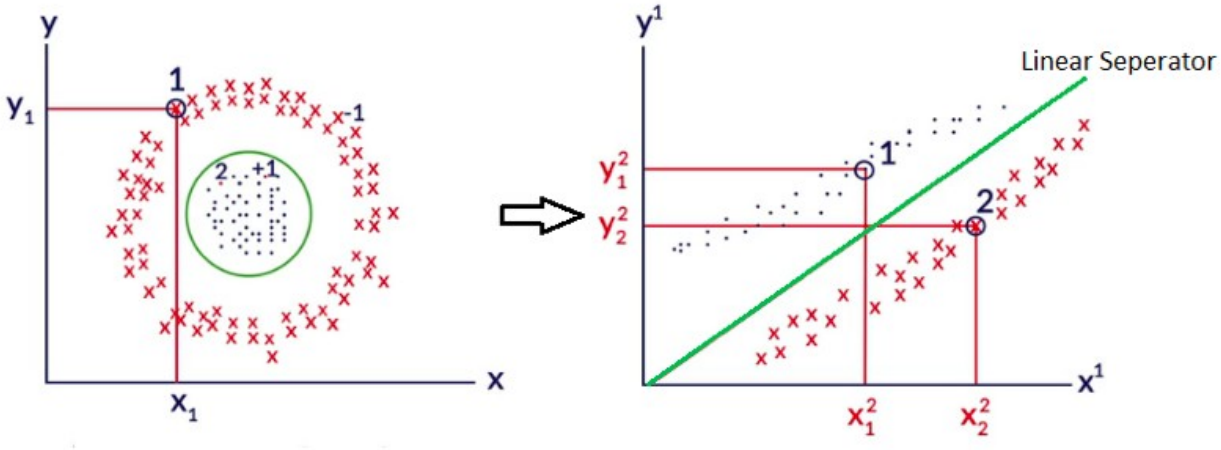
Dodatkowo, pomimo prostoty MLP są bardzo potężne - udowodniono, że perceptrony (ich powszechna nazwa) są uniwersalnym aproksymatorem, będącym w stanie przybliżyć dowolną funkcję z odpowiednio małym błędem, zakładając wystarczającą wielkość warstw sieci. Szczególne ich wersje potrafią nawet reprezentować drzewa decyzyjne.

Dla zainteresowanych polecamy doskonałą książkę "Dive into Deep Learning", z implementacjami w PyTorchu, klasyczną książkę "Deep Learning Book", oraz ten filmik, jeśli zastanawiałeś/-aś się, czemu używamy deep learning, a nie naprzykład (wide?) learning. (aka. czemu staramy się budować głębokie sieci, a nie płytkie za to szerokie)

Sieci MLP

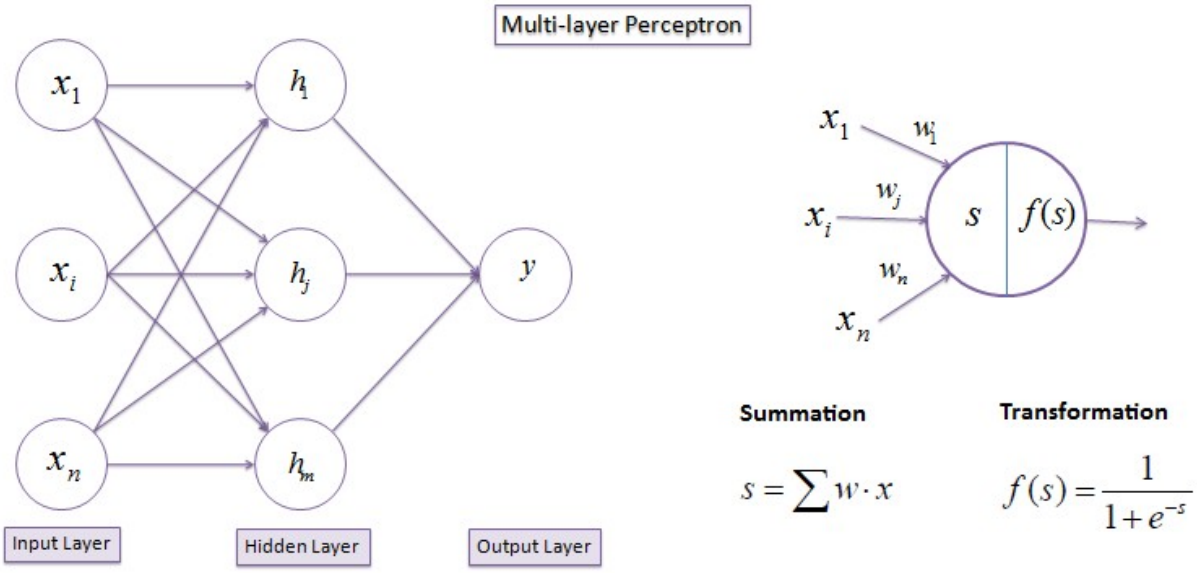
Dla przypomnienia, na wejściu mamy punkty ze zbioru treningowego, czyli d-wymiarowe wektory. W klasyfikacji chcemy znaleźć granicę decyzyjną, czyli krzywą, która oddzieli od siebie klasy. W wejściowej przestrzeni może być to trudne, bo chmury punktów z poszczególnych klas mogą być ze sobą dość pomieszane. Pamiętajmy też, że regresja logistyczna jest klasyfikatorem liniowym, czyli w danej przestrzeni potrafi oddzielić punkty tylko linią prostą.

Sieć MLP składa się z warstw. Każda z nich dokonuje nieliniowego przekształcenia przestrzeni (można o tym myśleć jak o składaniu przestrzeni jakąś prostą/łamaną), tak, aby w finalnej przestrzeni nasze punkty były możliwie liniowo separowalne. Wtedy ostatnia warstwa z sigmoidą będzie potrafiła je rozdzielić od siebie.



Poszczególne neurony składają się z iloczynu skalarnego wejść z wagami neuronu, oraz nieliniowej funkcji aktywacji. W PyTorchu są to osobne obiekty - `nn.Linear` oraz np. `nn.Sigmoid`. Funkcja aktywacji przyjmuje wynik iloczynu skalarnego i przekształca go, aby

sprawdzić, jak mocno reaguje neuron na dane wejście. Musi być nieliniowa z dwóch powodów. Po pierwsze, tylko nieliniowe przekształcenia są na tyle potężne, żeby umożliwić liniową separację danych w ostatniej warstwie. Po drugie, liniowe przekształcenia zwyczajnie nie działają. Aby zrozumieć czemu, trzeba zobaczyć, co matematycznie oznacza sieć MLP.

