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
```

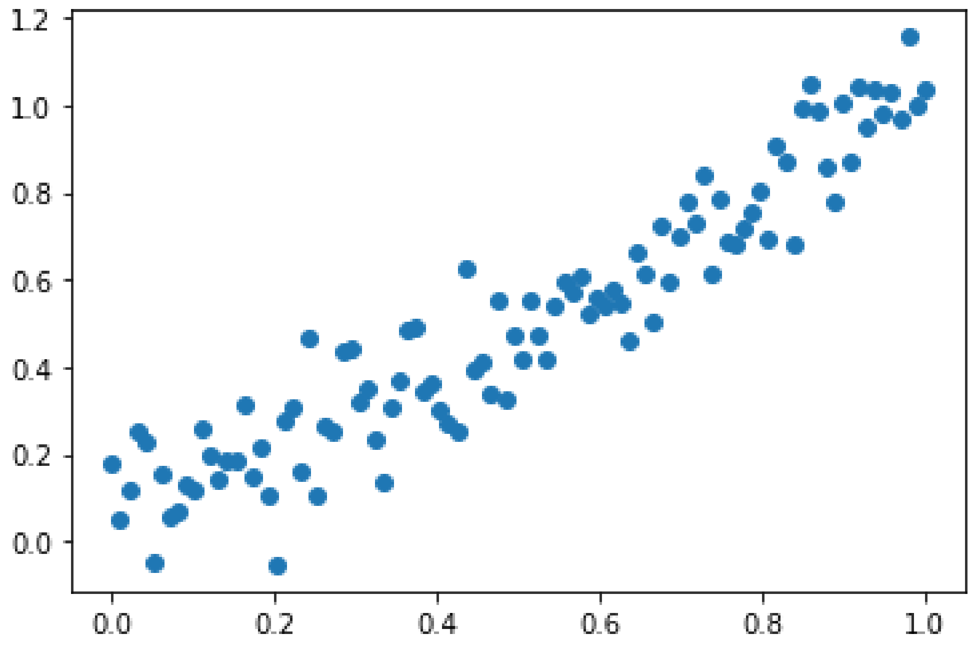
```
Collecting package metadata (current_repodata.json): ...working...
done
Solving environment: ...working... done

# All requested packages already installed.
```

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 0x2336d2d7310>
```



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.

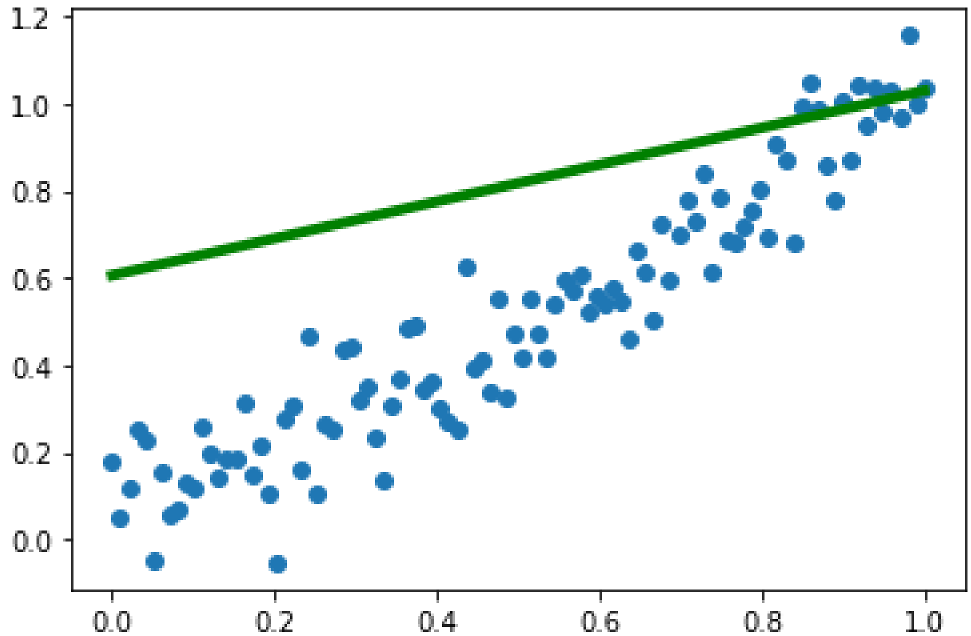
Zadanie 1 (0.5 punkt)

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:
    return np.sum(np.square(y - y_hat)) / y.shape[0]

