# 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.

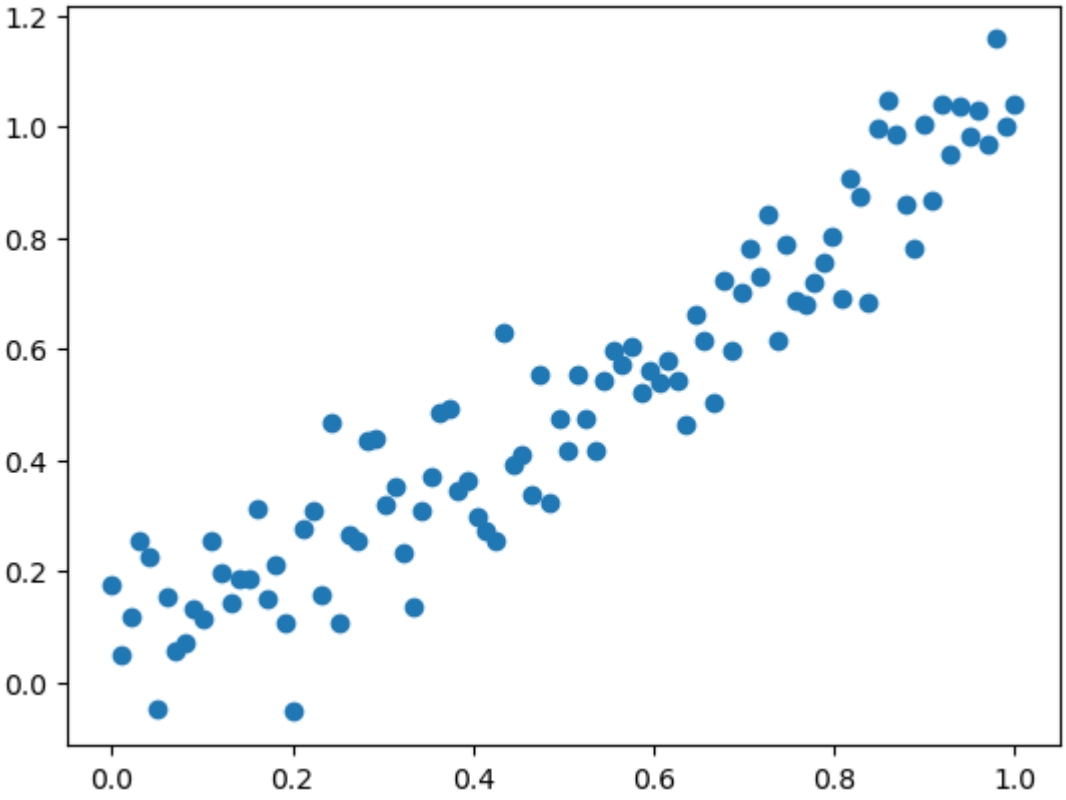
In [ ]:
```
# for conda users
# conda install -y matplotlib pandas pytorch torchvision -c pytorch -c co
```

# Wprowadzenie

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

In [ ]:
```
from typing import Tuple, Dict

import numpy as np
import matplotlib.pyplot as plt
```

In [ ]:
```
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)
plt.show()
```




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)**:

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

Od jakich $\alpha$ i $\beta$ zacząć? W najprostszym wypadku wystarczy po prostu je wylosować jako niewielkie liczby zmiennoprzecinkowe.

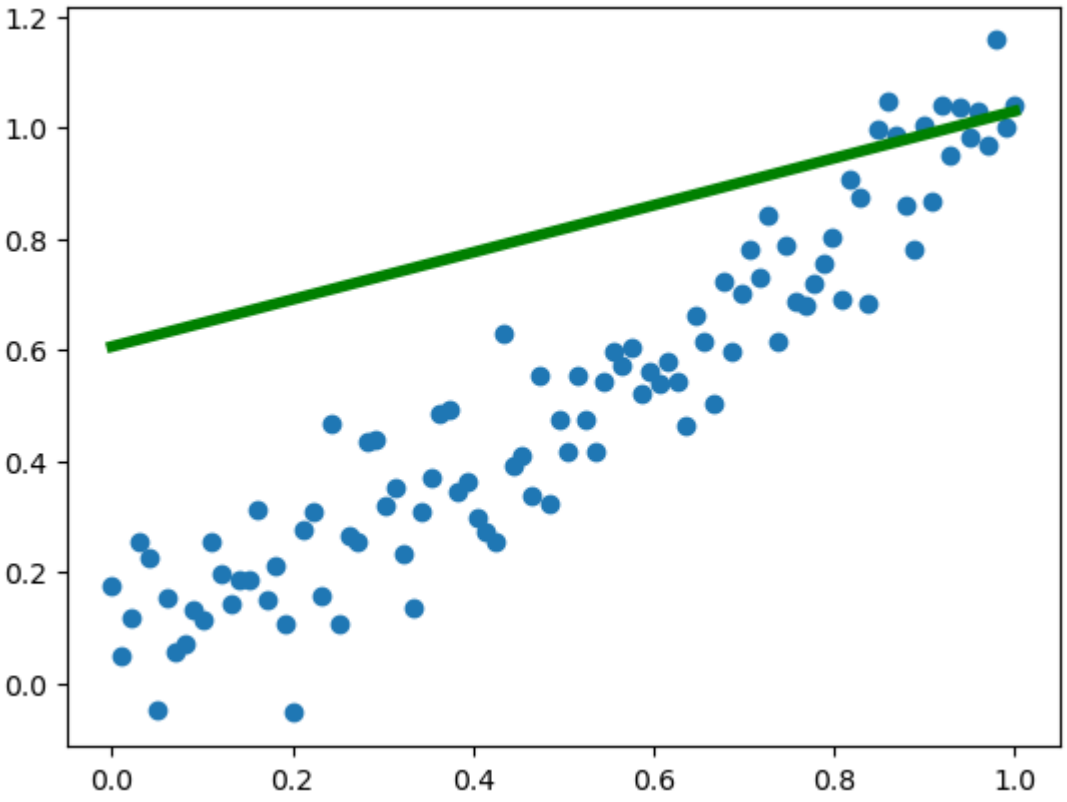
## Zadanie 1 (0.5 punkt)

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

In [ ]:
```
def mse(y: np.ndarray, y_hat: np.ndarray) -> float:
    return np.sum((y - y_hat)**2) / y.shape[0]
```

In [ ]:
```
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)
plt.show()
```

```
MSE: 0.133
```




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 $\epsilon$ można ją przybliżyć jako:

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

Przyglądając się temu równaniu widzimy, że:

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

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, zaś przeciwny zwrot to ten, w którym funkcja najszybciej spada.

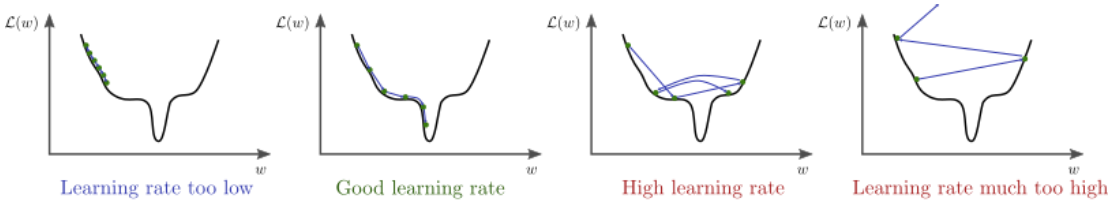
Stosując powyższe do optymalizacji, mamy:

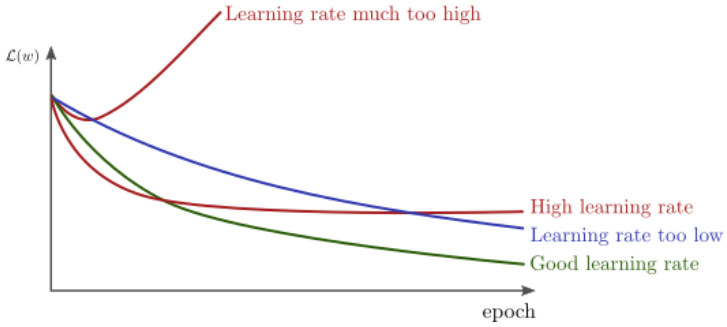
$$x_{t+1} = x_t - \alpha * \frac{f(x)}{dx}$$