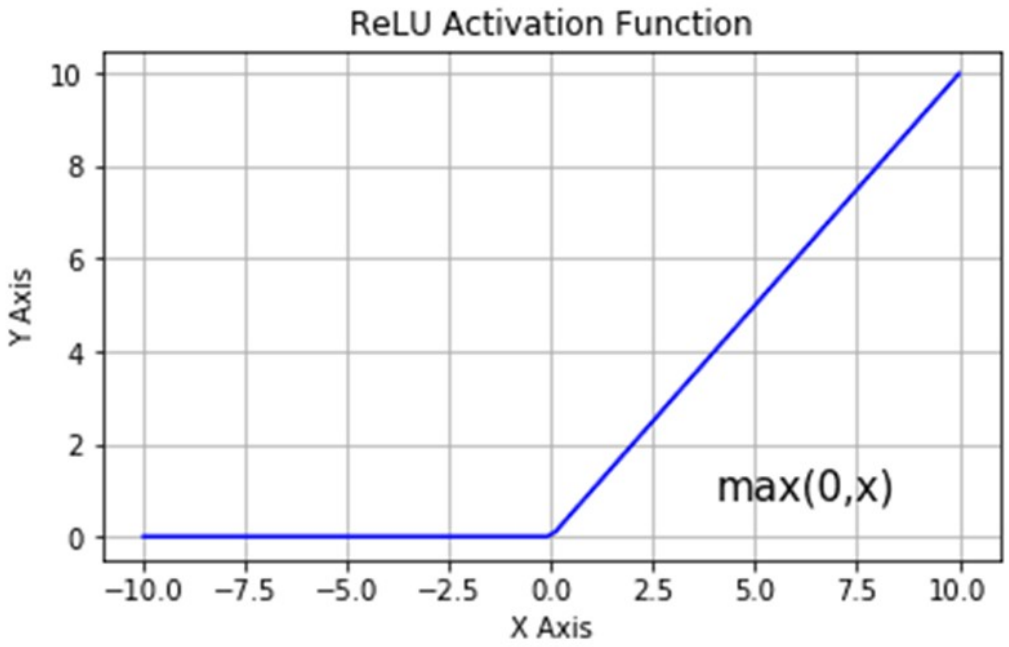


Zapisane matematycznie MLP to: $ h_1 = f_1(x) \ h_2 = f_2(h_1) \ h_3 = f_3(h_2) \ ... h_n = f_n(h_{n-1}) $ gdzie x to wejście f_i to funkcja aktywacji i-tej warstwy, a h_i to wyjście i-tej warstwy, nazywane **ukrytą reprezentacją (hidden representation)**, lub *latent representation*. Nazwa bierze się z tego, że w środku sieci wyciągamy cechy i wzorce w danych, które nie są widoczne na pierwszy rzut oka na wejściu.

Załóżmy, że nie mamy funkcji aktywacji, czyli mamy aktywację liniową $f(x) = x$. Zobaczmy na początku sieci: $ h_1 = f_1(x) = x h_2 = f_2(f_1) = f_2(x) = x ... h_n = f_n(f_{n-1}) = f_n(x) = x $ Jak widać, taka sieć niczego się nie nauczy. Wynika to z tego, że złożenie funkcji liniowych jest także funkcją liniową - patrz notatki z algebry :)

Jeżeli natomiast użyjemy nieliniowej funkcji aktywacji, często oznaczanej jako σ, to wszystko będzie działać. Co ważne, ostatnia warstwa, dająca wyjście sieci, ma zwykle inną aktywację od warstw wewnątrz sieci, bo też ma inne zadanie - zwrócić wartość dla klasyfikacji lub regresji. Na wyjściu korzysta się z funkcji liniowej (regresja), sigmoidalnej (klasyfikacja binarna) lub softmax (klasyfikacja wieloklasowa).

Wewnątrz sieci używano kiedyś sigmoidy oraz tangensa hiperbolicznego `tanh`, ale okazało się to nieefektywne przy uczeniu głębokich sieci o wielu warstwach. Nowoczesne sieci korzystają zwykle z funkcji ReLU (*rectified linear unit*), która jest zaskakująco prosta: $ReLU(x) = max(0, x)$. Okazało się, że bardzo dobrze nadaje się do treningu nawet bardzo głębokich sieci neuronowych. Nowsze funkcje aktywacji są głównie modyfikacjami ReLU.



MLP w PyTorchu

Warstwę neuronów w MLP nazywa się warstwą gęstą (*dense layer*) lub warstwą w pełni połączoną (*fully-connected layer*), i taki opis oznacza zwykle same neurony oraz funkcję aktywacji. PyTorch, jak już widzieliśmy, definiuje osobno transformację liniową oraz aktywację, a więc jedna warstwa składa się de facto z 2 obiektów, wywoływanych jeden po drugim. Inne frameworki, szczególnie wysokopoziomowe (np. Keras) łączą to często w jeden obiekt.

MLP składa się zatem z sekwencji obiektów, które potem wywołuje się jeden po drugim, gdzie wyjście poprzedniego to wejście kolejnego. Ale nie można tutaj używać Pythonowych list! Z perspektywy PyTorcha to wtedy niezależne obiekty i nie zostanie wtedy przekazany między nimi gradient. Trzeba tutaj skorzystać z `nn.Sequential`, aby tworzyć taki pipeline.

Rozmiary wejścia i wyjścia dla każdej warstwy trzeba w PyTorchu podawać explicite. Jest to po pierwsze edukacyjne, a po drugie często ułatwia wnioskowanie o działaniu sieci oraz jej debugowanie - mamy jasno podane, czego oczekujemy. Niektóre frameworki (np. Keras) obliczają to automatycznie.

Co ważne, ostatnia warstwa zwykle nie ma funkcji aktywacji. Wynika to z tego, że obliczanie wielu funkcji kosztu (np. entropii krzyżowej) na aktywacjach jest często niestabilne numerycznie. Z tego powodu PyTorch oferuje funkcje kosztu zawierające w środku aktywację dla ostatniej warstwy, a ich implementacje są stabilne numerycznie. Przykładowo, `nn.BCELoss` przyjmuje wejście z zaaplikowanymi już aktywacjami, ale może skutkować under/overflow, natomiast `nn.BCEWithLogitsLoss` przyjmuje wejście bez aktywacji, a w środku ma specjalną implementację łączącą binarną entropię krzyżową z aktywacją sigmoidalną. Oczywiście w związku z tym aby dokonać potem predykcji w praktyce, trzeba pamiętać o użyciu funkcji aktywacji. Często korzysta się przy tym z funkcji z modułu `torch.nn.functional`, które są w tym wypadku nieco wygodniejsze od klas wywoływalnych z `torch.nn`.

Całe sieci w PyTorchu tworzy się jako klasy dziedziczące po `nn.Module`. Co ważne, obiekty, z których tworzymy sieć, np. `nn.Linear`, także dziedziczą po tej klasie. Pozwala to na bardzo modułową budowę kodu, zgodną z zasadami OOP. W konstruktorze najpierw trzeba zawsze wywołać konstruktor rodzica - `super().__init__()`, a później tworzy się potrzebne obiekty i zapisuje jako atrybuty. Musimy też zdefiniować metodę `forward()`, która przyjmuje tensor `x` i zwraca wynik. Typowo ta metoda po prostu używa obiektów zdefiniowanych w konstruktorze.

UWAGA: nigdy w normalnych warunkach się nie woła metody `forward` ręcznie

Zadanie 4 (1 punkt)

Uzupełnij implementację 3-warstwowej sieci MLP. Użyj rozmiarów:

- pierwsza warstwa: input_size x 256
- druga warstwa: 256 x 128
- trzecia warstwa: 128 x 1

Użyj funkcji aktywacji ReLU.

Przydatne klasy:

- `nn.Sequential`
- `nn.Linear`
- `nn.ReLU`

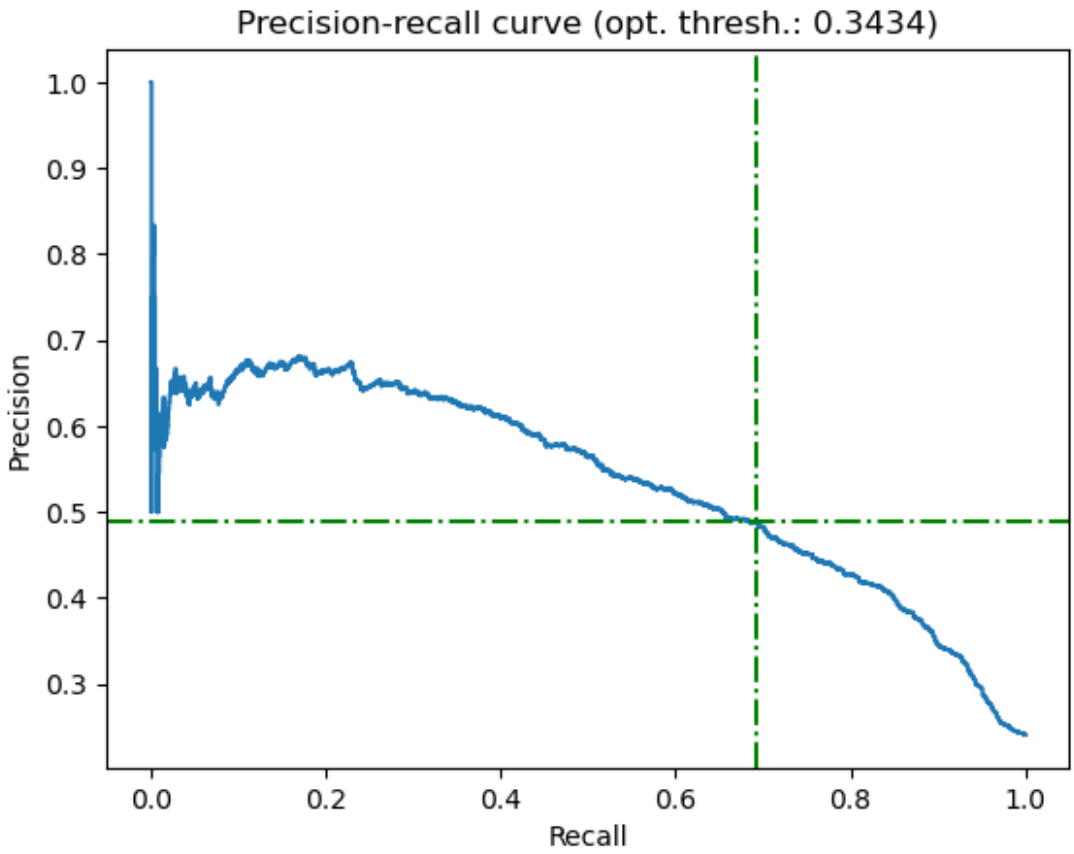
```
from torch import sigmoid

class MLP(nn.Module):
    def __init__(self, input_size: int):
        super().__init__()

        # implement me!
        self.mlp = nn.Sequential(
            nn.Linear(input_size, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )


    def forward(self, x):

        return self.mlp(x)

    def predict_proba(self, x):
        return sigmoid(self(x))

    def predict(self, x):
        y_pred_score = self.predict_proba(x)
        return torch.argmax(y_pred_score, dim=1)
```

```
learning_rate = 1e-3
model = MLP(input_size=X_train.shape[1])
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# note that we are using loss function with sigmoid built in
loss_fn = torch.nn.BCEWithLogitsLoss()
num_epochs = 2000
evaluation_steps = 200

for i in range(num_epochs):
    y_pred = model(X_train)
    loss = loss_fn(y_pred, y_train)
    loss.backward()

    optimizer.step()
    optimizer.zero_grad()

    if i % evaluation_steps == 0:
        print(f"Epoch {i} train loss: {loss.item():.4f}")

print(f"final loss: {loss.item():.4f}")
```

```
Epoch 0 train loss: 0.6891
Epoch 200 train loss: 0.6687
Epoch 400 train loss: 0.6508
Epoch 600 train loss: 0.6349
Epoch 800 train loss: 0.6206
Epoch 1000 train loss: 0.6077
Epoch 1200 train loss: 0.5961
Epoch 1400 train loss: 0.5855
Epoch 1600 train loss: 0.5759
Epoch 1800 train loss: 0.5673
final loss: 0.5594
```

```
model.eval()
with torch.no_grad():
    # positive class probabilities
    y_pred_valid_score = model.predict_proba(X_valid)
    y_pred_test_score = model.predict_proba(X_test)

auroc = roc_auc_score(y_test, y_pred_test_score)
print(f"AUROC: {100 * auroc:.2f}%")

plot_precision_recall_curve(y_valid, y_pred_valid_score)
```

```
AUROC: 79.41%
```



AUROC jest podobne, a precision i recall spadły - wypadamy wręcz gorzej od regresji liniowej! Skoro dodaliśmy więcej warstw, to może pojemność modelu jest teraz za duża i trzeba by go zregularyzować?

Sieci neuronowe bardzo łatwo przeuczają, bo są bardzo elastycznymi i pojemnymi modelami. Dlatego mają wiele różnych rodzajów regularyzacji, których używa się razem. Co ciekawe, udowodniono eksperymentalnie, że zbyt duże sieci z mocną regularyzacją działają lepiej niż mniejsze sieci, odpowiedniego rozmiaru, za to ze słabszą regularyzacją.

Pierwszy rodzaj regularyzacji to znana nam już **regularyzacja L2**, czyli penalizacja zbyt dużych wag. W kontekście sieci neuronowych nazywa się też ją czasem *weight decay*. W PyTorchu dodaje się ją jako argument do optymalizatora.

Regularyzacja specyficzna dla sieci neuronowych to **dropout**. Polega on na losowym wyłączaniu zadanego procenta neuronów podczas treningu. Pomimo prostoty okazała się niesamowicie skuteczna, szczególnie w treningu bardzo głębokich sieci. Co ważne, jest to mechanizm używany tylko podczas treningu - w trakcie predykcji za pomocą sieci wyłącza się ten mechanizm i dokonuje normalnie predykcji całą siecią. Podejście to można potraktować jak ensemble learning, podobny do lasów losowych - wyłączając losowe części sieci, w każdej iteracji trenujemy nieco inną sieć, co odpowiada uśrednianiu predykcji różnych algorytmów. Typowo stosuje się dość mocny dropout, rzędu 25-50%. W PyTorchu implementuje go warstwa `nn.Dropout`, aplikowana zazwyczaj po funkcji aktywacji.

Ostatni, a być może najważniejszy rodzaj regularyzacji to **wczesny stop (early stopping)**. W każdym kroku mocniej dostosowujemy terenową sieć do zbioru treningowego, a więc zbyt długi trening będzie skutkował przeuczeniem. W metodzie wczesnego stopu używamy wydzielonego zbioru walidacyjnego (pojedynczego, metoda holdout), sprawdzając co określoną liczbę epok wynik na tym zbiorze. Jeżeli nie uzyskamy wyniku lepszego od najlepszego dotychczas uzyskanego przez określoną liczbę epok, to przerywamy trening. Okres, przez który czekamy na uzyskanie lepszego wyniku, to cierpliwość (*patience*). Im mniejsze, tym mocniejszy jest ten rodzaj regularyzacji, ale trzeba z tym uważać, bo łatwo jest przesadzić i zbyt szybko przerywać trening. Niektóre implementacje uwzględniają tzw. *grace period*, czyli gwarantowaną minimalną liczbę epok, przez którą będziemy trenować sieć, niezależnie od wybranej cierpliwości.

Dodatkowo ryzyko przeuczenia można zmniejszyć, używając mniejszej stałej uczącej.

Zadanie 5 (1 punkt)

Zaimplementuj funkcję `evaluate_model()`, obliczającą metryki na zbiorze testowym:

- wartość funkcji kosztu (loss)
- AUROC
- optymalny próg
- F1-score przy optymalnym progu
- precyzję oraz recall dla optymalnego progu

Jeżeli podana jest wartość argumentu `threshold`, to użyj jej do zamiany prawdopodobieństw na twarde predykcje. W przeciwnym razie użyj funkcji `get_optimal_threshold` i oblicz optymalną wartość progu.