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: 0.133
```

```
[<matplotlib.lines.Line2D at 0x2336d3f19d0>]
```



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.

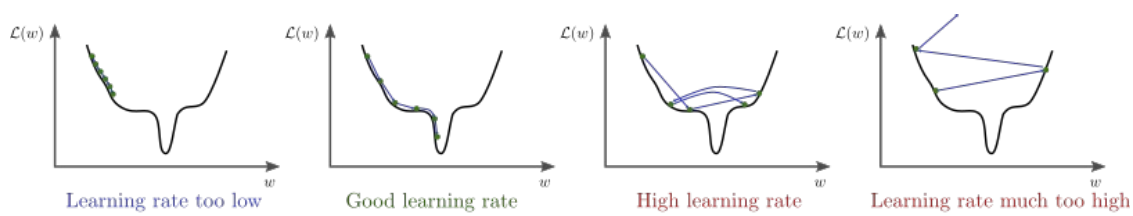
Stosując powyższe do optymalizacji, mamy:

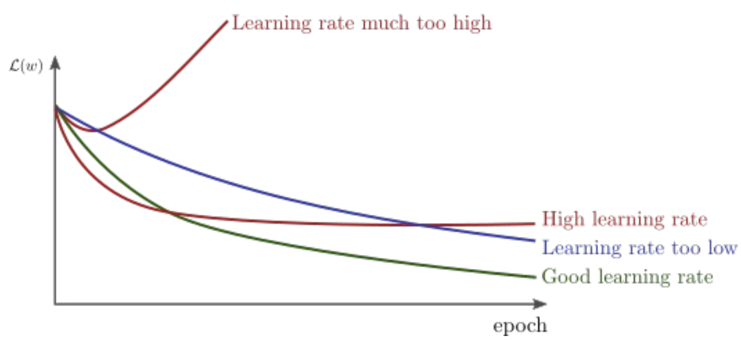
$$\large x_{t+1} = x_{t} - \alpha * \frac{f(x)}{dx} $$

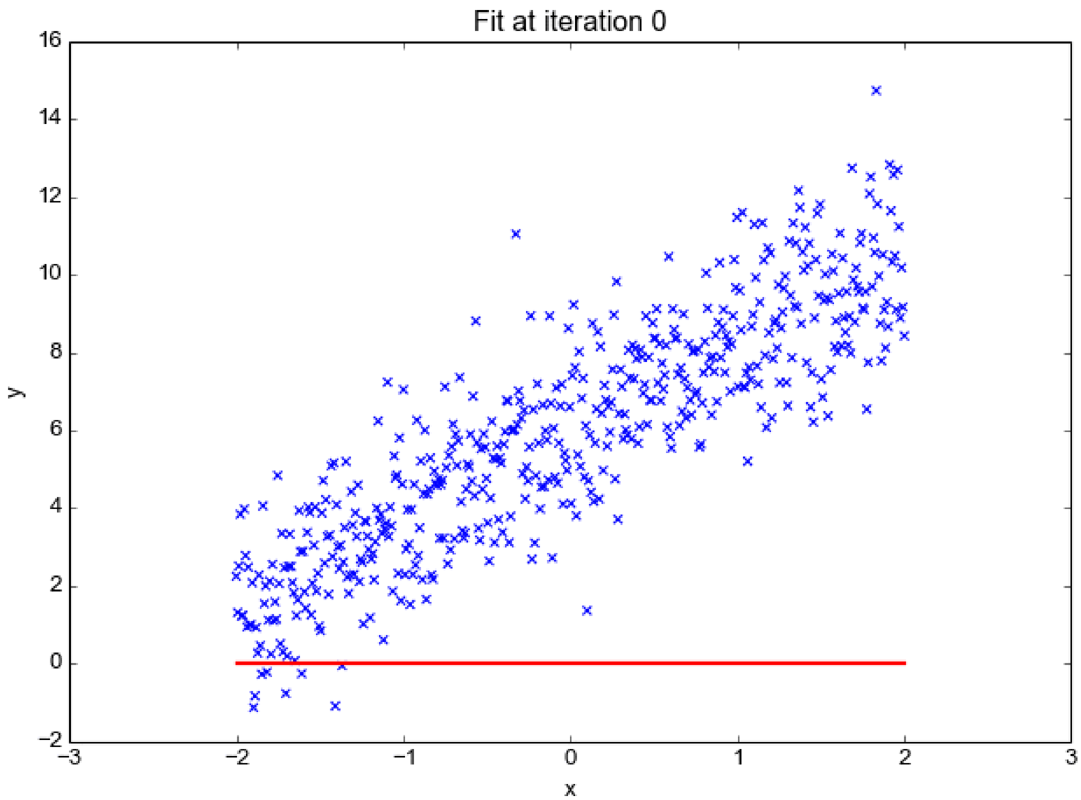
α to niewielka wartość (rzędu zwykle 10^{-5} - 10^{-2}), wprowadzona, aby trzymać się założenia o małej zmianie parametrów (ϵ). Nazywa się ją **stałą uczącą (learning rate)** i jest zwykle najważniejszym hiperparametrem podczas nauki sieci.

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)

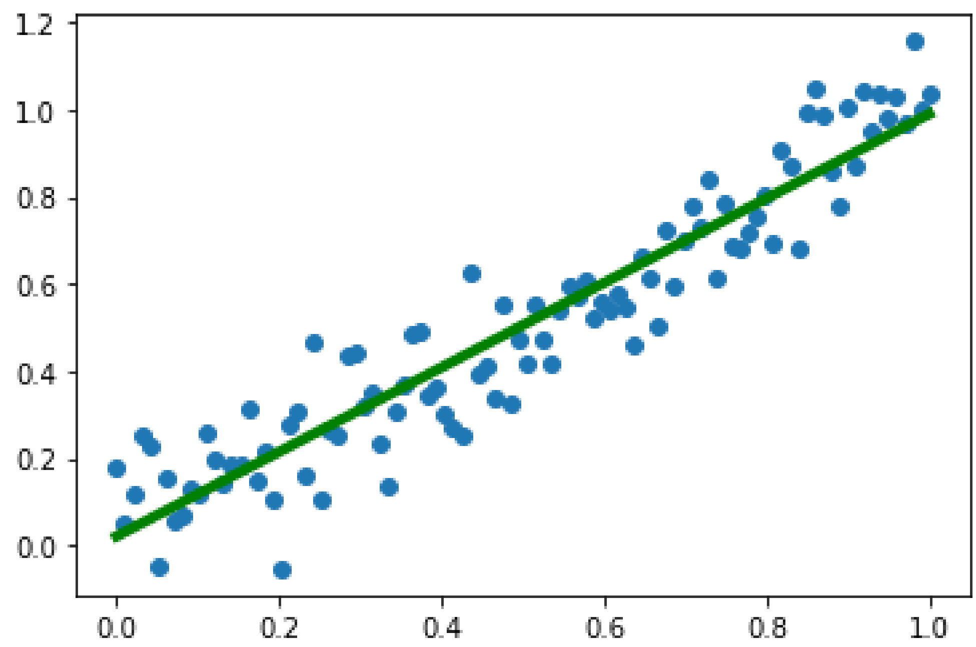
Zaimplementuj funkcję realizującą jedną epokę treningową. Oblicz predykcję przy aktualnych parametrach oraz zaktualizuj je zgodnie z powyższymi wzorami.

```
def optimize(
    x: np.ndarray, y: np.ndarray, a: float, b: float, learning_rate:
float = 0.1
):
    y_hat = a * x + b
    errors = y - y_hat
    N = errors.shape[0]
    new_a = a + learning_rate * -2 / N * np.sum(errors * (-x))
    new_b = b + learning_rate * -2 / N * np.sum(errors * (-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:  0.1330225119404028
step 100 loss:  0.012673197778527677
step 200 loss:  0.010257153540857817
step 300 loss:  0.0100948037549359
step 400 loss:  0.010083894412889118
step 500 loss:  0.010083161342973332
step 600 loss:  0.010083112083219709
step 700 loss:  0.010083108773135261
step 800 loss:  0.010083108550709076
step 900 loss:  0.01008310853576281
final loss: 0.010083108534760455
```

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

```
[<matplotlib.lines.Line2D at 0x2336d465c70>]
```



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([1.0788, 1.2032, 1.6649, 1.2569, 1.2888, 1.8319, 1.2636,
1.1608, 1.9806,
        1.0484])
tensor([0.0788, 0.2032, 0.6649, 0.2569, 0.2888, 0.8319, 0.2636,
0.1608, 0.9806,
        0.0484])
tensor(3.7779)
```

```
# 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([0.8895], requires_grad=True), tensor([0.0072],
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 czątkowych).

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

```
tensor(0.0136, 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([-0.0678])
```

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([-0.1355])
```

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(0.0136, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 100 loss:  tensor(0.0101, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 200 loss:  tensor(0.0101, 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")
```

```python
# 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: 0.0792
step 100 loss: 0.0146
step 200 loss: 0.0104
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.