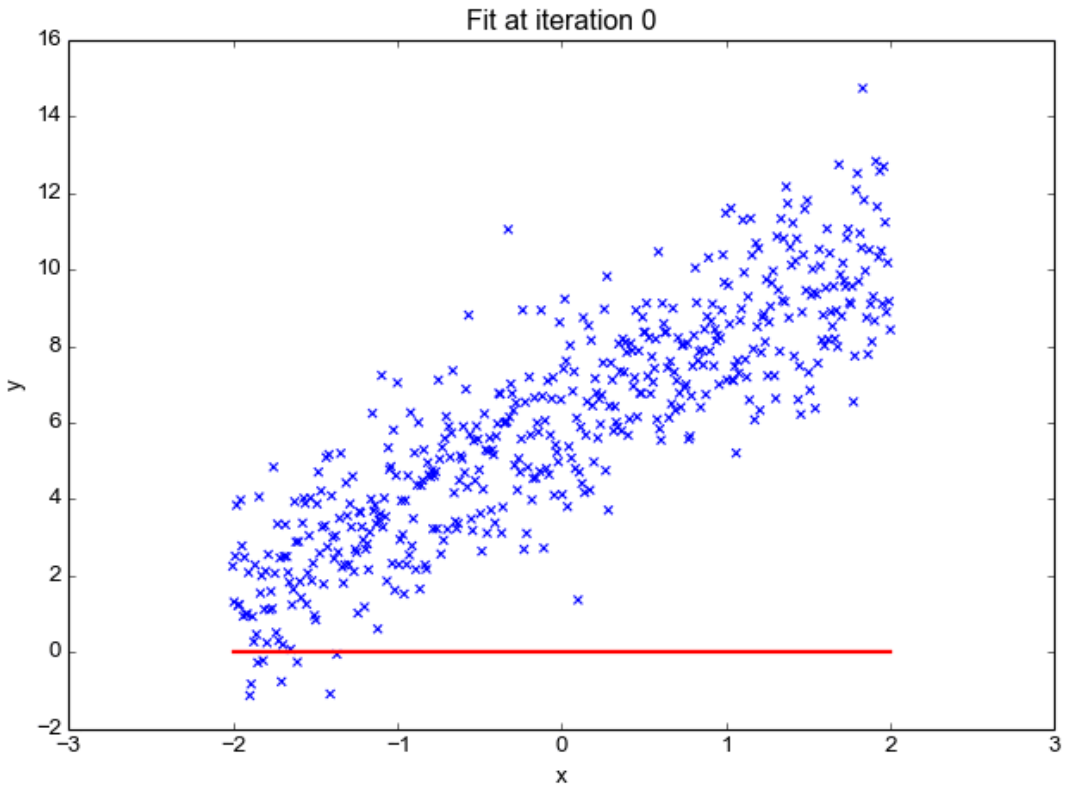
$\alpha$ to niewielka wartość (rzędu zwykle $10^{-5}$ - $10^{-2}$), wprowadzona, aby trzymać się założenia o małej zmianie parametrów ($\epsilon$). Nazywa się ją **stałą uczącą (learning rate)** i jest zwykle najważniejszym hiperparametrem podczas nauki sieci.

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, ale dokładniejszy trening. Jednak nie zawsze ona pozwala osiągnąć lepsze wyniki, bo może okazać się, że utkniemy w minimum lokalnym. Można także zmieniać stałą uczącą podczas treningu, co nazywa się **learning rate scheduling (LR scheduling)**. Obrazowo:




Policzmy więc pochodną dla naszej funkcji kosztu MSE. Pochodną liczymy po parametrach naszego modelu, bo to właśnie ich chcemy dopasować tak, żeby koszt był jak najmniejszy:

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

W powyższym wzorze tylko $y_i$ jest zależny od $a$ oraz $b$. Możemy wykorzystać tu regułę łańcuchową (*chain rule*) i policzyć pochodne po naszych parametrach w sposób następujący:

$$\frac{\mathrm{d}MSE}{\mathrm{d}a} = \frac{1}{N}\sum_{i}^{N}\frac{\mathrm{d}(y_i - \hat{y}_i)^2}{\mathrm{d}\hat{y}_i}\frac{\mathrm{d}\hat{y}_i}{\mathrm{d}a}$$

$$\frac{\mathrm{d}MSE}{\mathrm{d}b} = \frac{1}{N}\sum_{i}^{N}\frac{\mathrm{d}(y_i - \hat{y}_i)^2}{\mathrm{d}\hat{y}_i}\frac{\mathrm{d}\hat{y}_i}{\mathrm{d}b}$$

Policzmy te pochodne po kolei:

$$\frac{\mathrm{d}(y_i - \hat{y}_i)^2}{\mathrm{d}\hat{y}_i} = -2 \cdot (y_i - \hat{y}_i)$$

$$\frac{\mathrm{d}\hat{y}_i}{\mathrm{d}a} = x_i$$

$$\frac{\mathrm{d}\hat{y}_i}{\mathrm{d}b} = 1$$

Łącząc powyższe wyniki dostaniemy:

$$\frac{\mathrm{d}MSE}{\mathrm{d}a} = \frac{-2}{N}\sum_{i}^{N}(y_i - \hat{y}_i) \cdot x_i$$

$$\frac{\mathrm{d}MSE}{\mathrm{d}b} = \frac{-2}{N}\sum_{i}^{N}(y_i - \hat{y}_i)$$

Aktualizacja parametrów wygląda tak:

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

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

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.

## Zadanie 2 (1.0 punkt)

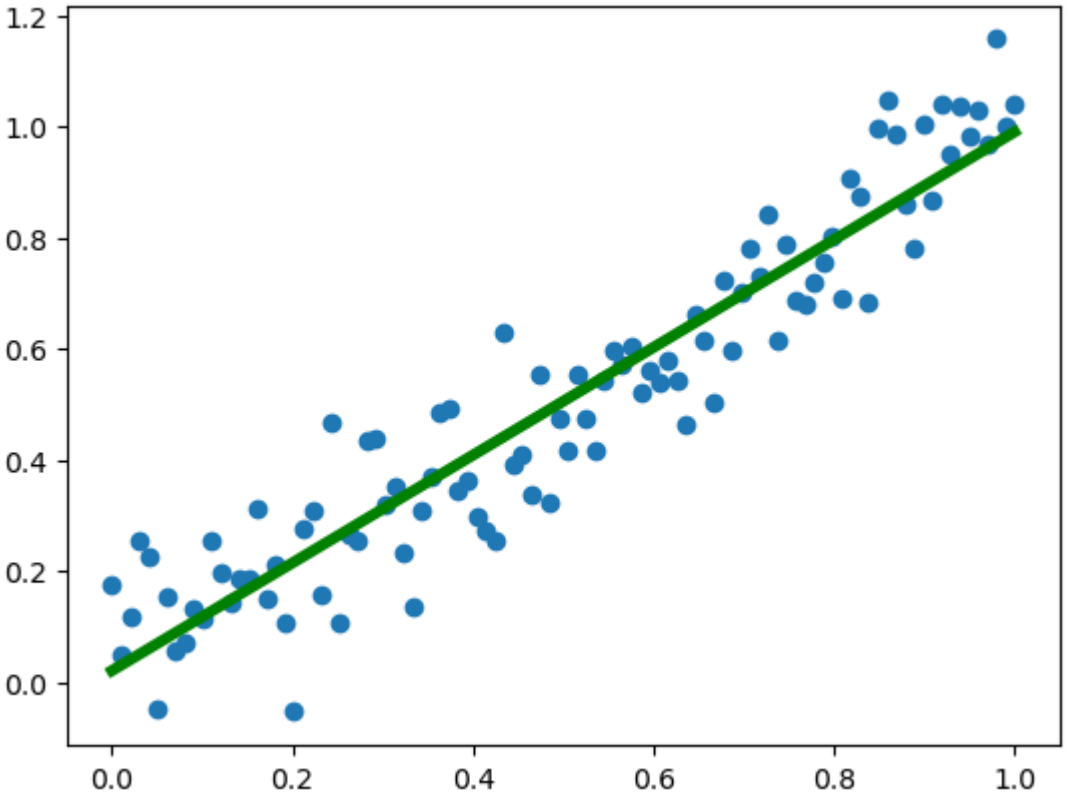
Zaimplementuj funkcję realizującą jedną epokę treningową. Zauważ, że `x` oraz `y` są wektorami. Oblicz predykcję przy aktualnych parametrach oraz zaktualizuj je zgodnie z powyższymi wzorami.

In [ ]:
```python
def optimize(
  x: np.ndarray, y: np.ndarray, a: float, b: float, learning_rate: float
):
  y_hat = a * x + b
  errors = y - y_hat
  n = y.shape[0]

  dfa = -2 / n * np.sum(errors * x)
  dfb = -2 / n * np.sum(errors)

  a -= learning_rate * dfa
  b -= learning_rate * dfb

  return a, b
```

In [ ]:
```python
for epoch in range(1000):
  loss = mse(y, a * x + b)
  a, b = optimize(x, y, a, b)
  if epoch % 100 == 0:
    print(f"Step {epoch} loss: ", loss)

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

```
Step 0 loss:  0.1330225119404028
Step 100 loss:  0.012673197778527677
Step 200 loss:  0.010257153540857817
Step 300 loss:  0.010094803754935901
Step 400 loss:  0.010083894412889118
Step 500 loss:  0.010083161342973334
Step 600 loss:  0.010083112083219712
Step 700 loss:  0.010083108773135263
Step 800 loss:  0.01008310855070908
Step 900 loss:  0.01008310853576281
Final loss: 0.010083108534760455
```

In [ ]:
```python
plt.scatter(x, y)
plt.plot(x, a * x + b, color="g", linewidth=4)
plt.show()
```




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

In [ ]:
```
import torch
import torch.nn as nn
import torch.optim as optim
```

In [ ]:
```
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([1.5810, 1.7808, 1.1891, 1.2316, 1.7162, 1.1231, 1.8544, 1.7425, 1.
7337,
        1.7293])
tensor([0.5810, 0.7808, 0.1891, 0.2316, 0.7162, 0.1231, 0.8544, 0.7425, 0.
7337,
        0.7293])
tensor(5.6819)
```

In [ ]:
```
# 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`.

In [ ]:
```
a = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)
a, b
```

Out[ ]:
```
(tensor([0.6660], requires_grad=True), tensor([0.5258], requires_grad=Tr
ue))
```

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).