Pamiętaj o przełączeniu modelu w tryb ewaluacji oraz o wyłączeniu obliczania gradientów.

```
from typing import Optional

from sklearn.metrics import precision_score, recall_score, f1_score
from torch import sigmoid

def evaluate_model(
    model: nn.Module,
    X: torch.Tensor,
    y: torch.Tensor,
    loss_fn: nn.Module,
    threshold: Optional[float]= None
) -> Dict[str, float]:
    # implement me!
    model.eval()


    with torch.no_grad():
        y_pred_score = model.predict_proba(X)
        loss = loss_fn(y_pred_score, y)
        auroc = roc_auc_score(y, y_pred_score)
```

```
    if threshold is None:
        precisions, recalls, thresholds = precision_recall_curve(y,
y_pred_score)
        _, threshold = get_optimal_threshold(precisions, recalls,
thresholds)

    y_pred = (y_pred_score > threshold).to(torch.int32)

    precision = precision_score(y, y_pred)
    recall =recall_score(y, y_pred)
    f1 = f1_score(y, y_pred)

    results = {
        "loss": loss,
        "AUROC": auroc,
        "optimal_threshold": threshold,
        "precision": precision,
        "recall": recall,
        "F1-score": f1,
    }
    return results
```

Zadanie 6 (1 punkt)

Zaimplementuj 3-warstwową sieć MLP z regularyzacją L2 oraz dropout (50%). Rozmiary warstw ukrytych mają wynosić 256 i 128.

```
class RegularizedMLP(nn.Module):
    def __init__(self, input_size: int, dropout_p: float = 0.5):
        super().__init__()

        # implement me!

        self.mlp = nn.Sequential(
            nn.Linear(input_size, 256),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(128, 1)
        )


    def forward(self, x):
        # implement me!
        return self.mlp(x)

    def predict_proba(self, x):
```

```
        return sigmoid(self(x))

    def predict(self, x):
        y_pred_score = self.predict_proba(x)
        return torch.argmax(y_pred_score, dim=1)
```

Opisaliśmy wcześniej podstawowy optymalizator w sieciach neuronowych - spadek wzdłuż gradientu. Jednak wymaga on użycia całego zbioru danych, aby obliczyć gradient, co jest często niewykonalne przez rozmiar zbioru. Dlatego wymyślono **stochastyczny spadek wzdłuż gradientu (stochastic gradient descent, SGD)**, w którym używamy 1 przykładu naraz, liczymy gradient tylko po nim i aktualizujemy parametry. Jest to oczywiście dość grube przybliżenie gradientu, ale pozwala robić szybko dużo małych kroków. Kompromisem, którego używa się w praktyce, jest **minibatch gradient descent**, czyli używanie batchy np. 32, 64 czy 128 przykładów.

Rzadko wspominanym, a ważnym faktem jest także to, że stochastyczność metody optymalizacji jest sama w sobie też metodą regularyzacji, a więc `batch_size` to także hiperparametr.

Obecnie najpopularniejszą odmianą SGD jest Adam, gdyż uczy on szybko sieć oraz daje bardzo dobre wyniki nawet przy niekoniecznie idealnie dobranych hiperparametrach. W PyTorchu najlepiej korzystać z jego implementacji `AdamW`, która jest nieco lepsza niż implementacja `Adam`. Jest to zasadniczo zawsze wybór domyślny przy treningu współczesnych sieci neuronowych.

Na razie użyjemy jednak minibatch SGD.

Poniżej znajduje się implementacja prostej klasy dziedziczącej po `Dataset` - tak w PyTorchu implementuje się własne zbiory danych. Użycie takich klas umożliwia użycie klas ładujących dane (`DataLoader`), które z kolei pozwalają łatwo ładować batche danych. Trzeba w takiej klasie zaimplementować metody:

- `__len__` - zwraca ilość punktów w zbiorze
- `__getitem__` - zwraca przykład ze zbioru pod danym indeksem oraz jego klasę

```
from torch.utils.data import Dataset

class MyDataset(Dataset):
    def __init__(self, data, y):
        super().__init__()

        self.data = data
        self.y = y

    def __len__(self):
        return self.data.shape[0]

    def __getitem__(self, idx):
        return self.data[idx], self.y[idx]
```

Zadanie 7 (2 punkty)

Zaimplementuj pętlę treningowo-walidacyjną dla sieci neuronowej. Wykorzystaj podane wartości hiperparametrów do treningu (stała ucząca, prawdopodobieństwo dropoutu, regularyzacja L2, rozmiar batcha, maksymalna liczba epok). Użyj optymalizatora SGD.

Dodatkowo zaimplementuj regularyzację przez early stopping. Sprawdzaj co epokę wynik na zbiorze walidacyjnym. Użyj podanej wartości patience, a jako metryki po prostu wartości funkcji kosztu. Może się tutaj przydać zaimplementowana funkcja `evaluate_model()`.