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

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:--
--:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:--
--:--:--     0
  2 3881k    2 95232    0     0  65738      0  0:01:00  0:00:01
0:00:59 65722
 31 3881k   31 1240k    0     0   541k      0  0:00:07  0:00:02
0:00:05  541k
 92 3881k   92 3580k    0     0   959k      0  0:00:04  0:00:03
0:00:01  959k
 92 3881k   92 3596k    0     0   821k      0  0:00:04  0:00:04
--:--:--  821k
100 3881k  100 3881k    0     0   743k      0  0:00:05  0:00:05
--:--:--  808k
```

```
import pandas as pd

columns = [
    "age",
    "workclass",
    "fnlwgt",
    "education",
    "education-num",
    "marital-status",
    "occupation",
    "relationship",
    "race",
    "sex",
    "capital-gain",
```

```python
    "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)
```

```python
# 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
```

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

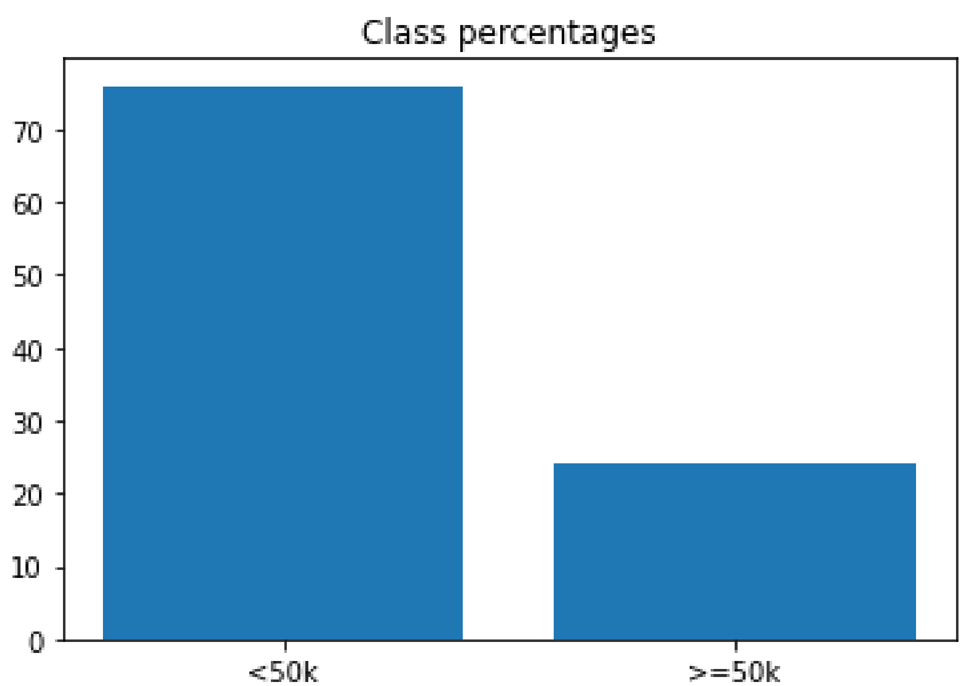
Uwaga co do typów - PyTorchu wszystko w sieci neuronowej musi być typu `float32`. W szczególności trzeba uważać na konwersje z Numpy'a, który używa domyślnie typu `float64`. Może ci się przydać metoda `.float()`.

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

# fill args below
model = nn.Linear(X_train.shape[1], 1)
activation = nn.Sigmoid()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
loss_fn = nn.BCELoss()
#implement me!
for i in range(3000):
    y_hat = activation(model(X_train))
    loss = loss_fn(y_hat, y_train)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

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

```
final loss: 0.4375
```

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():`.

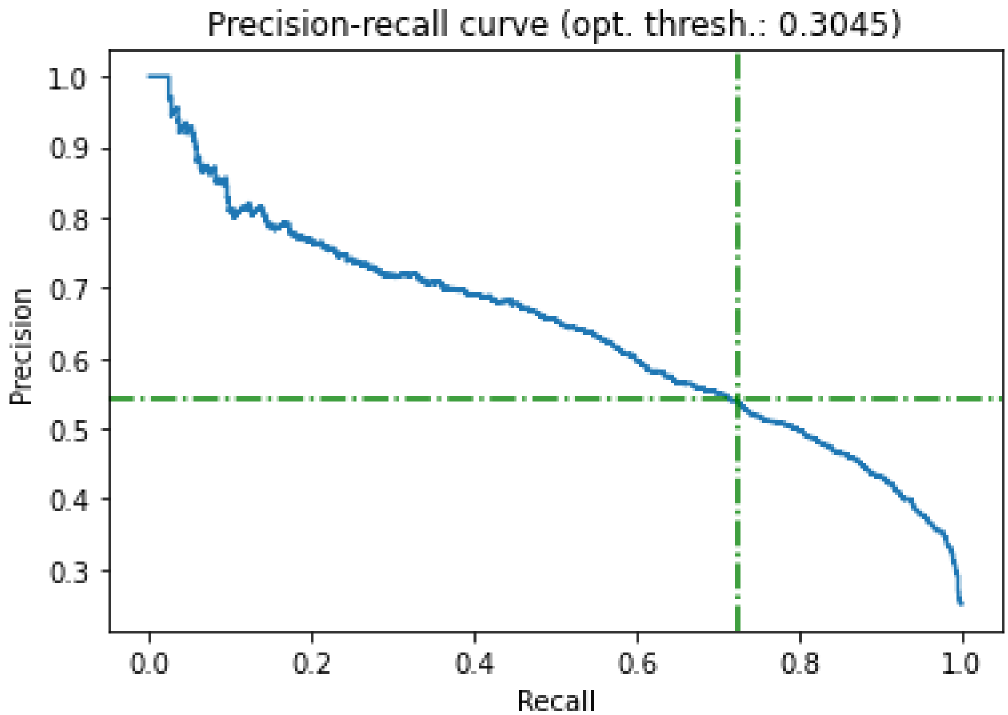
```
from sklearn.metrics import precision_recall_curve,
precision_recall_fscore_support, roc_auc_score