In [ ]:
```
mse = nn.MSELoss()
mse(y, a * x + b)
```

Out[ ]:
```
tensor(0.1425, 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.

In [ ]:
```
loss = mse(y, a * x + b)
loss.backward()
```

In [ ]:
```
print(a.grad)
```

```
tensor([0.3011])
```

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.

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

Out[ ]:
```
tensor([0.6022])
```

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

In [ ]:
```
learning_rate = 0.1
for epoch 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 epoch % 100 == 0:
        print(f"step {epoch} loss: ", loss)

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

```
step 0 loss:  tensor(0.1425, dtype=torch.float64, grad_fn=<MseLossBackward
0>)
step 100 loss:  tensor(0.0112, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 200 loss:  tensor(0.0102, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 300 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 400 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 500 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 600 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 700 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 800 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 900 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
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.

In [ ]:
```
# 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 epoch 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 epoch % 100 == 0:
    if loss < best_loss:
      best_model = (a.clone(), b.clone())
      best_loss  = loss
    print(f"Step {epoch} loss: {loss.item():.4f}")

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

```
Step 0 loss: 0.0678
Step 100 loss: 0.0139
Step 200 loss: 0.0103
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 rocznie, 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.

In [ ]:
```
!wget https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adu
```

```
--2023-11-20 22:16:09--  https://archive.ics.uci.edu/ml/machine-learning-d
atabases/adult/adult.data
Resolving archive.ics.uci.edu (archive.ics.uci.edu)... 128.195.10.252
Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|128.195.10.252|:44
3... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified
Saving to: ‘adult.data’

adult.data              [         <=>         ]   3,79M   235KB/s    in 38s

2023-11-20 22:16:47 (103 KB/s) - ‘adult.data’ saved [3974305]
```

In [ ]:
```
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-go
fnlwgt: continuous.
education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acd
education-num: continuous.
marital-status: Married-civ-spouse, Divorced, Never-married, Separated, W
occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-manage
relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Un
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, Ge
"""

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

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

In [ ]:
```
# attribution: https://www.kaggle.com/code/royshih23/topic7-classificatio
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=Tru
df['marital-status'].replace(['Married-civ-spouse'], 'Married',inplace=Tr
df['marital-status'].replace(['Married-spouse-absent'], 'NotMarried',inpl
df['marital-status'].replace(['Separated'], 'Separated',inplace=True)
df['marital-status'].replace(['Divorced'], 'Separated',inplace=True)
df['marital-status'].replace(['Widowed'], 'Widowed',inplace=True)
```

In [ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder, StandardSc


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', 'cap
continuous_X_train = X_train[continuous_cols]
categorical_X_train = X_train.loc[:, ~X_train.columns.isin(continuous_col

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

continuous_X_test = X_test[continuous_cols]
```

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

categorical_encoder = OneHotEncoder(sparse_output=False, handle_unknown='
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=
X_valid = np.concatenate([continuous_X_valid, categorical_X_valid], axis=
X_test = np.concatenate([continuous_X_test, categorical_X_test], axis=1)

X_train.shape, y_train.shape
```

Out[ ]:
```
((20838, 108), (20838,))
```

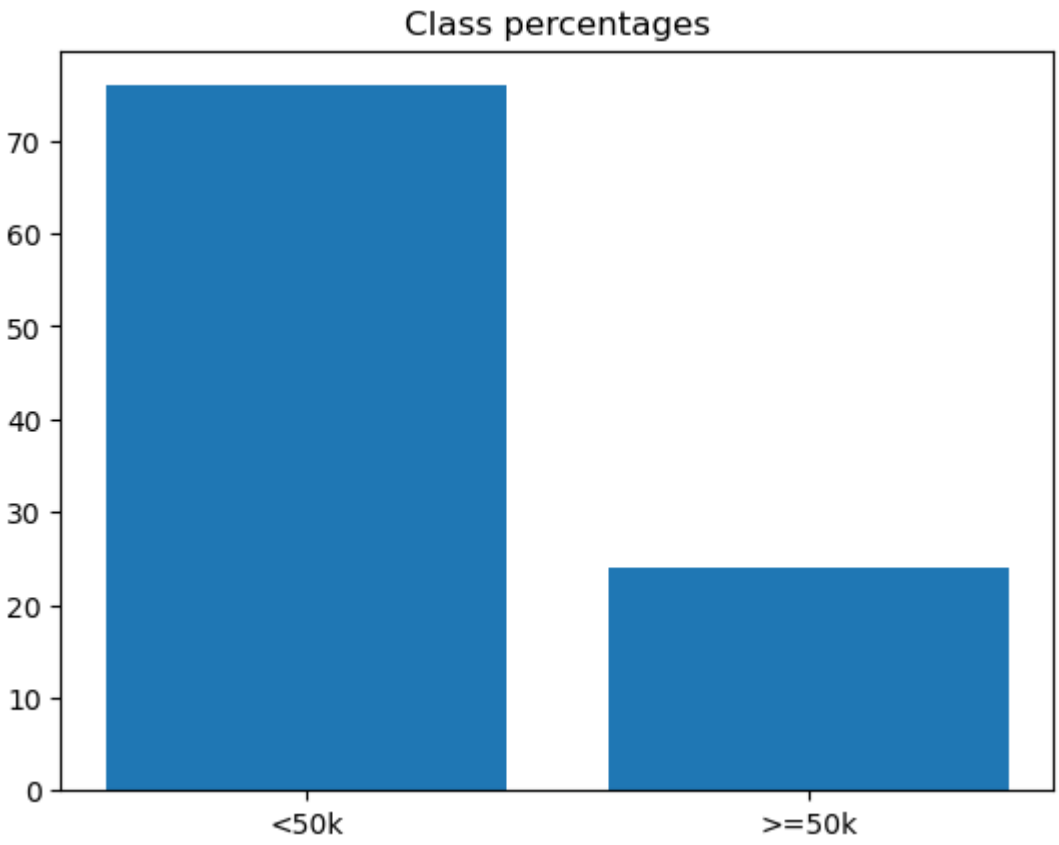
Uwaga co do typów - PyTorchu wszystko w sieci neuronowej musi być typu `float32`. W szczególności trzeba uważać na konwersje z Numpy'a, który używa domyślnie typu `float64`. Może ci się przydać metoda `.float()`.

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

In [ ]:
```
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:

In [ ]:
```
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 (1.0 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()`). Dopisz logowanie kosztu raz na 100 epok.

In [ ]:
```
learning_rate = 1e-3

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

EPOCH_NUMBER = 3000
for epoch in range(EPOCH_NUMBER):
```

```
  y_pred = model(X_train)
  y_actv = activation(y_pred)

  loss = loss_fn(y_actv, y_train)
  loss.backward()

  optimizer.step()
  optimizer.zero_grad()

  if epoch % 100 == 0:
    if loss < best_loss:
      best_model_params = [param.clone() for param in model.parameters()]
      best_loss         = loss
    print(f"Epoch {epoch} loss: {loss.item():.4f}")

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

```
Epoch 0 loss: 0.6886
Epoch 100 loss: 0.6590
Epoch 200 loss: 0.6341
Epoch 300 loss: 0.6128
Epoch 400 loss: 0.5946
Epoch 500 loss: 0.5788
Epoch 600 loss: 0.5650
Epoch 700 loss: 0.5528
Epoch 800 loss: 0.5421
Epoch 900 loss: 0.5325
Epoch 1000 loss: 0.5238
Epoch 1100 loss: 0.5160
Epoch 1200 loss: 0.5088
Epoch 1300 loss: 0.5022
Epoch 1400 loss: 0.4962
Epoch 1500 loss: 0.4906
Epoch 1600 loss: 0.4854
Epoch 1700 loss: 0.4806
Epoch 1800 loss: 0.4760
Epoch 1900 loss: 0.4718
Epoch 2000 loss: 0.4678
Epoch 2100 loss: 0.4640
Epoch 2200 loss: 0.4604
Epoch 2300 loss: 0.4570
Epoch 2400 loss: 0.4538
Epoch 2500 loss: 0.4508
Epoch 2600 loss: 0.4479
Epoch 2700 loss: 0.4451
Epoch 2800 loss: 0.4425
Epoch 2900 loss: 0.4399
Final loss: tensor(0.4375, grad_fn=<BinaryCrossEntropyBackward0>)
```

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():`.

In [ ]:
```
from sklearn.metrics import precision_recall_curve, precision_recall_fsco