Pamiętaj o tym, aby przechowywać najlepszy dotychczasowy wynik walidacyjny oraz najlepszy dotychczasowy model. Zapamiętaj też optymalny próg do klasyfikacji dla najlepszego modelu.

```
from copy import deepcopy

from torch.utils.data import DataLoader


learning_rate = 1e-3
dropout_p = 0.5
l2_reg = 1e-4
batch_size = 128
max_epochs = 300

early_stopping_patience = 4

model = RegularizedMLP(
    input_size=X_train.shape[1],
    dropout_p=dropout_p
)
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=learning_rate,
    weight_decay=l2_reg
)
loss_fn = torch.nn.BCEWithLogitsLoss()

train_dataset = MyDataset(X_train, y_train)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size)

steps_without_improvement = 0

best_val_loss = np.inf
best_model = None
best_threshold = None

for epoch_num in range(max_epochs):
    model.train()

    # note that we are using DataLoader to get batches
    for X_batch, y_batch in train_dataloader:
```

```
        y_pred=model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()



    # model evaluation, early stopping
    # implement me!

    model.eval()
    valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
    if valid_metrics["loss"] < best_val_loss:
        best_val_loss = valid_metrics["loss"]
        best_model = deepcopy(model)
        best_threshold = valid_metrics["optimal_threshold"]
        steps_without_improvement = 0

    else:
        steps_without_improvement += 1

    if steps_without_improvement >= early_stopping_patience:
        break


    print(f"Epoch {epoch_num} train loss: {loss.item():.4f}, eval loss
{valid_metrics['loss']}")
```

```
Epoch 0 train loss: 0.6952, eval loss 0.852538526058197
Epoch 1 train loss: 0.6608, eval loss 0.8447814583778381
Epoch 2 train loss: 0.6508, eval loss 0.837493896484375
Epoch 3 train loss: 0.6314, eval loss 0.8306215405464172
Epoch 4 train loss: 0.6181, eval loss 0.8241122961044312
Epoch 5 train loss: 0.6048, eval loss 0.8179751038551331
Epoch 6 train loss: 0.5869, eval loss 0.812166154384613
Epoch 7 train loss: 0.5832, eval loss 0.806692361831665
Epoch 8 train loss: 0.5811, eval loss 0.8015387654304504
Epoch 9 train loss: 0.5766, eval loss 0.7967032790184021
Epoch 10 train loss: 0.5531, eval loss 0.7921378016471863
Epoch 11 train loss: 0.5406, eval loss 0.7878825664520264
Epoch 12 train loss: 0.5449, eval loss 0.783960223197937
Epoch 13 train loss: 0.5232, eval loss 0.7802971005439758
Epoch 14 train loss: 0.5293, eval loss 0.7769126892089844
Epoch 15 train loss: 0.5323, eval loss 0.7737993001937866
Epoch 16 train loss: 0.5226, eval loss 0.7709004282951355
Epoch 17 train loss: 0.5204, eval loss 0.7682351469993591
Epoch 18 train loss: 0.5104, eval loss 0.7657490968704224
Epoch 19 train loss: 0.5220, eval loss 0.7633755207061768
Epoch 20 train loss: 0.5150, eval loss 0.7612382173538208
```

```
Epoch 21 train loss: 0.5104, eval loss 0.7592003345489502
Epoch 22 train loss: 0.5042, eval loss 0.7572935223579407
Epoch 23 train loss: 0.4959, eval loss 0.7555285096168518
Epoch 24 train loss: 0.4928, eval loss 0.753786563873291
Epoch 25 train loss: 0.4855, eval loss 0.7521377205848694
Epoch 26 train loss: 0.4910, eval loss 0.7505959868431091
Epoch 27 train loss: 0.4699, eval loss 0.7491058111190796
Epoch 28 train loss: 0.4875, eval loss 0.7476587891578674
Epoch 29 train loss: 0.4829, eval loss 0.7462690472602844
Epoch 30 train loss: 0.4724, eval loss 0.7448899149894714
Epoch 31 train loss: 0.4757, eval loss 0.7435784935951233
Epoch 32 train loss: 0.4853, eval loss 0.74232017993927
Epoch 33 train loss: 0.4853, eval loss 0.741034209728241
Epoch 34 train loss: 0.4567, eval loss 0.7398272752761841
Epoch 35 train loss: 0.4685, eval loss 0.7386093735694885
Epoch 36 train loss: 0.4704, eval loss 0.7374585270881653
Epoch 37 train loss: 0.4696, eval loss 0.7363336682319641
Epoch 38 train loss: 0.4641, eval loss 0.7352632284164429
Epoch 39 train loss: 0.4690, eval loss 0.7341976165771484
Epoch 40 train loss: 0.4688, eval loss 0.7331566214561462
Epoch 41 train loss: 0.4368, eval loss 0.7321379780769348
Epoch 42 train loss: 0.4465, eval loss 0.7311496138572693
Epoch 43 train loss: 0.4521, eval loss 0.7301950454711914
Epoch 44 train loss: 0.4501, eval loss 0.7292529344558716
Epoch 45 train loss: 0.4452, eval loss 0.728357195854187
Epoch 46 train loss: 0.4476, eval loss 0.7274801135063171
Epoch 47 train loss: 0.4305, eval loss 0.7265860438346863
Epoch 48 train loss: 0.4292, eval loss 0.7257511615753174
Epoch 49 train loss: 0.4364, eval loss 0.7249653935432434
Epoch 50 train loss: 0.4358, eval loss 0.7241794466972351
Epoch 51 train loss: 0.4211, eval loss 0.723431408405304
Epoch 52 train loss: 0.4399, eval loss 0.7227059602737427
Epoch 53 train loss: 0.4077, eval loss 0.7220008969306946
Epoch 54 train loss: 0.4155, eval loss 0.7212997674942017
Epoch 55 train loss: 0.4387, eval loss 0.7206302881240845
Epoch 56 train loss: 0.4140, eval loss 0.7199241518974304
Epoch 57 train loss: 0.4127, eval loss 0.7193080186843872
Epoch 58 train loss: 0.4113, eval loss 0.7187532782554626
Epoch 59 train loss: 0.4117, eval loss 0.7181420922279358
Epoch 60 train loss: 0.4084, eval loss 0.7175132632255554
Epoch 61 train loss: 0.4021, eval loss 0.7170206904411316
Epoch 62 train loss: 0.4187, eval loss 0.7164224982261658
Epoch 63 train loss: 0.4114, eval loss 0.7158994078636169
Epoch 64 train loss: 0.4440, eval loss 0.7153238654136658
Epoch 65 train loss: 0.4107, eval loss 0.7148650288581848
Epoch 66 train loss: 0.4274, eval loss 0.714397668838501
Epoch 67 train loss: 0.3946, eval loss 0.7139744162559509
Epoch 68 train loss: 0.3884, eval loss 0.7135292887687683
Epoch 69 train loss: 0.3949, eval loss 0.7131195068359375
```

```
Epoch 70 train loss: 0.3873, eval loss 0.7127622961997986
Epoch 71 train loss: 0.3901, eval loss 0.7122630476951599
Epoch 72 train loss: 0.4078, eval loss 0.711896538734436
Epoch 73 train loss: 0.4147, eval loss 0.7115167379379272
Epoch 74 train loss: 0.3961, eval loss 0.7111409306526184
Epoch 75 train loss: 0.3889, eval loss 0.7108055353164673
Epoch 76 train loss: 0.3863, eval loss 0.7104467153549194
Epoch 77 train loss: 0.3640, eval loss 0.7100492119789124
Epoch 78 train loss: 0.3833, eval loss 0.7097035646438599
Epoch 79 train loss: 0.4108, eval loss 0.7094058990478516
Epoch 80 train loss: 0.3774, eval loss 0.7090610265731812
Epoch 81 train loss: 0.3659, eval loss 0.7087874412536621
Epoch 82 train loss: 0.3757, eval loss 0.7085199952125549
Epoch 83 train loss: 0.3991, eval loss 0.7082452774047852
Epoch 84 train loss: 0.3910, eval loss 0.7080349326133728
Epoch 85 train loss: 0.4141, eval loss 0.7078137397766113
Epoch 86 train loss: 0.4271, eval loss 0.7075047492980957
Epoch 87 train loss: 0.3986, eval loss 0.7072383761405945
Epoch 88 train loss: 0.4168, eval loss 0.7070150971412659
Epoch 89 train loss: 0.3864, eval loss 0.7068608403205872
Epoch 90 train loss: 0.3910, eval loss 0.7066547870635986
Epoch 91 train loss: 0.3726, eval loss 0.7063753008842468
Epoch 92 train loss: 0.4139, eval loss 0.7061708569526672
Epoch 93 train loss: 0.3785, eval loss 0.7059900760650635
Epoch 94 train loss: 0.3740, eval loss 0.7057666778564453
Epoch 95 train loss: 0.4002, eval loss 0.7055736184120178
Epoch 96 train loss: 0.3901, eval loss 0.7053857445716858
Epoch 97 train loss: 0.4095, eval loss 0.7051407098770142
Epoch 98 train loss: 0.4093, eval loss 0.704922080039978
Epoch 99 train loss: 0.4121, eval loss 0.7047613859176636
Epoch 100 train loss: 0.3513, eval loss 0.7045262455940247
Epoch 101 train loss: 0.3691, eval loss 0.7043920755386353
Epoch 102 train loss: 0.3850, eval loss 0.7042363882064819
Epoch 103 train loss: 0.4193, eval loss 0.7040722370147705
Epoch 104 train loss: 0.4016, eval loss 0.7039257884025574
Epoch 105 train loss: 0.4023, eval loss 0.7037689685821533
Epoch 106 train loss: 0.3935, eval loss 0.7036698460578918
Epoch 107 train loss: 0.3707, eval loss 0.7034748792648315
Epoch 108 train loss: 0.3985, eval loss 0.7033095359802246
Epoch 109 train loss: 0.4091, eval loss 0.7031876444816589
Epoch 110 train loss: 0.3773, eval loss 0.7030513286590576
Epoch 111 train loss: 0.4277, eval loss 0.7028793692588806
Epoch 112 train loss: 0.4103, eval loss 0.7027419209480286
Epoch 113 train loss: 0.3799, eval loss 0.7026118636131287
Epoch 114 train loss: 0.3889, eval loss 0.7024964690208435
Epoch 115 train loss: 0.3656, eval loss 0.7023651599884033
Epoch 116 train loss: 0.3817, eval loss 0.702237606048584