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: 85.11%
```

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.

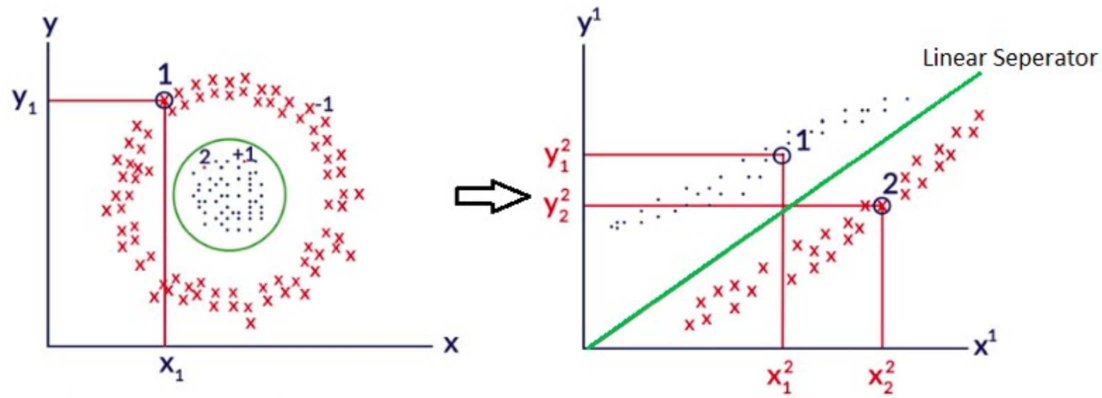
Dodatkowo, pomimo prostoty MLP są bardzo potężne - udowodniono, że perceptrony (ich powszechna nazwa) są uniwersalnym aproksymatorem, będącym w stanie przybliżyć dowolną funkcję z odpowiednio małym błędem, zakładając wystarczającą wielkość warstw sieci. Szczególne ich wersje potrafią nawet reprezentować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

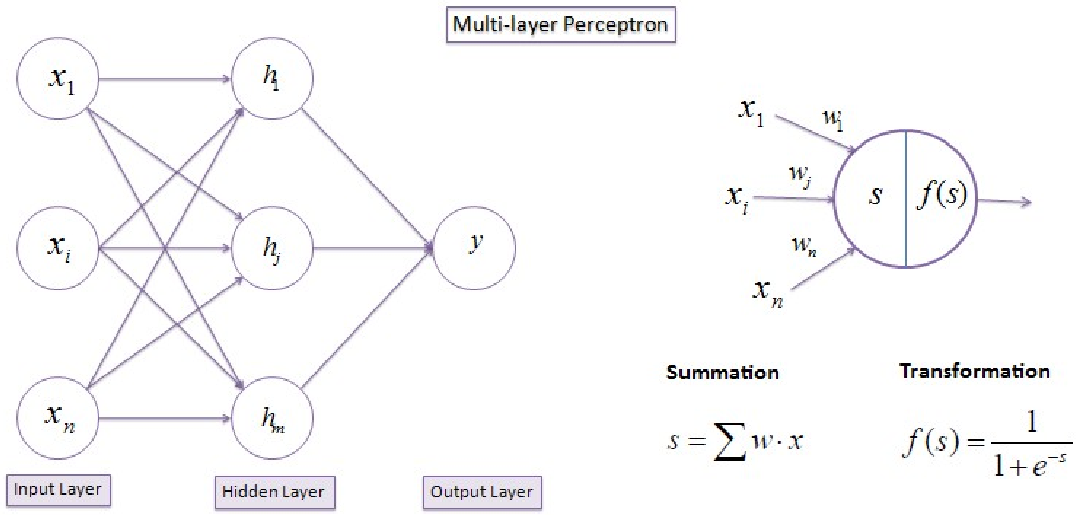
Sieci MLP

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.

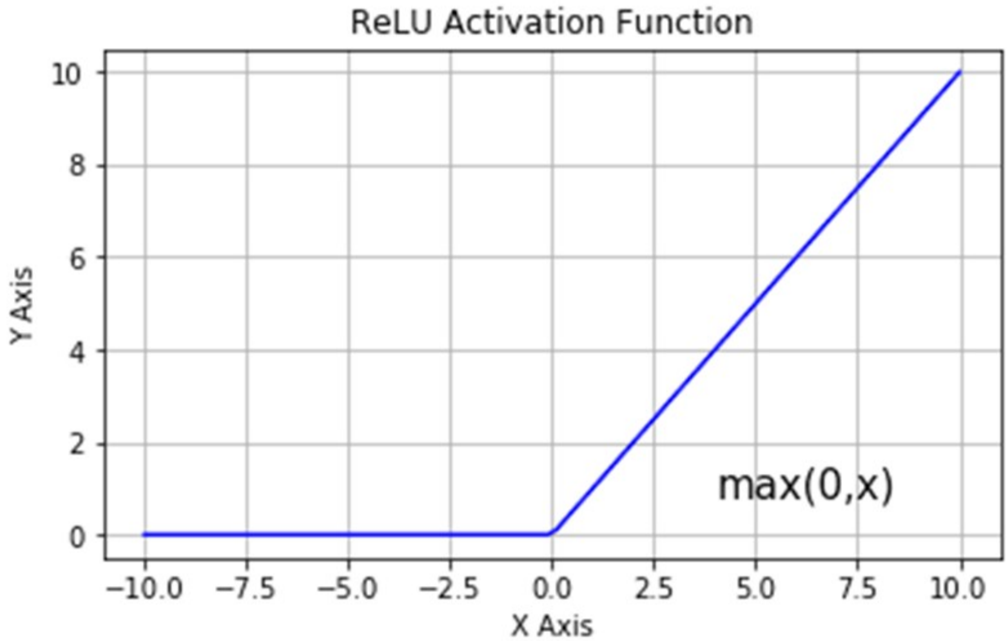


Zapisane matematycznie MLP to: $ h_1 = f_1(x) \ h_2 = f_2(h_1) \ h_3 = f_3(h_2) \ ... h_n = f_n(h_{n-1}) $ gdzie x to wejście f_i to funkcja aktywacji i-tej warstwy, a h_i to wyjście i-tej warstwy, nazywane **ukrytą reprezentacją (hidden representation)**, lub *latent representation*. Nazwa bierze się 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`

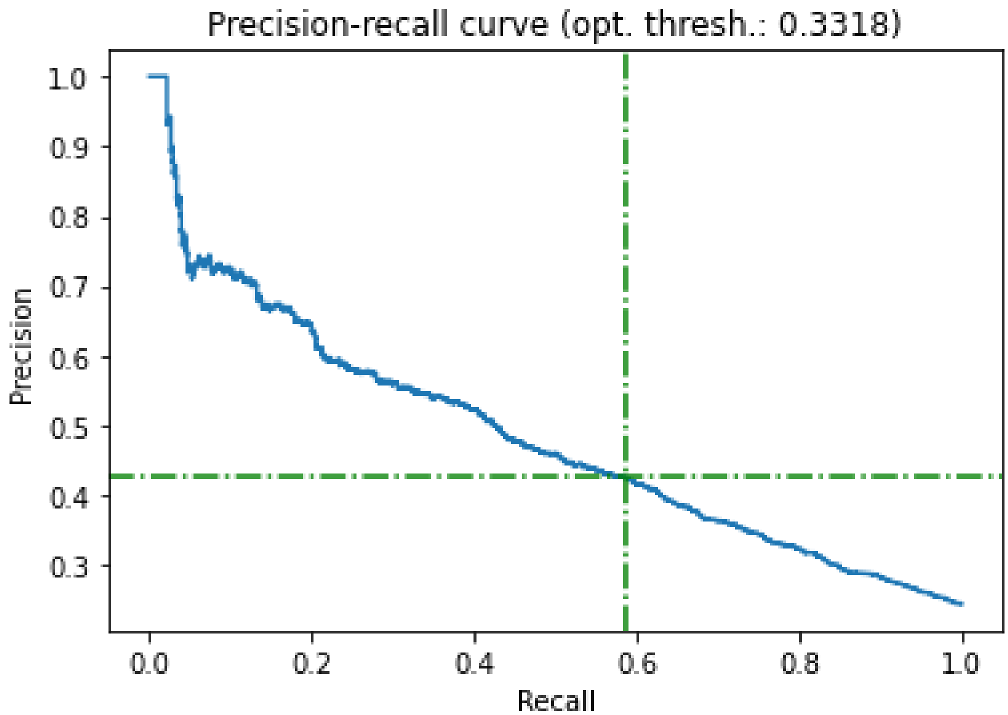
```
from torch import sigmoid