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: 86.02%
```

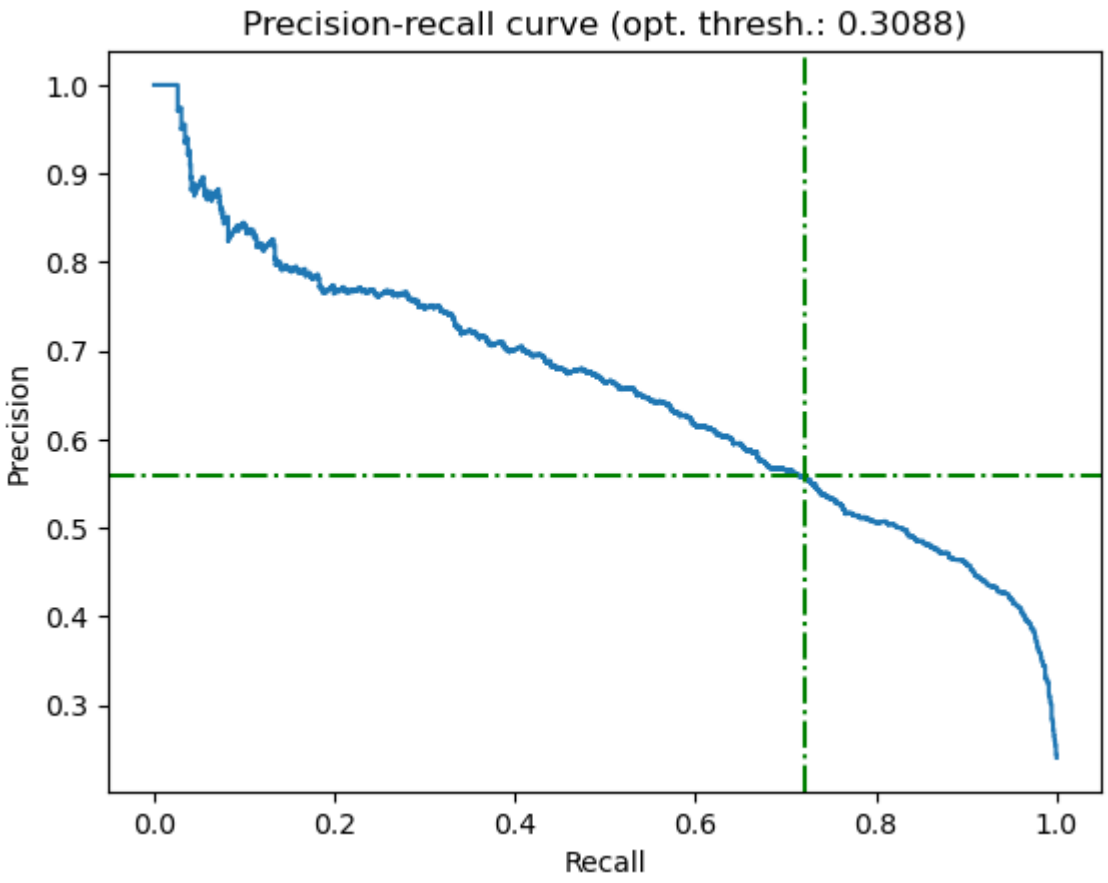
Jest to całkiem dobry wynik, a może być jeszcze lepszy. Sprawdźmy dla pewności jeszcze inne metryki: precyzję, recall oraz F1-score. Dodatkowo narysujemy krzywą precision-recall, czyli jak zmieniają się te metryki w zależności od przyjętego progu (threshold) prawdopodobieństwa, powyżej którego przyjmujemy 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.

In [ ]:
```
from sklearn.metrics import PrecisionRecallDisplay


def get_optimal_threshold(
  precisions: np.array,
  recalls: np.array,
  thresholds: np.array
) -> Tuple[int, float]:

  numerator = 2 * precisions * recalls
  denominator = precisions + recalls
  f1_scores = np.divide(numerator, denominator, out=np.zeros_like(numerat

  optimal_idx = np.argmax(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
  optimal_idx, optimal_threshold = get_optimal_threshold(precisions, reca

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

In [ ]:
```
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 $\Theta$.

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.

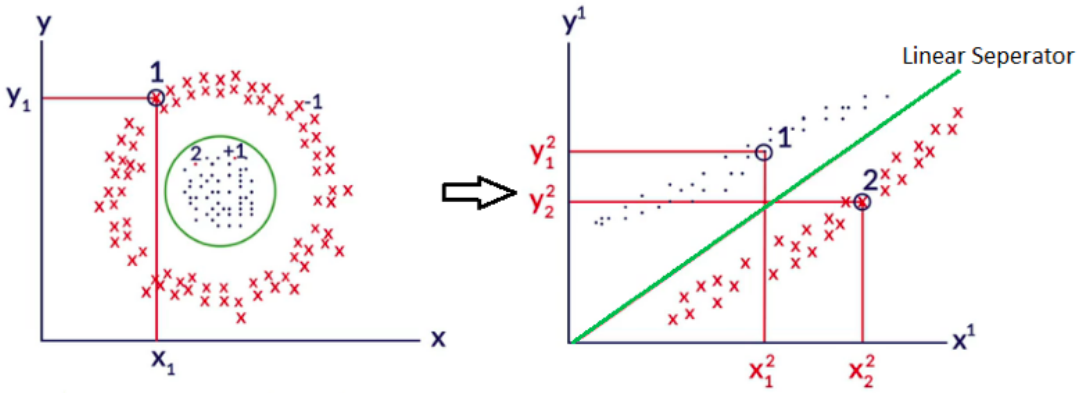
Dodatkowo, pomimo prostoty MLP są bardzo potężne - udowodniono, że perceptrony (ich powszechna nazwa) są uniwersalnym aproksymatorem, będącym w stanie przybliżyć dowolną funkcję z odpowiednio małym błędem, zakładając wystarczającą wielkość warstw sieci. Szczególne ich wersje potrafią nawet reprezentować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

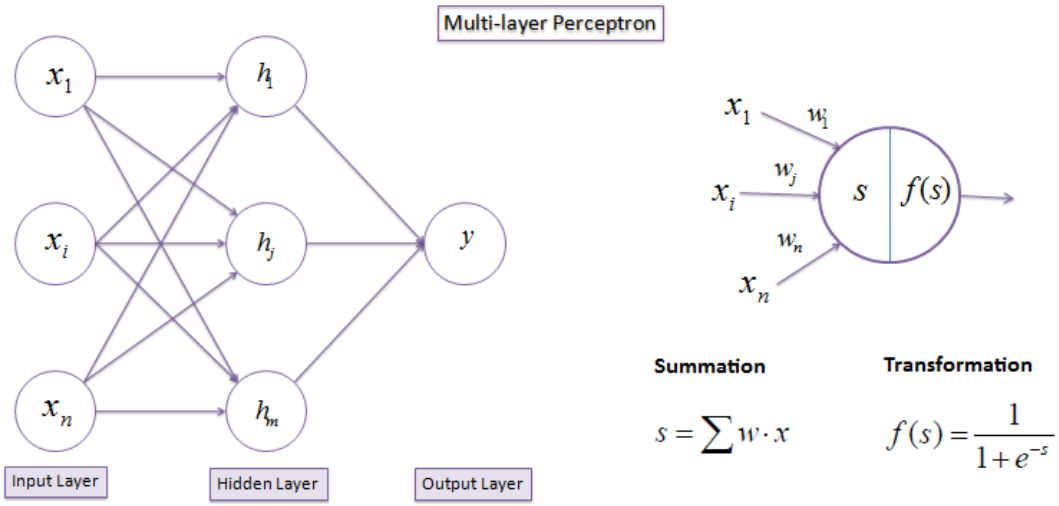
## Sieci MLP

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:

$$
\begin{aligned}
h_1 &= f_1(x) \\
h_2 &= f_2(h_1) \\
h_3 &= f_3(h_2) \\
&\dots \\
h_n &= f_n(h_{n-1})
\end{aligned}
$$

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 uczymy się na danych $x$ o jednym wymiarze (dla uproszczenia wzorów) oraz nie mamy funkcji aktywacji, czyli wykorzystujemy tak naprawdę aktywację liniową $f(x) = x$. Zobaczmy jak będą wyglądać dane przechodząc przez kolejne warstwy:

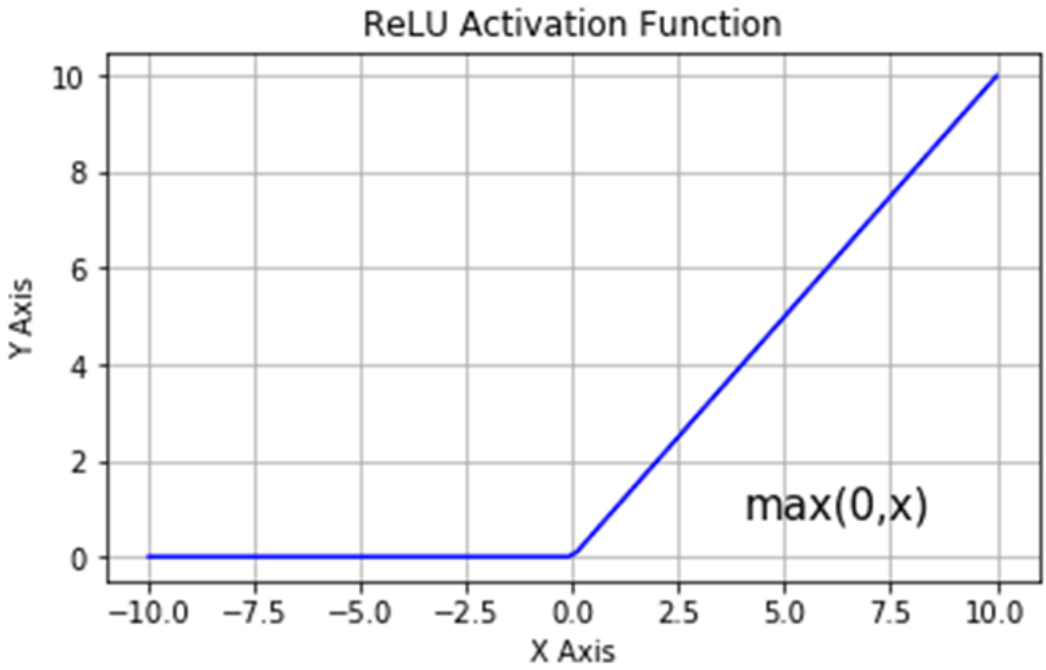
$$
\begin{aligned}
h_1 &= f_1(xw_1) = xw_1 \\
h_2 &= f_2(h_1w_2) = xw_1w_2 \\
&\dots \\
h_n &= f_n(h_{n-1}w_n) = xw_1w_2 \dots w_n
\end{aligned}
$$

gdzie $w_i$ to jest parametr $i$-tej warstwy sieci, $x$ to są dane (w naszym przypadku jedna liczba) wejściowa, a $h_i$ to wyjście $i$-tej warstwy.

Jak widać, taka sieć o $n$ warstwach jest równoważna sieci o jednej warstwie z parametrem $w = w_1 w_2 \dots w_n$. 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 $\sigma$, 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. Każdy atrybut dziedziczący po `nn.Module` lub `nn.Parameter` jest uważany za taki, który zawiera parametry sieci, a więc przy wywołaniu metody `parameters()` - parametry z tych atrybutów pojawią się w liście wszystkich parametrów. 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 (0.5 punktu)

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`

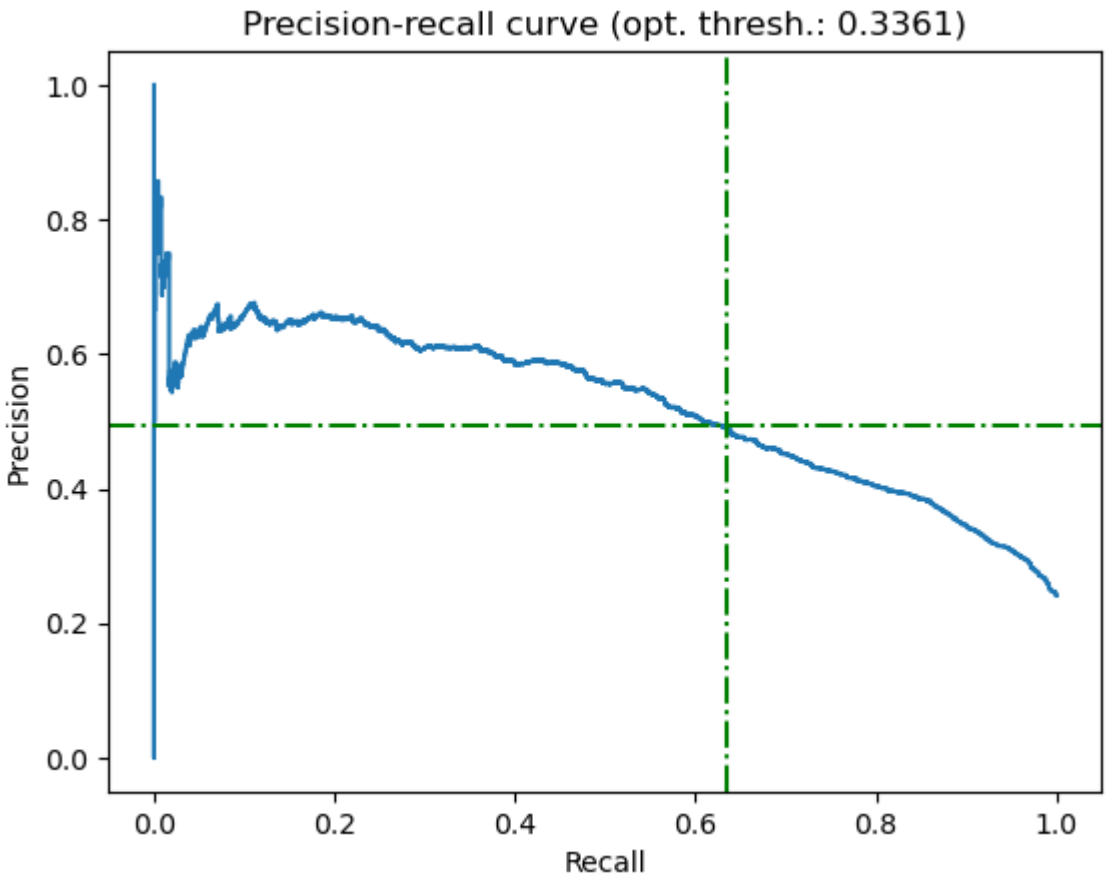
In [ ]:
```
from torch import sigmoid