Epoch 117 train loss: 0.3754, eval loss 0.7020906209945679
Epoch 118 train loss: 0.3958, eval loss 0.7020364999771118
```

```
Epoch 119 train loss: 0.3936, eval loss 0.7019681930541992
Epoch 120 train loss: 0.3584, eval loss 0.7018890976905823
Epoch 121 train loss: 0.4162, eval loss 0.7017627954483032
Epoch 122 train loss: 0.3866, eval loss 0.7016105055809021
Epoch 123 train loss: 0.3911, eval loss 0.7015247344970703
Epoch 124 train loss: 0.3842, eval loss 0.7013405561447144
Epoch 125 train loss: 0.3557, eval loss 0.7012491822242737
Epoch 126 train loss: 0.3928, eval loss 0.7011481523513794
Epoch 127 train loss: 0.3931, eval loss 0.7010536193847656
Epoch 128 train loss: 0.3852, eval loss 0.7010198831558228
Epoch 129 train loss: 0.3875, eval loss 0.7009243965148926
Epoch 130 train loss: 0.3514, eval loss 0.7008318901062012
Epoch 131 train loss: 0.3853, eval loss 0.7007560729980469
Epoch 132 train loss: 0.3844, eval loss 0.7006420493125916
Epoch 133 train loss: 0.4046, eval loss 0.7005570530891418
Epoch 134 train loss: 0.3663, eval loss 0.7004724740982056
Epoch 135 train loss: 0.3835, eval loss 0.7003970146179199
Epoch 136 train loss: 0.3830, eval loss 0.7003124356269836
Epoch 137 train loss: 0.4025, eval loss 0.7001650333404541
Epoch 138 train loss: 0.4112, eval loss 0.7000963687896729
Epoch 139 train loss: 0.4050, eval loss 0.7000649571418762
Epoch 140 train loss: 0.3711, eval loss 0.6999674439430237
Epoch 141 train loss: 0.3466, eval loss 0.6998969316482544
Epoch 142 train loss: 0.4078, eval loss 0.6997929215431213
Epoch 143 train loss: 0.4043, eval loss 0.6997553110122681
Epoch 144 train loss: 0.3989, eval loss 0.6997184157371521
Epoch 145 train loss: 0.3699, eval loss 0.6996532082557678
Epoch 146 train loss: 0.3699, eval loss 0.6995357275009155
Epoch 147 train loss: 0.3792, eval loss 0.6994367241859436
Epoch 148 train loss: 0.3904, eval loss 0.6994823217391968
Epoch 149 train loss: 0.3782, eval loss 0.699437141418457
Epoch 150 train loss: 0.3821, eval loss 0.6993144154548645
Epoch 151 train loss: 0.3675, eval loss 0.6992172598838806
Epoch 152 train loss: 0.3735, eval loss 0.6990756392478943
Epoch 153 train loss: 0.3671, eval loss 0.6989575624465942
Epoch 154 train loss: 0.3686, eval loss 0.698883056640625
Epoch 155 train loss: 0.3853, eval loss 0.6988019943237305
Epoch 156 train loss: 0.3963, eval loss 0.6987795233726501
Epoch 157 train loss: 0.3804, eval loss 0.698716402053833
Epoch 158 train loss: 0.3784, eval loss 0.698591411113739
Epoch 159 train loss: 0.3792, eval loss 0.6985664963722229
Epoch 160 train loss: 0.3984, eval loss 0.6984566450119019
Epoch 161 train loss: 0.3825, eval loss 0.6983974575996399
Epoch 162 train loss: 0.3923, eval loss 0.6983429789543152
Epoch 163 train loss: 0.4139, eval loss 0.6982354521751404
Epoch 164 train loss: 0.3662, eval loss 0.6981289982795715
Epoch 165 train loss: 0.3819, eval loss 0.6980928778648376
Epoch 166 train loss: 0.3797, eval loss 0.6980288624763489
Epoch 167 train loss: 0.3463, eval loss 0.6980435252189636
```

```
Epoch 168 train loss: 0.3494, eval loss 0.6980633735656738
Epoch 169 train loss: 0.3873, eval loss 0.6979510188102722
Epoch 170 train loss: 0.3515, eval loss 0.6979514956474304
Epoch 171 train loss: 0.3540, eval loss 0.6978887915611267
Epoch 172 train loss: 0.3704, eval loss 0.6978268027305603
Epoch 173 train loss: 0.3431, eval loss 0.6977304816246033
Epoch 174 train loss: 0.3696, eval loss 0.6976357102394104
Epoch 175 train loss: 0.3548, eval loss 0.697573184967041
Epoch 176 train loss: 0.3850, eval loss 0.6975356340408325
Epoch 177 train loss: 0.3890, eval loss 0.697443962097168
Epoch 178 train loss: 0.4278, eval loss 0.6974478960037231
Epoch 179 train loss: 0.3440, eval loss 0.6974015235900879
Epoch 180 train loss: 0.3854, eval loss 0.6973325610160828
Epoch 181 train loss: 0.3564, eval loss 0.6972711086273193
Epoch 182 train loss: 0.3906, eval loss 0.6972498893737793
Epoch 183 train loss: 0.3611, eval loss 0.6971007585525513
Epoch 184 train loss: 0.3777, eval loss 0.6971412301063538
Epoch 185 train loss: 0.3518, eval loss 0.6970838904380798
Epoch 186 train loss: 0.3566, eval loss 0.6970468163490295
Epoch 187 train loss: 0.3617, eval loss 0.6969197988510132
Epoch 188 train loss: 0.3714, eval loss 0.6969252228736877
Epoch 189 train loss: 0.4073, eval loss 0.6968162059783936
Epoch 190 train loss: 0.3772, eval loss 0.696768581867218
Epoch 191 train loss: 0.3768, eval loss 0.6967877745628357
Epoch 192 train loss: 0.3563, eval loss 0.6967105269432068
Epoch 193 train loss: 0.3525, eval loss 0.6965380311012268
Epoch 194 train loss: 0.3853, eval loss 0.6966174244880676
Epoch 195 train loss: 0.4072, eval loss 0.6966013312339783
Epoch 196 train loss: 0.3795, eval loss 0.6964720487594604
Epoch 197 train loss: 0.3608, eval loss 0.6964084506034851
Epoch 198 train loss: 0.3737, eval loss 0.6963383555412292
Epoch 199 train loss: 0.3546, eval loss 0.6963152289390564
Epoch 200 train loss: 0.3774, eval loss 0.6962599158287048
Epoch 201 train loss: 0.3616, eval loss 0.6962199807167053
Epoch 202 train loss: 0.3907, eval loss 0.6962152719497681
Epoch 203 train loss: 0.4123, eval loss 0.6961861848831177
Epoch 204 train loss: 0.3775, eval loss 0.6961241364479065
Epoch 205 train loss: 0.3333, eval loss 0.6960281133651733
Epoch 206 train loss: 0.3506, eval loss 0.6959818005561829
Epoch 207 train loss: 0.3711, eval loss 0.6959806084632874
Epoch 208 train loss: 0.3537, eval loss 0.6959847807884216
Epoch 209 train loss: 0.3824, eval loss 0.6959474086761475
Epoch 210 train loss: 0.3894, eval loss 0.695905864238739
Epoch 211 train loss: 0.3524, eval loss 0.6958378553390503
Epoch 212 train loss: 0.3314, eval loss 0.6957435011863708
Epoch 213 train loss: 0.4224, eval loss 0.6957308650016785
Epoch 214 train loss: 0.3718, eval loss 0.6957226991653442
Epoch 215 train loss: 0.3725, eval loss 0.6956433653831482
Epoch 216 train loss: 0.3727, eval loss 0.6955615878105164
```

```
Epoch 217 train loss: 0.3851, eval loss 0.6954931020736694
Epoch 218 train loss: 0.3576, eval loss 0.6954536437988281
Epoch 219 train loss: 0.3658, eval loss 0.6953887939453125
Epoch 220 train loss: 0.3812, eval loss 0.6953696012496948
Epoch 221 train loss: 0.3755, eval loss 0.695427656173706
Epoch 222 train loss: 0.3471, eval loss 0.6953871846199036
Epoch 223 train loss: 0.3636, eval loss 0.6953979730606079
Epoch 224 train loss: 0.3625, eval loss 0.6953356266021729
Epoch 225 train loss: 0.3444, eval loss 0.6953402161598206
Epoch 226 train loss: 0.3886, eval loss 0.6953089833259583
Epoch 227 train loss: 0.3767, eval loss 0.6952751278877258
Epoch 228 train loss: 0.3556, eval loss 0.6952149868011475
Epoch 229 train loss: 0.3542, eval loss 0.6950907111167908
Epoch 230 train loss: 0.3772, eval loss 0.695110559463501
Epoch 231 train loss: 0.3695, eval loss 0.6950029730796814
Epoch 232 train loss: 0.3867, eval loss 0.6949502825737
Epoch 233 train loss: 0.3525, eval loss 0.6949299573898315
Epoch 234 train loss: 0.3476, eval loss 0.6948167681694031
Epoch 235 train loss: 0.4098, eval loss 0.6947630047798157
Epoch 236 train loss: 0.3442, eval loss 0.6946502923965454
Epoch 237 train loss: 0.3597, eval loss 0.6947042942047119
Epoch 238 train loss: 0.3805, eval loss 0.6947042942047119
Epoch 239 train loss: 0.3496, eval loss 0.6945981979370117
Epoch 240 train loss: 0.3787, eval loss 0.6945808529853821
Epoch 241 train loss: 0.3751, eval loss 0.6945599317550659
Epoch 242 train loss: 0.3750, eval loss 0.6945001482963562
Epoch 243 train loss: 0.3760, eval loss 0.69443678855896
Epoch 244 train loss: 0.3756, eval loss 0.6944568157196045
Epoch 245 train loss: 0.3709, eval loss 0.6943949460983276
Epoch 246 train loss: 0.3452, eval loss 0.6943833827972412
Epoch 247 train loss: 0.3878, eval loss 0.6943016648292542
Epoch 248 train loss: 0.3382, eval loss 0.6942453384399414
Epoch 249 train loss: 0.3598, eval loss 0.694162905216217
Epoch 250 train loss: 0.3722, eval loss 0.6942481994628906
Epoch 251 train loss: 0.3545, eval loss 0.6942470669746399
Epoch 252 train loss: 0.3822, eval loss 0.694223940372467
```

```
test_metrics = evaluate_model(best_model, X_test, y_test, loss_fn,
best_threshold)

print(f"AUROC: {100 * test_metrics['AUROC']:.2f}%")
print(f"F1: {100 * test_metrics['F1-score']:.2f}%")
print(f"Precision: {100 * test_metrics['precision']:.2f}%")
print(f"Recall: {100 * test_metrics['recall']:.2f}%")
```

```
AUROC: 89.95%
F1: 67.67%
Precision: 57.79%
Recall: 81.63%
```