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

        # implement me!
#         raise NotImplementedError
        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):
        # implement me!
#         raise NotImplementedError
        return self.mlp(x)

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

    def predict(self, x):
```

```python
        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: 0.6827
Epoch 200 train loss: 0.6620
Epoch 400 train loss: 0.6441
Epoch 600 train loss: 0.6283
Epoch 800 train loss: 0.6143
Epoch 1000 train loss: 0.6019
Epoch 1200 train loss: 0.5908
Epoch 1400 train loss: 0.5808
Epoch 1600 train loss: 0.5719
Epoch 1800 train loss: 0.5639
final loss: 0.5566
```

```python
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: 69.32%
```



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()
#     raise NotImplementedError
    with torch.no_grad():
        y_hat = model(X)

    auroc = roc_auc_score(y, y_hat)

    if threshold is None:
        precisions, recalls, thresholds = precision_recall_curve(y, 
y_hat)
```

```
        _, threshold = get_optimal_threshold(precisions, recalls, 
thresholds)

    y_pred = np.array(threshold < y_hat, float)

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

    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!
#         raise NotImplementedError
        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:
        # model training
        # implement me!
```

```
#         raise NotImplementedError
        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:
#         raise NotImplementedError
        best_model = deepcopy(model)
        best_val_loss = valid_metrics['loss']
        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: 0.6753, eval loss 0.6730343103408813
Epoch 1 train loss: 0.6531, eval loss 0.6564854979515076
Epoch 2 train loss: 0.6489, eval loss 0.6420695781707764
Epoch 3 train loss: 0.6299, eval loss 0.6293230652809143
Epoch 4 train loss: 0.6204, eval loss 0.6179828643798828
Epoch 5 train loss: 0.6048, eval loss 0.6078071594238281
Epoch 6 train loss: 0.5932, eval loss 0.5986267924308777
Epoch 7 train loss: 0.5893, eval loss 0.5902979969978333
Epoch 8 train loss: 0.5739, eval loss 0.5826938152313232
Epoch 9 train loss: 0.5678, eval loss 0.5757415890693665
Epoch 10 train loss: 0.5701, eval loss 0.569320559501648
Epoch 11 train loss: 0.5630, eval loss 0.5634186863899231
Epoch 12 train loss: 0.5467, eval loss 0.5579308867454529
Epoch 13 train loss: 0.5428, eval loss 0.5528228282928467
Epoch 14 train loss: 0.5513, eval loss 0.5480796694755554
Epoch 15 train loss: 0.5340, eval loss 0.5436269640922546
Epoch 16 train loss: 0.5366, eval loss 0.5394163131713867
Epoch 17 train loss: 0.5299, eval loss 0.5354220271110535
Epoch 18 train loss: 0.5294, eval loss 0.5316218137741089
Epoch 19 train loss: 0.5205, eval loss 0.5279554128646851
Epoch 20 train loss: 0.5140, eval loss 0.5244095921516418
Epoch 21 train loss: 0.5284, eval loss 0.5209852457046509
Epoch 22 train loss: 0.5090, eval loss 0.5176106691360474
Epoch 23 train loss: 0.5112, eval loss 0.514309823513031
```

```
Epoch 24 train loss: 0.5041, eval loss 0.5110863447189331
Epoch 25 train loss: 0.5040, eval loss 0.5079024434089661
Epoch 26 train loss: 0.4915, eval loss 0.5047450661659241
Epoch 27 train loss: 0.4910, eval loss 0.5015979409217834
Epoch 28 train loss: 0.5006, eval loss 0.4984790086746216
Epoch 29 train loss: 0.4829, eval loss 0.4953109920024872
Epoch 30 train loss: 0.4778, eval loss 0.4921754002571106
Epoch 31 train loss: 0.4964, eval loss 0.48901572823524475
Epoch 32 train loss: 0.4939, eval loss 0.48584771156311035
Epoch 33 train loss: 0.4792, eval loss 0.48270079493522644
Epoch 34 train loss: 0.4796, eval loss 0.4795263111591339
Epoch 35 train loss: 0.4814, eval loss 0.4763558804988861
Epoch 36 train loss: 0.4648, eval loss 0.4731592833995819
Epoch 37 train loss: 0.4608, eval loss 0.4699855148792267
Epoch 38 train loss: 0.4703, eval loss 0.46683788299560547
Epoch 39 train loss: 0.4660, eval loss 0.4636763334274292
Epoch 40 train loss: 0.4857, eval loss 0.4605135917663574
Epoch 41 train loss: 0.4641, eval loss 0.45739755034446716
Epoch 42 train loss: 0.4562, eval loss 0.45429012179374695
Epoch 43 train loss: 0.4647, eval loss 0.45122042298316956
Epoch 44 train loss: 0.4406, eval loss 0.4481394290924072
Epoch 45 train loss: 0.4497, eval loss 0.44510403275489807
Epoch 46 train loss: 0.4445, eval loss 0.44211867451667786
Epoch 47 train loss: 0.4636, eval loss 0.4391302168369293
Epoch 48 train loss: 0.4474, eval loss 0.43616560101509094
Epoch 49 train loss: 0.4403, eval loss 0.433294415473938
Epoch 50 train loss: 0.4471, eval loss 0.4304647743701935
Epoch 51 train loss: 0.4271, eval loss 0.42768630385398865
Epoch 52 train loss: 0.4425, eval loss 0.4249480366706848
Epoch 53 train loss: 0.4326, eval loss 0.4222424626350403
Epoch 54 train loss: 0.4158, eval loss 0.41957515478134155
Epoch 55 train loss: 0.4319, eval loss 0.4169592559337616
Epoch 56 train loss: 0.4251, eval loss 0.41441985964775085
Epoch 57 train loss: 0.4308, eval loss 0.41192007064819336
Epoch 58 train loss: 0.4021, eval loss 0.409496933221817
Epoch 59 train loss: 0.4325, eval loss 0.4071432948112488
Epoch 60 train loss: 0.4204, eval loss 0.4048580825328827
Epoch 61 train loss: 0.4194, eval loss 0.4026276767253876
Epoch 62 train loss: 0.4299, eval loss 0.4004490077495575
Epoch 63 train loss: 0.4060, eval loss 0.3983266353607178
Epoch 64 train loss: 0.4424, eval loss 0.396285742521286
Epoch 65 train loss: 0.4155, eval loss 0.39429226517677307
Epoch 66 train loss: 0.4265, eval loss 0.3923707604408264
Epoch 67 train loss: 0.3975, eval loss 0.39052730798721313
Epoch 68 train loss: 0.4053, eval loss 0.3887217938899994
Epoch 69 train loss: 0.4211, eval loss 0.38698697090148926
Epoch 70 train loss: 0.4123, eval loss 0.3853330910205841
Epoch 71 train loss: 0.4035, eval loss 0.38372689485549927
Epoch 72 train loss: 0.4280, eval loss 0.3821866512298584
Epoch 73 train loss: 0.4112, eval loss 0.3806842565536499
```

```
Epoch 74 train loss: 0.4091, eval loss 0.37925994396209717
Epoch 75 train loss: 0.4129, eval loss 0.3778876622543335
Epoch 76 train loss: 0.4026, eval loss 0.37657666206359863
Epoch 77 train loss: 0.4044, eval loss 0.37528613209724426
Epoch 78 train loss: 0.3799, eval loss 0.3740912973880768
Epoch 79 train loss: 0.4023, eval loss 0.3729289174079895
Epoch 80 train loss: 0.4202, eval loss 0.3718225955963135
Epoch 81 train loss: 0.4028, eval loss 0.37074992060661316