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

    self.layers = 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.layers.forward(x)

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

  def predict(self, x, threshold: float = 0.5):
    y_pred_score = self.predict_proba(x)
    return (y_pred_score > threshold).to(torch.int32)
```

In [ ]:
```
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 epoch in range(num_epochs):
    y_pred = model(X_train)
    loss = loss_fn(y_pred, y_train)
    loss.backward()

    optimizer.step()
    optimizer.zero_grad()

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

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

```
Epoch 0 train loss: 0.6806
Epoch 200 train loss: 0.6615
Epoch 400 train loss: 0.6446
Epoch 600 train loss: 0.6295
Epoch 800 train loss: 0.6160
Epoch 1000 train loss: 0.6037
Epoch 1200 train loss: 0.5927
Epoch 1400 train loss: 0.5826
Epoch 1600 train loss: 0.5735
Epoch 1800 train loss: 0.5652
Final loss: 0.5577
```

In [ ]:
```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: 79.53%
```




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 (1.5 punktu)

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.

In [ ]:
```python
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]:

  model.eval()
  with torch.no_grad():
    y_pred   = model.predict_proba(X)
    loss_val = loss_fn(y_pred, y)

  auroc = roc_auc_score(y, y_pred)

  if not threshold:
    precisions, recalls, thresholds = precision_recall_curve(y, y_pred)
    _, optimal_threshold = get_optimal_threshold(precisions, recalls, thr
    threshold = optimal_threshold

  y_pred = torch.where(y_pred > threshold, 1, 0)

  fone_score = f1_score(y, y_pred)
  prec_score = precision_score(y, y_pred)
  recl_score = recall_score(y, y_pred)

  return {
    "loss_value":      loss_val,
    "auroc":           auroc,
    "threshold":       threshold,
    "f1_score":        fone_score,
    "precision_score": prec_score,
    "recall_score":    recl_score
  }
```

## Zadanie 6 (0.5 punktu)

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

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

    self.layers = nn.Sequential(
      nn.Linear(input_size, 256),
      nn.ReLU(),
      nn.Dropout(),
      nn.Linear(256, 128),
```

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

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

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

  def predict(self, x, threshold: float = 0.5):
    y_pred_score = self.predict_proba(x)
    return (y_pred_score > threshold).to(torch.int32)
```

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

In [ ]:
```
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 (1.5 punktu)

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.

In [ ]:
```
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
```

In [ ]:
```
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()

  valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
  if valid_metrics["loss_value"] < best_val_loss:
    best_model               = deepcopy(model)
    best_loss                = valid_metrics["loss_value"]
    best_threshold           = valid_metrics["threshold"]
    steps_without_improvement = 0
  else:
    steps_without_improvement += 1
    if steps_without_improvement >= early_stopping_patience:
      print("Early stopping.")
      break

  print(f"Epoch {epoch_num} train loss: {loss.item():.4f}, eval loss {val
```

```
Epoch 0 train loss: 0.6419, eval loss 0.8359695672988892
Epoch 1 train loss: 0.6192, eval loss 0.8307075500488281
Epoch 2 train loss: 0.6192, eval loss 0.8257341980934143
Epoch 3 train loss: 0.6100, eval loss 0.8210088610649109
Epoch 4 train loss: 0.5955, eval loss 0.8165168166160583
Epoch 5 train loss: 0.5932, eval loss 0.8122302889823914
Epoch 6 train loss: 0.5842, eval loss 0.8081240057945251
Epoch 7 train loss: 0.5725, eval loss 0.8042243123054504
Epoch 8 train loss: 0.5704, eval loss 0.8005241751670837
Epoch 9 train loss: 0.5749, eval loss 0.7969619631767273
Epoch 10 train loss: 0.5674, eval loss 0.7935758829116821
Epoch 11 train loss: 0.5557, eval loss 0.790386974811554
Epoch 12 train loss: 0.5577, eval loss 0.7873281836509705
Epoch 13 train loss: 0.5490, eval loss 0.7844016551971436
Epoch 14 train loss: 0.5407, eval loss 0.7816137075424194
Epoch 15 train loss: 0.5328, eval loss 0.7789721488952637
Epoch 16 train loss: 0.5241, eval loss 0.7764389514923096
Epoch 17 train loss: 0.5194, eval loss 0.7740429043769836
Epoch 18 train loss: 0.5239, eval loss 0.7717512845993042
Epoch 19 train loss: 0.5186, eval loss 0.7695837616920471
Epoch 20 train loss: 0.5227, eval loss 0.767526388168335
Epoch 21 train loss: 0.5126, eval loss 0.765583336353302
Epoch 22 train loss: 0.5268, eval loss 0.7636951804161072
Epoch 23 train loss: 0.5062, eval loss 0.7618765234947205
Epoch 24 train loss: 0.5093, eval loss 0.7601606845855713
Epoch 25 train loss: 0.5054, eval loss 0.7584608793258667
Epoch 26 train loss: 0.5059, eval loss 0.7568527460098267
Epoch 27 train loss: 0.4951, eval loss 0.755283772945404
Epoch 28 train loss: 0.4929, eval loss 0.7537832260131836
Epoch 29 train loss: 0.5036, eval loss 0.7523267865180969
Epoch 30 train loss: 0.4920, eval loss 0.7509069442749023
Epoch 31 train loss: 0.4897, eval loss 0.7495148777961731
Epoch 32 train loss: 0.4953, eval loss 0.748151957988739
Epoch 33 train loss: 0.4875, eval loss 0.7468515038490295
Epoch 34 train loss: 0.4807, eval loss 0.7455230951309204
Epoch 35 train loss: 0.4841, eval loss 0.7442586421966553
Epoch 36 train loss: 0.4817, eval loss 0.7430747747421265
Epoch 37 train loss: 0.4866, eval loss 0.7418782114982605
Epoch 38 train loss: 0.4707, eval loss 0.7406918406486511
Epoch 39 train loss: 0.4680, eval loss 0.7395147681236267
Epoch 40 train loss: 0.4766, eval loss 0.7383894324302673
Epoch 41 train loss: 0.4543, eval loss 0.7373298406600952
Epoch 42 train loss: 0.4674, eval loss 0.7362666726112366
Epoch 43 train loss: 0.4543, eval loss 0.7352033257484436
Epoch 44 train loss: 0.4661, eval loss 0.7341445088386536
Epoch 45 train loss: 0.4621, eval loss 0.7331475615501404
Epoch 46 train loss: 0.4429, eval loss 0.7321717739105225
Epoch 47 train loss: 0.4412, eval loss 0.7312132716178894
Epoch 48 train loss: 0.4419, eval loss 0.7302569150924683
Epoch 49 train loss: 0.4471, eval loss 0.7292929291725159
Epoch 50 train loss: 0.4373, eval loss 0.7284128069877625
Epoch 51 train loss: 0.4282, eval loss 0.7275411486625671
Epoch 52 train loss: 0.4393, eval loss 0.7267094850540161
Epoch 53 train loss: 0.4187, eval loss 0.7259257435798645
Epoch 54 train loss: 0.4343, eval loss 0.7251390814781189
Epoch 55 train loss: 0.4325, eval loss 0.724348247051239
Epoch 56 train loss: 0.4264, eval loss 0.7235807180404663
Epoch 57 train loss: 0.4180, eval loss 0.7228695154190063
Epoch 58 train loss: 0.4230, eval loss 0.7221900224685669
Epoch 59 train loss: 0.4105, eval loss 0.7214955687522888
```