Wyniki wyglądają już dużo lepiej.

Na koniec laboratorium dołożymy do naszego modelu jeszcze 3 powrzechnie używane techniki, które są bardzo proste, a pozwalają często ulepszyć wynik modelu.

Pierwszą z nich są **warstwy normalizacji (normalization layers)**. Powstały one początkowo z założeniem, że przez przekształcenia przestrzeni dokonywane przez sieć zmienia się rozkład prawdopodobieństw pomiędzy warstwami, czyli tzw. *internal covariate shift*. Później okazało się, że zastosowanie takiej normalizacji wygładza powierzchnię funkcji kosztu, co ułatwia i przyspiesza optymalizację. Najpowszechniej używaną normalizacją jest **batch normalization (batch norm)**.

Drugim ulepszeniem jest dodanie **wag klas (class weights)**. Mamy do czynienia z problemem klasyfikacji niezbalansowanej, więc klasa mniejszościowa, ważniejsza dla nas, powinna dostać większą wagę. Implementuje się to trywialnie prosto - po prostu mnożymy wartość funkcji kosztu dla danego przykładu przez wagę dla prawdziwej klasy tego przykładu. Praktycznie każdy klasyfikator operujący na jakiejś ważonej funkcji może działać w ten sposób, nie tylko sieci neuronowe.

Ostatnim ulepszeniem jest zamiana SGD na optymalizator Adam, a konkretnie na optymalizator `AdamW`. Jest to przykład **optymalizatora adaptacyjnego (adaptive optimizer)**, który potrafi zaadaptować stałą uczącą dla każdego parametru z osobna w trakcie treningu. Wykorzystuje do tego gradienty - w uproszczeniu, im większa wariancja gradientu, tym mniejsze kroki w tym kierunku robimy.

Zadanie 8 (1 punkt)

Zaimplementuj model `NormalizingMLP`, o takiej samej strukturze jak `RegularizedMLP`, ale dodatkowo z warstwami `BatchNorm1d` pomiędzy warstwami `Linear` oraz `ReLU`.

Za pomocą funkcji `compute_class_weight()` oblicz wagi dla poszczególnych klas. Użyj opcji `"balanced"`. Przekaż do funkcji kosztu wagę klasy pozytywnej (pamiętaj, aby zamienić ją na tensor).

Zamień używany optymalizator na `AdamW`.

Na koniec skopiuj resztę kodu do treningu z poprzedniego zadania, wytrenuj sieć i oblicz wyniki na zbiorze testowym.

```
class NormalizingMLP(nn.Module):
    def __init__(self, input_size: int, dropout_p: float = 0.5):
        super().__init__()

        # implement me!

        self.mlp = nn.Sequential(
            nn.Linear(input_size, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(256, 128),
```

```
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(128, 1)

        )

    def forward(self, x):
        return self.mlp(x)

    def predict_proba(self, x):
        return sigmoid(self(x))

    def predict(self, x):
        y_pred_score = self.predict_proba(x)
        return torch.argmax(y_pred_score, dim=1)

from sklearn.utils.class_weight import compute_class_weight

weights = compute_class_weight(
    class_weight="balanced",classes=np.unique(y_train),
y=np.ravel(y_train)
)

learning_rate = 1e-3
dropout_p = 0.5
l2_reg = 1e-4
batch_size = 128
max_epochs = 300

early_stopping_patience = 4

model = NormalizingMLP(
    input_size=X_train.shape[1],
    dropout_p=dropout_p
)
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=learning_rate,
    weight_decay=l2_reg
)
loss_fn =
torch.nn.BCEWithLogitsLoss(pos_weight=torch.from_numpy(weights)[1])

train_dataset = MyDataset(X_train, y_train)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size)

steps_without_improvement = 0

best_val_loss = np.inf
```

```
best_model = None
best_threshold = None

for epoch_num in range(max_epochs):
    model.train()

    # note that we are using DataLoader to get batches
    for X_batch, y_batch in train_dataloader:
        y_pred=model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()



    # model evaluation, early stopping
    # implement me!

    model.eval()
    valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
    if valid_metrics["loss"] < best_val_loss:
        best_val_loss = valid_metrics["loss"]
        best_model = deepcopy(model)
        best_threshold = valid_metrics["optimal_threshold"]
        steps_without_improvement = 0

    else:
        steps_without_improvement += 1

    print(f"Epoch {epoch_num} train loss: {loss.item():.4f}, eval loss
{valid_metrics['loss']}")

    if steps_without_improvement >= early_stopping_patience:
        break
```

```
Epoch 0 train loss: 0.5506, eval loss 0.8218947052955627
Epoch 1 train loss: 0.5431, eval loss 0.8182551860809326
Epoch 2 train loss: 0.5506, eval loss 0.8153528571128845
Epoch 3 train loss: 0.5921, eval loss 0.8151989579200745
Epoch 4 train loss: 0.6590, eval loss 0.8140314817428589
Epoch 5 train loss: 0.5026, eval loss 0.8114859461784363
Epoch 6 train loss: 0.5528, eval loss 0.8116480708122253
Epoch 7 train loss: 0.5790, eval loss 0.8114549517631531
Epoch 8 train loss: 0.5774, eval loss 0.811625599861145
Epoch 9 train loss: 0.4704, eval loss 0.8111162185668945
Epoch 10 train loss: 0.5454, eval loss 0.8113364577293396
Epoch 11 train loss: 0.6561, eval loss 0.8099157810211182
Epoch 12 train loss: 0.5214, eval loss 0.8092721700668335
Epoch 13 train loss: 0.6115, eval loss 0.8097178936004639
```

```
Epoch 14 train loss: 0.5263, eval loss 0.809548020362854
Epoch 15 train loss: 0.5389, eval loss 0.8084733486175537
Epoch 16 train loss: 0.4307, eval loss 0.8092349767684937
Epoch 17 train loss: 0.4888, eval loss 0.8085934519767761
Epoch 18 train loss: 0.5106, eval loss 0.8089302182197571
Epoch 19 train loss: 0.4474, eval loss 0.8089715242385864
```

```
test_metrics = evaluate_model(best_model, X_test, y_test, loss_fn,
best_threshold)

print(f"AUROC: {100 * test_metrics['AUROC']:.2f}%")
print(f"F1: {100 * test_metrics['F1-score']:.2f}%")
print(f"Precision: {100 * test_metrics['precision']:.2f}%")
print(f"Recall: {100 * test_metrics['recall']:.2f}%")
```

```
AUROC: 90.74%
F1: 69.52%
Precision: 61.95%
Recall: 79.21%
```

Pytania kontrolne (1 punkt)

1. Wymień 4 najważniejsze twoim zdaniem hiperparametry sieci neuronowej.
2. Czy widzisz jakiś problem w użyciu regularyzacji L1 w treningu sieci neuronowych? Czy dropout może twoim zdaniem stanowić alternatywę dla tego rodzaju regularyzacji?
3. Czy użycie innej metryki do wczesnego stopu da taki sam model końcowy? Czemu?