Epoch 82 train loss: 0.3912, eval loss 0.36975574493408203
Epoch 83 train loss: 0.3842, eval loss 0.36878135800361633
Epoch 84 train loss: 0.3930, eval loss 0.3678527772426605
Epoch 85 train loss: 0.4136, eval loss 0.3669714033603668
Epoch 86 train loss: 0.4037, eval loss 0.3661035895347595
Epoch 87 train loss: 0.3938, eval loss 0.3653072714805603
Epoch 88 train loss: 0.3983, eval loss 0.36453405022621155
Epoch 89 train loss: 0.4148, eval loss 0.363802969455719
Epoch 90 train loss: 0.3995, eval loss 0.36311233043670654
Epoch 91 train loss: 0.3644, eval loss 0.3624410927295685
Epoch 92 train loss: 0.4189, eval loss 0.3617822527885437
Epoch 93 train loss: 0.3930, eval loss 0.3611588478088379
Epoch 94 train loss: 0.3881, eval loss 0.36054301261901855
Epoch 95 train loss: 0.3911, eval loss 0.35997092723846436
Epoch 96 train loss: 0.4046, eval loss 0.35943278670310974
Epoch 97 train loss: 0.4102, eval loss 0.35890132188796997
Epoch 98 train loss: 0.3918, eval loss 0.35840824246406555
Epoch 99 train loss: 0.4070, eval loss 0.3579252064228058
Epoch 100 train loss: 0.3833, eval loss 0.35744374990463257
Epoch 101 train loss: 0.3921, eval loss 0.3569883406162262
Epoch 102 train loss: 0.4050, eval loss 0.3565453588962555
Epoch 103 train loss: 0.3628, eval loss 0.3561065196990967
Epoch 104 train loss: 0.3846, eval loss 0.3556804656982422
Epoch 105 train loss: 0.3661, eval loss 0.355266809463501
Epoch 106 train loss: 0.4069, eval loss 0.35487666726112366
Epoch 107 train loss: 0.3766, eval loss 0.3545050621032715
Epoch 108 train loss: 0.3923, eval loss 0.35413530468940735
Epoch 109 train loss: 0.3755, eval loss 0.3537631928920746
Epoch 110 train loss: 0.3920, eval loss 0.3534010350704193
Epoch 111 train loss: 0.3880, eval loss 0.3530585467815399
Epoch 112 train loss: 0.3602, eval loss 0.3527171313762665
Epoch 113 train loss: 0.4043, eval loss 0.3523823618888855
Epoch 114 train loss: 0.3729, eval loss 0.3520634174346924
Epoch 115 train loss: 0.3861, eval loss 0.35174745321273804
Epoch 116 train loss: 0.3816, eval loss 0.3514111340045929
Epoch 117 train loss: 0.3952, eval loss 0.3511126637458801
Epoch 118 train loss: 0.3935, eval loss 0.3508177697658539
Epoch 119 train loss: 0.3991, eval loss 0.35053586959838867
Epoch 120 train loss: 0.3865, eval loss 0.35024186968803406
Epoch 121 train loss: 0.3993, eval loss 0.34995806217193604
Epoch 122 train loss: 0.3833, eval loss 0.34968316555023193
Epoch 123 train loss: 0.4013, eval loss 0.34940966963768005
```

```
Epoch 124 train loss: 0.3418, eval loss 0.34912344813346863
Epoch 125 train loss: 0.4083, eval loss 0.34885814785957336
Epoch 126 train loss: 0.4020, eval loss 0.34860774874687195
Epoch 127 train loss: 0.3673, eval loss 0.34836769104003906
Epoch 128 train loss: 0.3921, eval loss 0.3481082022190094
Epoch 129 train loss: 0.3577, eval loss 0.34785696864128113
Epoch 130 train loss: 0.4406, eval loss 0.34760770201683044
Epoch 131 train loss: 0.3721, eval loss 0.347372442483902
Epoch 132 train loss: 0.4045, eval loss 0.34713372588157654
Epoch 133 train loss: 0.4185, eval loss 0.34691551327705383
Epoch 134 train loss: 0.3998, eval loss 0.3466799557209015
Epoch 135 train loss: 0.3809, eval loss 0.34643831849098206
Epoch 136 train loss: 0.3858, eval loss 0.34620794653892517
Epoch 137 train loss: 0.3732, eval loss 0.3459852933883667
Epoch 138 train loss: 0.3711, eval loss 0.34575939178466797
Epoch 139 train loss: 0.3743, eval loss 0.34555989503860474

Epoch 140 train loss: 0.3576, eval loss 0.34535539150238037
Epoch 141 train loss: 0.3691, eval loss 0.34513917565345764
Epoch 142 train loss: 0.3886, eval loss 0.3449239432811737
Epoch 143 train loss: 0.3725, eval loss 0.3447071611881256
Epoch 144 train loss: 0.3739, eval loss 0.3444983661174774
Epoch 145 train loss: 0.3619, eval loss 0.3442804515361786
Epoch 146 train loss: 0.3738, eval loss 0.34406036138534546
Epoch 147 train loss: 0.3907, eval loss 0.34385228157043457
Epoch 148 train loss: 0.4088, eval loss 0.3436600863933563
Epoch 149 train loss: 0.3748, eval loss 0.3434622883796692
Epoch 150 train loss: 0.3713, eval loss 0.3432718813419342
Epoch 151 train loss: 0.3720, eval loss 0.3430734872817993
Epoch 152 train loss: 0.3674, eval loss 0.34287238121032715
Epoch 153 train loss: 0.3907, eval loss 0.342695415019989
Epoch 154 train loss: 0.3362, eval loss 0.3424813449382782
Epoch 155 train loss: 0.3882, eval loss 0.34229764342308044
Epoch 156 train loss: 0.3863, eval loss 0.3421133756637573
Epoch 157 train loss: 0.3789, eval loss 0.3419206738471985
Epoch 158 train loss: 0.3683, eval loss 0.3417299687862396
Epoch 159 train loss: 0.3637, eval loss 0.3415624499320984
Epoch 160 train loss: 0.3604, eval loss 0.3413981795310974
Epoch 161 train loss: 0.3737, eval loss 0.3412129580974579
Epoch 162 train loss: 0.3914, eval loss 0.3410419523715973
Epoch 163 train loss: 0.3550, eval loss 0.34085702896118164
Epoch 164 train loss: 0.3812, eval loss 0.34068965911865234
Epoch 165 train loss: 0.3579, eval loss 0.34049907326698303
Epoch 166 train loss: 0.3926, eval loss 0.34030991792678833
Epoch 167 train loss: 0.3451, eval loss 0.34014183282852173
Epoch 168 train loss: 0.3710, eval loss 0.33997058868408203
Epoch 169 train loss: 0.3684, eval loss 0.33979952335357666
Epoch 170 train loss: 0.3681, eval loss 0.33962562680244446
Epoch 171 train loss: 0.3648, eval loss 0.33946356177330017
Epoch 172 train loss: 0.3794, eval loss 0.339302122592926
```

```
Epoch 173 train loss: 0.3637, eval loss 0.3391442596912384
Epoch 174 train loss: 0.3527, eval loss 0.3389860987663269
Epoch 175 train loss: 0.3904, eval loss 0.33882373571395874
Epoch 176 train loss: 0.3991, eval loss 0.3386586308479309
Epoch 177 train loss: 0.3796, eval loss 0.3385051488876343
Epoch 178 train loss: 0.3809, eval loss 0.3383575677871704
Epoch 179 train loss: 0.3362, eval loss 0.33820608258247375
Epoch 180 train loss: 0.3357, eval loss 0.33805641531944275
Epoch 181 train loss: 0.3549, eval loss 0.3378869295120239
Epoch 182 train loss: 0.4057, eval loss 0.3377372920513153
Epoch 183 train loss: 0.3932, eval loss 0.33758872747421265
Epoch 184 train loss: 0.3877, eval loss 0.3374456465244293
Epoch 185 train loss: 0.3893, eval loss 0.3372938632965088
Epoch 186 train loss: 0.3691, eval loss 0.337139368057251
Epoch 187 train loss: 0.3773, eval loss 0.33698445558547974
Epoch 188 train loss: 0.3612, eval loss 0.3368452191352844
Epoch 189 train loss: 0.3801, eval loss 0.3367058038711548
Epoch 190 train loss: 0.3808, eval loss 0.33656588196754456
Epoch 191 train loss: 0.3590, eval loss 0.3364177644252777