```
Epoch 60 train loss: 0.4244, eval loss 0.7208036184310913
Epoch 61 train loss: 0.4243, eval loss 0.7201569676399231
Epoch 62 train loss: 0.4156, eval loss 0.7195321917533875
Epoch 63 train loss: 0.4221, eval loss 0.7189339995384216
Epoch 64 train loss: 0.4202, eval loss 0.718332827091217
Epoch 65 train loss: 0.4291, eval loss 0.7177069187164307
Epoch 66 train loss: 0.4177, eval loss 0.7171757817268372
Epoch 67 train loss: 0.4044, eval loss 0.7166396379470825
Epoch 68 train loss: 0.4031, eval loss 0.7161715030670166
Epoch 69 train loss: 0.4140, eval loss 0.7156906723976135
Epoch 70 train loss: 0.4237, eval loss 0.7152223587036133
Epoch 71 train loss: 0.3910, eval loss 0.7147325873374939
Epoch 72 train loss: 0.4027, eval loss 0.7142418622970581
Epoch 73 train loss: 0.4078, eval loss 0.7137539386749268
Epoch 74 train loss: 0.3774, eval loss 0.7133319973945618
Epoch 75 train loss: 0.4068, eval loss 0.7128475904464722
Epoch 76 train loss: 0.3957, eval loss 0.7124720215797424
Epoch 77 train loss: 0.4137, eval loss 0.7120166420936584
Epoch 78 train loss: 0.4080, eval loss 0.711646556854248
Epoch 79 train loss: 0.3917, eval loss 0.7112928032875061
Epoch 80 train loss: 0.3949, eval loss 0.7109098434448242
Epoch 81 train loss: 0.4092, eval loss 0.7106028199195862
Epoch 82 train loss: 0.3972, eval loss 0.710250198841095
Epoch 83 train loss: 0.4071, eval loss 0.7099107503890991
Epoch 84 train loss: 0.3989, eval loss 0.7095770239830017
Epoch 85 train loss: 0.4042, eval loss 0.7092357873916626
Epoch 86 train loss: 0.4014, eval loss 0.7089340090751648
Epoch 87 train loss: 0.3815, eval loss 0.7086380124092102
Epoch 88 train loss: 0.3989, eval loss 0.7082684636116028
Epoch 89 train loss: 0.3787, eval loss 0.707989513874054
Epoch 90 train loss: 0.3707, eval loss 0.7077352404594421
Epoch 91 train loss: 0.4141, eval loss 0.7075051069259644
Epoch 92 train loss: 0.3743, eval loss 0.7072166204452515
Epoch 93 train loss: 0.3883, eval loss 0.706953763961792
Epoch 94 train loss: 0.3709, eval loss 0.7067587375640869
Epoch 95 train loss: 0.4231, eval loss 0.7065950632095337
Epoch 96 train loss: 0.3857, eval loss 0.7062981724739075
Epoch 97 train loss: 0.3867, eval loss 0.706038773059845
Epoch 98 train loss: 0.4217, eval loss 0.7058247327804565
Epoch 99 train loss: 0.3889, eval loss 0.7055943012237549
Epoch 100 train loss: 0.4002, eval loss 0.7053996324539185
Epoch 101 train loss: 0.3742, eval loss 0.7051815986633301
Epoch 102 train loss: 0.3889, eval loss 0.7050158977508545
Epoch 103 train loss: 0.3896, eval loss 0.7048733234405518
Epoch 104 train loss: 0.3921, eval loss 0.7047006487846375
Epoch 105 train loss: 0.3642, eval loss 0.7045515179634094
Epoch 106 train loss: 0.3939, eval loss 0.7043105959892273
Epoch 107 train loss: 0.3687, eval loss 0.7041144967079163
Epoch 108 train loss: 0.3829, eval loss 0.7039615511894226
Epoch 109 train loss: 0.3850, eval loss 0.7038257718086243
Epoch 110 train loss: 0.3760, eval loss 0.7036851644515991
Epoch 111 train loss: 0.3958, eval loss 0.7035030722618103
Epoch 112 train loss: 0.4049, eval loss 0.7033950090408325
Epoch 113 train loss: 0.3858, eval loss 0.7032511830329895
Epoch 114 train loss: 0.3782, eval loss 0.7031558752059937
Epoch 115 train loss: 0.4004, eval loss 0.703014612197876
Epoch 116 train loss: 0.4058, eval loss 0.7028425931930542
Epoch 117 train loss: 0.3898, eval loss 0.7027133107185364
Epoch 118 train loss: 0.3792, eval loss 0.7025798559188843
Epoch 119 train loss: 0.3817, eval loss 0.7024667859077454
```

```
Epoch 120 train loss: 0.3994, eval loss 0.7022824287414551
Epoch 121 train loss: 0.3734, eval loss 0.7022104263305664
Epoch 122 train loss: 0.3709, eval loss 0.7021746635437012
Epoch 123 train loss: 0.3569, eval loss 0.7020354270935059
Epoch 124 train loss: 0.3477, eval loss 0.7018452882766724
Epoch 125 train loss: 0.3865, eval loss 0.7017658352851868
Epoch 126 train loss: 0.4314, eval loss 0.7016569972038269
Epoch 127 train loss: 0.3639, eval loss 0.7015568614006042
Epoch 128 train loss: 0.3873, eval loss 0.7014085650444031
Epoch 129 train loss: 0.4007, eval loss 0.7013245820999146
Epoch 130 train loss: 0.3798, eval loss 0.7012611627578735
Epoch 131 train loss: 0.3601, eval loss 0.701178252696991
Epoch 132 train loss: 0.4057, eval loss 0.7010915279388428
Epoch 133 train loss: 0.3956, eval loss 0.7009837627410889
Epoch 134 train loss: 0.3609, eval loss 0.700900673866272
Epoch 135 train loss: 0.3748, eval loss 0.7008785009384155
Epoch 136 train loss: 0.3820, eval loss 0.7007763385772705
Epoch 137 train loss: 0.4141, eval loss 0.700709342956543
Epoch 138 train loss: 0.3669, eval loss 0.7006173133850098
Epoch 139 train loss: 0.3647, eval loss 0.7005269527435303
Epoch 140 train loss: 0.3911, eval loss 0.7004841566085815
Epoch 141 train loss: 0.4138, eval loss 0.7003775238990784
Epoch 142 train loss: 0.3733, eval loss 0.7003341317176819
Epoch 143 train loss: 0.3980, eval loss 0.7002590298652649
Epoch 144 train loss: 0.3989, eval loss 0.7000805139541626
Epoch 145 train loss: 0.3821, eval loss 0.7000865340232849
Epoch 146 train loss: 0.3700, eval loss 0.7000086903572083
Epoch 147 train loss: 0.3888, eval loss 0.6999402046203613
Epoch 148 train loss: 0.4001, eval loss 0.699894905090332
Epoch 149 train loss: 0.4121, eval loss 0.6998023986816406
Epoch 150 train loss: 0.4021, eval loss 0.6997036337852478
Epoch 151 train loss: 0.3726, eval loss 0.6996412873268127
Epoch 152 train loss: 0.3852, eval loss 0.699583113193512
Epoch 153 train loss: 0.3801, eval loss 0.6994649171829224
Epoch 154 train loss: 0.3751, eval loss 0.699383020401001
Epoch 155 train loss: 0.3749, eval loss 0.6993076801300049
Epoch 156 train loss: 0.4096, eval loss 0.6991987228393555
Epoch 157 train loss: 0.3866, eval loss 0.6990968585014343
Epoch 158 train loss: 0.3576, eval loss 0.6990466117858887
Epoch 159 train loss: 0.3767, eval loss 0.6989502906799316
Epoch 160 train loss: 0.3875, eval loss 0.6988334655761719
Epoch 161 train loss: 0.3716, eval loss 0.6987734436988831
Epoch 162 train loss: 0.3985, eval loss 0.6987884640693665
Epoch 163 train loss: 0.4013, eval loss 0.6987117528915405
Epoch 164 train loss: 0.3839, eval loss 0.6986832618713379
Epoch 165 train loss: 0.3500, eval loss 0.6986438035964966
Epoch 166 train loss: 0.3815, eval loss 0.698599100112915
Epoch 167 train loss: 0.3943, eval loss 0.6985163688659668
Epoch 168 train loss: 0.3692, eval loss 0.6984887719154358
Epoch 169 train loss: 0.3485, eval loss 0.6984305381774902
Epoch 170 train loss: 0.3770, eval loss 0.6983975172042847
Epoch 171 train loss: 0.3857, eval loss 0.6982951760292053
Epoch 172 train loss: 0.3757, eval loss 0.6982659697532654
Epoch 173 train loss: 0.3522, eval loss 0.6982356905937195
Epoch 174 train loss: 0.3520, eval loss 0.6981613636016846
Epoch 175 train loss: 0.3408, eval loss 0.6981473565101624
Epoch 176 train loss: 0.4114, eval loss 0.6981222033500671
Epoch 177 train loss: 0.3732, eval loss 0.6980452537536621
Epoch 178 train loss: 0.4294, eval loss 0.6979841589927673
Epoch 179 train loss: 0.3836, eval loss 0.6978399157524109
```