1 Learning Rate, Rozmiar batcha, liczba warstw sieci, funkcja aktywacji

2 Nie ma niczego złego w użyciu regularyzacji L1 w treningu sieci neuronowych. Regularyzacja L1 może być użyteczna do wprowadzenia rzadkości w wagach, co może być korzystne w przypadku, gdy chcemy, aby niektóre wagi były dokładnie równe zero. Dropout, który losowo eliminuje niektóre neurony podczas treningu, może stanowić alternatywę dla regularyzacji L1. Oba te mechanizmy mają na celu zapobieganie przeuczeniu poprzez ograniczenie złożoności modelu i poprawę jego generalizacji

3 Użycie innej metryki do wczesnego stopu może prowadzić do różnych modeli końcowych. Wczesne zatrzymanie z użyciem metryki precyzji może prowadzić do modelu, który ma wysoką precyzję, ale niską czułość. Z drugiej strony, wczesne zatrzymanie z użyciem metryki czułości może prowadzić do modelu, który ma wysoką czułość, ale niską precyzję.

Akceleracja sprzętowa (dla zainteresowanych)

Jak wcześniej wspominaliśmy, użycie akceleracji sprzętowej, czyli po prostu GPU do obliczeń, jest bardzo efektywne w przypadku sieci neuronowych. Karty graficzne bardzo efektywnie mnożą macierze, a sieci neuronowe to, jak można było się przekonać, dużo mnożenia macierzy.

W PyTorchu jest to dosyć łatwe, ale trzeba robić to explicite. Służy do tego metoda `.to()`, która przenosi tensory między CPU i GPU. Poniżej przykład, jak to się robi (oczywiście trzeba mieć skonfigurowane GPU, żeby działało):

```
import time

model = NormalizingMLP(
    input_size=X_train.shape[1],
    dropout_p=dropout_p
).to('cuda')

optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate,
weight_decay=1e-4)

# note that we are using loss function with sigmoid built in
loss_fn =
torch.nn.BCEWithLogitsLoss(pos_weight=torch.from_numpy(weights)
[1].to('cuda'))

step_counter = 0
time_from_eval = time.time()
for epoch_id in range(30):
    for batch_x, batch_y in train_dataloader:
        batch_x = batch_x.to('cuda')
        batch_y = batch_y.to('cuda')

        loss = loss_fn(model(batch_x), batch_y)
        loss.backward()

        optimizer.step()
        optimizer.zero_grad()

        if step_counter % evaluation_steps == 0:
            print(f"Epoch {epoch_id} train loss: {loss.item():.4f},
time: {time.time() - time_from_eval}")
            time_from_eval = time.time()

        step_counter += 1

test_res = evaluate_model(model.to('cpu'), X_test, y_test,
loss_fn.to('cpu'), threshold=0.5)

print(f"AUROC: {100 * test_metrics['AUROC']:.2f}%")
print(f"F1: {100 * test_metrics['F1-score']:.2f}%")
```

```
PyTorch version: 2.1.0
**********
_CUDA version:
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
```

```
Built on Fri_Nov__3_17:51:05_Pacific_Daylight_Time_2023
Cuda compilation tools, release 12.3, V12.3.103
Build cuda_12.3.r12.3/compiler.33492891_0
**********
CUDNN version: None
Available GPU devices: 0

---------------------------------------------------------------------------
-----
AssertionError                            Traceback (most recent call
last)
Cell In[4], line 9
      7 print(f'CUDNN version: {torch.backends.cudnn.version()}')
      8 print(f'Available GPU devices: {torch.cuda.device_count()}')
----> 9 print(f'Device Name: {torch.cuda.get_device_name()}')
     10 import time
     12 model = NormalizingMLP(
     13     input_size=X_train.shape[1],
     14     dropout_p=dropout_p
     15 ).to('cuda')

File ~\anaconda3\envs\PSI\lib\site-packages\torch\cuda\
__init__.py:419, in get_device_name(device)
    407 def get_device_name(device: Optional[_device_t] = None) ->
str:
    408     r"""Gets the name of a device.
    409
    410     Args:
   (...)
    417         str: the name of the device
    418     """
--> 419     return get_device_properties(device).name

File ~\anaconda3\envs\PSI\lib\site-packages\torch\cuda\
__init__.py:449, in get_device_properties(device)
    439 def get_device_properties(device: _device_t) ->
_CudaDeviceProperties:
    440     r"""Gets the properties of a device.
    441
    442     Args:
   (...)
    447         _CudaDeviceProperties: the properties of the device
    448     """
--> 449     _lazy_init()  # will define _get_device_properties
    450     device = _get_device_index(device, optional=True)
    451     if device < 0 or device >= device_count():

File ~\anaconda3\envs\PSI\lib\site-packages\torch\cuda\
__init__.py:289, in _lazy_init()
    284     raise RuntimeError(
```

```
    285         "Cannot re-initialize CUDA in forked subprocess. To
use CUDA with "
    286         "multiprocessing, you must use the 'spawn' start
method"
    287     )
    288 if not hasattr(torch._C, "_cuda_getDeviceCount"):
--> 289     raise AssertionError("Torch not compiled with CUDA
enabled")
    290 if _cudart is None:
    291     raise AssertionError(
    292         "libcudart functions unavailable. It looks like you
have a broken build?"
    293     )

AssertionError: Torch not compiled with CUDA enabled
```

Wyniki mogą się różnić z modelem na CPU, zauważ o ile szybszy jest ten model w porównaniu z CPU (przynajmniej w przypadków scenariuszy tak będzie ;)).

Dla zainteresowanych polecamy tę serie artykułów

Zadanie dla chętnych

Jak widzieliśmy, sieci neuronowe mają bardzo dużo hiperparametrów. Przeszukiwanie ich grid search'em jest więc niewykonalne, a chociaż random search by działał, to potrzebowałby wielu iteracji, co też jest kosztowne obliczeniowo.

Zaimplementuj inteligentne przeszukiwanie przestrzeni hiperparametrów za pomocą biblioteki Optuna. Implementuje ona między innymi algorytm Tree Parzen Estimator (TPE), należący do grupy algorytmów typu Bayesian search. Typowo osiągają one bardzo dobre wyniki, a właściwie zawsze lepsze od przeszukiwania losowego. Do tego wystarcza im często niewielka liczba kroków.

Zaimplementuj 3-warstwową sieć MLP, gdzie pierwsza warstwa ma rozmiar ukryty N, a druga N // 2. Ucz ją optymalizatorem Adam przez maksymalnie 300 epok z cierpliwością 10.

Przeszukaj wybrane zakresy dla hiperparametrów:

- rozmiar warstw ukrytych (N)
- stała ucząca
- batch size
- siła regularyzacji L2
- prawdopodobieństwo dropoutu

Wykorzystaj przynajmniej 30 iteracji. Następnie przełącz algorytm na losowy (Optuna także jego implementuje), wykonaj 30 iteracji i porównaj jakość wyników.

Przydatne materiały:

- Optuna code examples - PyTorch

- Auto-Tuning Hyperparameters with Optuna and PyTorch
- Hyperparameter Tuning of Neural Networks with Optuna and PyTorch
- Using Optuna to Optimize PyTorch Hyperparameters