Epoch 192 train loss: 0.3587, eval loss 0.3362772464752197
Epoch 193 train loss: 0.3534, eval loss 0.3361387252807617
Epoch 194 train loss: 0.3909, eval loss 0.33602166175842285
Epoch 195 train loss: 0.3666, eval loss 0.3358691930770874
Epoch 196 train loss: 0.3602, eval loss 0.3357236087322235
Epoch 197 train loss: 0.3859, eval loss 0.3355933725833893
Epoch 198 train loss: 0.3952, eval loss 0.33545252680778503
Epoch 199 train loss: 0.3389, eval loss 0.3353259563446045
Epoch 200 train loss: 0.3546, eval loss 0.3352090120315552
Epoch 201 train loss: 0.3857, eval loss 0.3350929319858551
Epoch 202 train loss: 0.3779, eval loss 0.3349702060222626
Epoch 203 train loss: 0.3740, eval loss 0.3348533809185028
Epoch 204 train loss: 0.3826, eval loss 0.3347330689430237
Epoch 205 train loss: 0.3573, eval loss 0.3346089720726013
Epoch 206 train loss: 0.3220, eval loss 0.3344917893409729
Epoch 207 train loss: 0.3559, eval loss 0.33437639474868774
Epoch 208 train loss: 0.3760, eval loss 0.33424970507621765
Epoch 209 train loss: 0.3958, eval loss 0.3341151475906372
Epoch 210 train loss: 0.3461, eval loss 0.3340105712413788
Epoch 211 train loss: 0.3538, eval loss 0.3338894248008728
Epoch 212 train loss: 0.3752, eval loss 0.3337700664997101
Epoch 213 train loss: 0.3679, eval loss 0.3336498737335205
Epoch 214 train loss: 0.3303, eval loss 0.33352234959602356
Epoch 215 train loss: 0.3365, eval loss 0.33341455459594727
Epoch 216 train loss: 0.3502, eval loss 0.3332969546318054
Epoch 217 train loss: 0.3639, eval loss 0.33319583535194397
Epoch 218 train loss: 0.3498, eval loss 0.3330766558647156
Epoch 219 train loss: 0.3314, eval loss 0.33296018838882446
Epoch 220 train loss: 0.3657, eval loss 0.3328346908092499
Epoch 221 train loss: 0.3396, eval loss 0.3327367305755615
Epoch 222 train loss: 0.3485, eval loss 0.33263128995895386
```

```
Epoch 223 train loss: 0.3758, eval loss 0.33251386880874634
Epoch 224 train loss: 0.3868, eval loss 0.3324088454246521
Epoch 225 train loss: 0.3498, eval loss 0.33230116963386536
Epoch 226 train loss: 0.3884, eval loss 0.33220845460891724
Epoch 227 train loss: 0.3727, eval loss 0.33210381865501404
Epoch 228 train loss: 0.3707, eval loss 0.33201614022254944
Epoch 229 train loss: 0.4086, eval loss 0.33193472027778625
Epoch 230 train loss: 0.3459, eval loss 0.3318181335926056
Epoch 231 train loss: 0.3877, eval loss 0.3317030072212219
Epoch 232 train loss: 0.3795, eval loss 0.33159035444259644
Epoch 233 train loss: 0.3513, eval loss 0.3314867615699768
Epoch 234 train loss: 0.3656, eval loss 0.3314044177532196
Epoch 235 train loss: 0.3727, eval loss 0.3313121497631073
Epoch 236 train loss: 0.3471, eval loss 0.33121341466903687
Epoch 237 train loss: 0.3519, eval loss 0.3311147093772888
Epoch 238 train loss: 0.3726, eval loss 0.33101508021354675
Epoch 239 train loss: 0.3642, eval loss 0.33090919256210327
Epoch 240 train loss: 0.3585, eval loss 0.33081379532814026
Epoch 241 train loss: 0.3902, eval loss 0.3307323753833771
Epoch 242 train loss: 0.3899, eval loss 0.330636590719223
Epoch 243 train loss: 0.3693, eval loss 0.3305324912071228
Epoch 244 train loss: 0.3834, eval loss 0.3304499685764313
Epoch 245 train loss: 0.3905, eval loss 0.3303454518318176
Epoch 246 train loss: 0.3592, eval loss 0.3302602469921112
Epoch 247 train loss: 0.3908, eval loss 0.33018702268600464
Epoch 248 train loss: 0.3763, eval loss 0.33011117577552795
Epoch 249 train loss: 0.3554, eval loss 0.3300016224384308
Epoch 250 train loss: 0.3619, eval loss 0.32992225885391235
Epoch 251 train loss: 0.3596, eval loss 0.32982927560806274
Epoch 252 train loss: 0.3774, eval loss 0.32974275946617126
Epoch 253 train loss: 0.3831, eval loss 0.32964131236076355
Epoch 254 train loss: 0.3545, eval loss 0.3295636475086212
Epoch 255 train loss: 0.3716, eval loss 0.32948732376098633
Epoch 256 train loss: 0.3531, eval loss 0.32940250635147095
Epoch 257 train loss: 0.3469, eval loss 0.329321026802063
Epoch 258 train loss: 0.3530, eval loss 0.3292410969734192
Epoch 259 train loss: 0.3732, eval loss 0.32915276288986206
Epoch 260 train loss: 0.3594, eval loss 0.3290754556655884
Epoch 261 train loss: 0.3653, eval loss 0.32899290323257446
Epoch 262 train loss: 0.3451, eval loss 0.32889875769615173
Epoch 263 train loss: 0.3800, eval loss 0.3288373649120331
Epoch 264 train loss: 0.3808, eval loss 0.32876071333885193
Epoch 265 train loss: 0.3526, eval loss 0.32869890332221985
Epoch 266 train loss: 0.3612, eval loss 0.32862308621406555
Epoch 267 train loss: 0.3557, eval loss 0.32856348156929016
Epoch 268 train loss: 0.3846, eval loss 0.328479528427124
Epoch 269 train loss: 0.3763, eval loss 0.3284088373184204
Epoch 270 train loss: 0.3727, eval loss 0.3283337950706482
Epoch 271 train loss: 0.3563, eval loss 0.3282414674758911
Epoch 272 train loss: 0.3718, eval loss 0.32816606760025024
```

```
Epoch 273 train loss: 0.3567, eval loss 0.3281022608280182
Epoch 274 train loss: 0.3762, eval loss 0.32802867889404297
Epoch 275 train loss: 0.3305, eval loss 0.3279608488082886
Epoch 276 train loss: 0.3751, eval loss 0.3278774917125702
Epoch 277 train loss: 0.3549, eval loss 0.32780441641807556

Epoch 278 train loss: 0.3585, eval loss 0.32773277163505554
Epoch 279 train loss: 0.3509, eval loss 0.3276740610599518
Epoch 280 train loss: 0.3944, eval loss 0.3276080787181854
Epoch 281 train loss: 0.3741, eval loss 0.32752981781959534
Epoch 282 train loss: 0.3987, eval loss 0.327476441860199
Epoch 283 train loss: 0.3611, eval loss 0.3274003267288208
Epoch 284 train loss: 0.3420, eval loss 0.32733920216560364
Epoch 285 train loss: 0.3407, eval loss 0.32725751399993896
Epoch 286 train loss: 0.3538, eval loss 0.3271823823451996
Epoch 287 train loss: 0.3486, eval loss 0.32713034749031067
Epoch 288 train loss: 0.3467, eval loss 0.3270409405231476
Epoch 289 train loss: 0.3773, eval loss 0.32697632908821106
Epoch 290 train loss: 0.3312, eval loss 0.3269132375717163
Epoch 291 train loss: 0.3512, eval loss 0.32685399055480957
Epoch 292 train loss: 0.3605, eval loss 0.3267829418182373
Epoch 293 train loss: 0.3330, eval loss 0.32671862840652466
Epoch 294 train loss: 0.3833, eval loss 0.3266723155975342
Epoch 295 train loss: 0.3897, eval loss 0.3266034424304962
Epoch 296 train loss: 0.3412, eval loss 0.3265434503555298
Epoch 297 train loss: 0.3968, eval loss 0.3264923095703125
Epoch 298 train loss: 0.3416, eval loss 0.3264205753803253
Epoch 299 train loss: 0.3759, eval loss 0.3263565003871918
```

```python
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: 90.22%
F1: 68.49%
Precision: 60.81%
Recall: 78.38%
```