```
Epoch 180 train loss: 0.3909, eval loss 0.6977853775024414
Epoch 181 train loss: 0.3678, eval loss 0.6977248787879944
Epoch 182 train loss: 0.3756, eval loss 0.6976987719535828
Epoch 183 train loss: 0.3772, eval loss 0.6976467967033386
Epoch 184 train loss: 0.4129, eval loss 0.6975979804992676
Epoch 185 train loss: 0.3833, eval loss 0.6975610256195068
Epoch 186 train loss: 0.3825, eval loss 0.697484016418457
Epoch 187 train loss: 0.3801, eval loss 0.6975089311599731
Epoch 188 train loss: 0.3598, eval loss 0.6974390745162964
Epoch 189 train loss: 0.3943, eval loss 0.6973654627799988
Epoch 190 train loss: 0.3624, eval loss 0.6972814202308655
Epoch 191 train loss: 0.3598, eval loss 0.6972253918647766
Epoch 192 train loss: 0.4121, eval loss 0.6972525119781494
Epoch 193 train loss: 0.3732, eval loss 0.6972081661224365
Epoch 194 train loss: 0.3914, eval loss 0.6971340179443359
Epoch 195 train loss: 0.3875, eval loss 0.6970708966255188
Epoch 196 train loss: 0.3774, eval loss 0.6970222592353821
Epoch 197 train loss: 0.3845, eval loss 0.6969365477561951
Epoch 198 train loss: 0.4018, eval loss 0.6969214677810669
Epoch 199 train loss: 0.3625, eval loss 0.696811854839325
Epoch 200 train loss: 0.3812, eval loss 0.6968382596969604
Epoch 201 train loss: 0.3793, eval loss 0.6968482732772827
Epoch 202 train loss: 0.3633, eval loss 0.6968106031417847
Epoch 203 train loss: 0.3784, eval loss 0.6967118978500366
Epoch 204 train loss: 0.3905, eval loss 0.6967114806175232
Epoch 205 train loss: 0.3906, eval loss 0.6966749429702759
Epoch 206 train loss: 0.3477, eval loss 0.6965528130531311
Epoch 207 train loss: 0.4214, eval loss 0.6965188980102539
Epoch 208 train loss: 0.3780, eval loss 0.6965135931968689
Epoch 209 train loss: 0.3823, eval loss 0.6964234709739685
Epoch 210 train loss: 0.3601, eval loss 0.696414053440094
Epoch 211 train loss: 0.3753, eval loss 0.6964148283004761
Epoch 212 train loss: 0.3805, eval loss 0.6963136792182922
Epoch 213 train loss: 0.3915, eval loss 0.6962493062019348
Epoch 214 train loss: 0.4010, eval loss 0.6962209939956665
Epoch 215 train loss: 0.3987, eval loss 0.6961894631385803
Epoch 216 train loss: 0.3723, eval loss 0.6961283087730408
Epoch 217 train loss: 0.3649, eval loss 0.6960132718086243
Epoch 218 train loss: 0.3704, eval loss 0.6960404515266418
Epoch 219 train loss: 0.3940, eval loss 0.6959710717201233
Epoch 220 train loss: 0.3651, eval loss 0.695892870426178
Epoch 221 train loss: 0.3708, eval loss 0.6959003806114197
Epoch 222 train loss: 0.3344, eval loss 0.6958309412002563
Epoch 223 train loss: 0.4226, eval loss 0.6957778334617615
Epoch 224 train loss: 0.3976, eval loss 0.6958048343658447
Epoch 225 train loss: 0.3846, eval loss 0.6957671642303467
Epoch 226 train loss: 0.3878, eval loss 0.6957259178161621
Epoch 227 train loss: 0.3624, eval loss 0.6956996917724609
Epoch 228 train loss: 0.3795, eval loss 0.6956557035446167
Epoch 229 train loss: 0.3676, eval loss 0.6956471800804138
Epoch 230 train loss: 0.3607, eval loss 0.695625901222229
Epoch 231 train loss: 0.3759, eval loss 0.6955676078796387
Epoch 232 train loss: 0.4148, eval loss 0.6955001354217529
Epoch 233 train loss: 0.3568, eval loss 0.6954624652862549
Epoch 234 train loss: 0.3386, eval loss 0.6953831911087036
Epoch 235 train loss: 0.3315, eval loss 0.6953690052032471
Epoch 236 train loss: 0.3503, eval loss 0.6953175663948059
Epoch 237 train loss: 0.3707, eval loss 0.6952835917472839
Epoch 238 train loss: 0.3925, eval loss 0.6952733397483826
Epoch 239 train loss: 0.3793, eval loss 0.6952155232429504
```

```
Epoch 240 train loss: 0.3315, eval loss 0.695175290107727
Epoch 241 train loss: 0.3691, eval loss 0.6951366066932678
Epoch 242 train loss: 0.3741, eval loss 0.6950811743736267
Epoch 243 train loss: 0.3732, eval loss 0.6950109004974365
Epoch 244 train loss: 0.3766, eval loss 0.6950476765632629
Epoch 245 train loss: 0.3921, eval loss 0.6950182914733887
Epoch 246 train loss: 0.3633, eval loss 0.6949671506881714
Epoch 247 train loss: 0.3681, eval loss 0.695021390914917
Epoch 248 train loss: 0.3673, eval loss 0.6949431896209717
Epoch 249 train loss: 0.3560, eval loss 0.6949110627174377
Epoch 250 train loss: 0.3780, eval loss 0.6948904991149902
Epoch 251 train loss: 0.3613, eval loss 0.6948779821395874
Epoch 252 train loss: 0.3554, eval loss 0.6947166323661804
Epoch 253 train loss: 0.3652, eval loss 0.6947092413902283
Epoch 254 train loss: 0.3657, eval loss 0.6947083473205566
Epoch 255 train loss: 0.3739, eval loss 0.6947410702705383
Epoch 256 train loss: 0.3706, eval loss 0.6946691870689392
Epoch 257 train loss: 0.3663, eval loss 0.694666862487793
Epoch 258 train loss: 0.3826, eval loss 0.6945868730545044
Epoch 259 train loss: 0.4023, eval loss 0.6945371031761169
Epoch 260 train loss: 0.3382, eval loss 0.6945485472679138
Epoch 261 train loss: 0.3449, eval loss 0.6945175528526306
Epoch 262 train loss: 0.3625, eval loss 0.6944393515586853
Epoch 263 train loss: 0.3423, eval loss 0.6944217085838318
Epoch 264 train loss: 0.3695, eval loss 0.694378137588501
Epoch 265 train loss: 0.3709, eval loss 0.6943097710609436
Epoch 266 train loss: 0.4048, eval loss 0.6942969560623169
Epoch 267 train loss: 0.3819, eval loss 0.6942798495292664
Epoch 268 train loss: 0.3505, eval loss 0.6942678689956665
Epoch 269 train loss: 0.3662, eval loss 0.694242000579834
Epoch 270 train loss: 0.3520, eval loss 0.6942072510719299
Epoch 271 train loss: 0.3397, eval loss 0.6941716074943542
Epoch 272 train loss: 0.3619, eval loss 0.6941474676132202
Epoch 273 train loss: 0.3806, eval loss 0.6941177845001221
Epoch 274 train loss: 0.3390, eval loss 0.6940914392471313
Epoch 275 train loss: 0.3678, eval loss 0.6940489411354065
Epoch 276 train loss: 0.3526, eval loss 0.6940839886665344
Epoch 277 train loss: 0.3200, eval loss 0.6939892768859863
Epoch 278 train loss: 0.3341, eval loss 0.6939908862113953
Epoch 279 train loss: 0.3591, eval loss 0.6940016150474548
Epoch 280 train loss: 0.3696, eval loss 0.6940655708312988
Epoch 281 train loss: 0.3669, eval loss 0.6940117478370667
Epoch 282 train loss: 0.3620, eval loss 0.6939165592193604
Epoch 283 train loss: 0.3440, eval loss 0.6938917636871338
Epoch 284 train loss: 0.4015, eval loss 0.6939125061035156
Epoch 285 train loss: 0.3949, eval loss 0.693828284740448
Epoch 286 train loss: 0.3564, eval loss 0.6938614249229431
Epoch 287 train loss: 0.3521, eval loss 0.6938629150390625
Epoch 288 train loss: 0.3657, eval loss 0.6938005089759827
Epoch 289 train loss: 0.3496, eval loss 0.6937581300735474
Epoch 290 train loss: 0.3895, eval loss 0.6937039494514465
Epoch 291 train loss: 0.3713, eval loss 0.6936453580856323
Epoch 292 train loss: 0.3676, eval loss 0.6936452984809875
Epoch 293 train loss: 0.3693, eval loss 0.6935561895370483
Epoch 294 train loss: 0.3805, eval loss 0.6935620903968811
Epoch 295 train loss: 0.3784, eval loss 0.6935297250747681
Epoch 296 train loss: 0.3671, eval loss 0.6935275793075562
Epoch 297 train loss: 0.3756, eval loss 0.6934828162193298
Epoch 298 train loss: 0.3485, eval loss 0.6934263110160828
Epoch 299 train loss: 0.3152, eval loss 0.6934300661087036
```

In [ ]:
```
test_metrics = evaluate_model(best_model, X_test, y_test, loss_fn, best_t

print(f"AUROC: {100 * test_metrics['auroc']:.2f}%")
print(f"F1: {100 * test_metrics['f1_score']:.2f}%")
print(f"Precision: {100 * test_metrics['precision_score']:.2f}%")
print(f"Recall: {100 * test_metrics['recall_score']:.2f}%")
```