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!
#         raise NotImplementedError
        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):
#         raise NotImplementedError
        return self.mlp(x)

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

    def predict(self, x):
```

```python
        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.numpy()),
    y=y_train.numpy().reshape(-1).tolist()
)

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!
```

```python
    model.eval()
    valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
    if valid_metrics['loss'] < best_val_loss:
#         raise NotImplementedError
        best_model = deepcopy(model)
        best_val_loss = valid_metrics['loss']
        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: 0.5687, eval loss 0.48443281650543213
Epoch 1 train loss: 0.5691, eval loss 0.47575125098228455
Epoch 2 train loss: 0.5487, eval loss 0.4733056426048279
Epoch 3 train loss: 0.5253, eval loss 0.47324442863464355
Epoch 4 train loss: 0.5556, eval loss 0.472064733505249
Epoch 5 train loss: 0.5397, eval loss 0.4752527177333832
Epoch 6 train loss: 0.5670, eval loss 0.4722655415534973
Epoch 7 train loss: 0.5291, eval loss 0.47184860706329346
Epoch 8 train loss: 0.5692, eval loss 0.4715957045555115
Epoch 9 train loss: 0.5434, eval loss 0.47439780831336975
Epoch 10 train loss: 0.4934, eval loss 0.4747065603733063
Epoch 11 train loss: 0.5106, eval loss 0.47646668553352356
```

```python
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: 90.78%
F1: 69.63%
Precision: 61.10%
Recall: 80.93%
```

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?

Odpowiedzi:

1. Głębokość sieci (liczba warstw), liczba węzłów w poszczególnych warstwach, stosowana regularyzacja, learning rate.
2. Regularyzacja L1 może prowadzić do zerowania współczynników w mniej istotnych węzłach. W tym przypadku lepiej korzystać z dropoutu zamiast kompletnie rezygnować z informacji w węźle.
3. Użycie innej metryki może dać różną liczbę epok uczenia. W efekcie modele mogłyby się od siebie różnić.

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}%")
```

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