```
AUROC: 90.25%
F1: 68.50%
Precision: 62.59%
Recall: 75.64%
```

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 (0.5 punktu)

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.

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

    self.layers = nn.Sequential(
      nn.Linear(input_size, 256),
      nn.BatchNorm1d(),
      nn.ReLU(),
      nn.Dropout(dropout_p),
      nn.Linear(256, 128),
      nn.BatchNorm1d(),
      nn.ReLU(dropout_p),
      nn.Dropout(),
      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, threshold: float = 0.5):
    y_pred_score = self.predict_proba(x)
    return (y_pred_score > threshold).to(torch.int32)
```

In [ ]:
```python
# define all the hyperparameters
learning_rate           = 1e-3
dropout_p               = 0.5
l2_reg                  = 1e-4
batch_size              = 128
max_epochs              = 300

early_stopping_patience = 4
```

In [ ]:
```python
# training loop
from sklearn.utils.class_weight import compute_class_weight

model = RegularizedMLP(
  input_size=X_train.shape[1],
  dropout_p=dropout_p
)

optimizer = torch.optim.AdamW(
  model.parameters(),
  lr=learning_rate,
  weight_decay=l2_reg
)

weights = compute_class_weight("balanced", classes=np.unique(y_train), y=
loss_fn = torch.nn.BCEWithLogitsLoss(pos_weight=torch.tensor(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
```

```python
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()

  valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
  if valid_metrics["loss_value"] < best_val_loss:
    best_model                = deepcopy(model)
    best_loss                 = valid_metrics["loss_value"]
    best_threshold            = valid_metrics["threshold"]
    steps_without_improvement = 0
  else:
    steps_without_improvement += 1
    if steps_without_improvement >= early_stopping_patience:
      print("Early stopping.")
      break

  print(f"Epoch {epoch_num} train loss: {loss.item():.4f}, eval loss {val
```

```
Epoch 0 train loss: 0.5629, eval loss 0.8193220496177673
Epoch 1 train loss: 0.5306, eval loss 0.8150820732116699
Epoch 2 train loss: 0.5135, eval loss 0.8128636479377747
Epoch 3 train loss: 0.5472, eval loss 0.8111741542816162
Epoch 4 train loss: 0.5212, eval loss 0.8113449811935425
Epoch 5 train loss: 0.5242, eval loss 0.8105567097663879
Epoch 6 train loss: 0.4768, eval loss 0.8092515468597412
Epoch 7 train loss: 0.4858, eval loss 0.8099179863929749
Epoch 8 train loss: 0.5164, eval loss 0.8091886639595032
Epoch 9 train loss: 0.5254, eval loss 0.8097275495529175
Epoch 10 train loss: 0.5637, eval loss 0.8083905577659607
Epoch 11 train loss: 0.4929, eval loss 0.8095413446426392
Epoch 12 train loss: 0.4753, eval loss 0.8078070878982544
Epoch 13 train loss: 0.5272, eval loss 0.8085379004478455
Epoch 14 train loss: 0.4813, eval loss 0.8069289922714233
Epoch 15 train loss: 0.4653, eval loss 0.8077027797698975
Epoch 16 train loss: 0.5005, eval loss 0.80728679895401
Epoch 17 train loss: 0.4982, eval loss 0.8065657615661621
Epoch 18 train loss: 0.4531, eval loss 0.8057993054389954
Epoch 19 train loss: 0.5112, eval loss 0.8077937364578247
Epoch 20 train loss: 0.5484, eval loss 0.8063627481460571
Epoch 21 train loss: 0.4982, eval loss 0.80589759349823
Epoch 22 train loss: 0.4841, eval loss 0.806516170501709
Epoch 23 train loss: 0.5069, eval loss 0.8044775724411011
Epoch 24 train loss: 0.4533, eval loss 0.8046756982803345
Epoch 25 train loss: 0.5010, eval loss 0.8064488768577576
Epoch 26 train loss: 0.4360, eval loss 0.8052209615707397
Epoch 27 train loss: 0.4546, eval loss 0.8043792843818665
Epoch 28 train loss: 0.4908, eval loss 0.8041768074035645
Epoch 29 train loss: 0.4312, eval loss 0.8034918308258057
Epoch 30 train loss: 0.4244, eval loss 0.8039012551307678
Epoch 31 train loss: 0.5120, eval loss 0.8041486144065857
Epoch 32 train loss: 0.4598, eval loss 0.804166316986084
Epoch 33 train loss: 0.4753, eval loss 0.8037480711936951
Epoch 34 train loss: 0.5000, eval loss 0.8053298592567444
Epoch 35 train loss: 0.4561, eval loss 0.8041807413101196
Epoch 36 train loss: 0.4539, eval loss 0.8042764663696289
Epoch 37 train loss: 0.4671, eval loss 0.8024868369102478
Epoch 38 train loss: 0.3966, eval loss 0.8041428923606873
Epoch 39 train loss: 0.4660, eval loss 0.8023486137390137
Epoch 40 train loss: 0.4503, eval loss 0.802384614944458
Epoch 41 train loss: 0.4580, eval loss 0.8022723197937012
Epoch 42 train loss: 0.4410, eval loss 0.8027135729789734
Epoch 43 train loss: 0.4839, eval loss 0.8027647137641907
Epoch 44 train loss: 0.4241, eval loss 0.8023892045021057
Epoch 45 train loss: 0.4769, eval loss 0.801960825920105
Epoch 46 train loss: 0.4977, eval loss 0.80154949426651
Epoch 47 train loss: 0.3815, eval loss 0.8034011721611023
Epoch 48 train loss: 0.4481, eval loss 0.8008747100830078
Epoch 49 train loss: 0.4483, eval loss 0.8017311096191406
Epoch 50 train loss: 0.4447, eval loss 0.8028062582015991
Epoch 51 train loss: 0.3782, eval loss 0.8020015954971313
Epoch 52 train loss: 0.3866, eval loss 0.8013289570808411
Epoch 53 train loss: 0.3962, eval loss 0.8004018664360046
Epoch 54 train loss: 0.4302, eval loss 0.8008633255958557
Epoch 55 train loss: 0.4106, eval loss 0.8021411299705505
Epoch 56 train loss: 0.5431, eval loss 0.8010581135749817
Epoch 57 train loss: 0.3945, eval loss 0.8023607134819031
Epoch 58 train loss: 0.4012, eval loss 0.8019098043441772
Epoch 59 train loss: 0.4175, eval loss 0.8006412982940674
```

```
Epoch 60 train loss: 0.4110, eval loss 0.8009186387062073
Epoch 61 train loss: 0.3615, eval loss 0.8017953038215637
Epoch 62 train loss: 0.3281, eval loss 0.8026179671287537
Epoch 63 train loss: 0.4115, eval loss 0.8020371198654175
Epoch 64 train loss: 0.3882, eval loss 0.8022221922874451
Epoch 65 train loss: 0.3703, eval loss 0.801249086856842
Epoch 66 train loss: 0.4234, eval loss 0.8015450835227966
Epoch 67 train loss: 0.4064, eval loss 0.8017520904541016
Epoch 68 train loss: 0.3508, eval loss 0.8015761971473694
Epoch 69 train loss: 0.4235, eval loss 0.8021101951599121
Epoch 70 train loss: 0.4079, eval loss 0.8027858138084412
Epoch 71 train loss: 0.3834, eval loss 0.8007208108901978
Epoch 72 train loss: 0.3800, eval loss 0.8001072406768799
Epoch 73 train loss: 0.3184, eval loss 0.8012461066246033
Epoch 74 train loss: 0.4074, eval loss 0.8020753264427185
Epoch 75 train loss: 0.3603, eval loss 0.80271977186203
Epoch 76 train loss: 0.3838, eval loss 0.8012269735336304
Epoch 77 train loss: 0.3576, eval loss 0.8008494973182678
Epoch 78 train loss: 0.3733, eval loss 0.8013019561767578
Epoch 79 train loss: 0.3834, eval loss 0.8021460175514221
Epoch 80 train loss: 0.3693, eval loss 0.8006877899169922
Epoch 81 train loss: 0.3834, eval loss 0.8005322217941284
Epoch 82 train loss: 0.3798, eval loss 0.8010575175285339
Epoch 83 train loss: 0.4373, eval loss 0.8011093735694885
Epoch 84 train loss: 0.3668, eval loss 0.8003514409065247
Epoch 85 train loss: 0.3536, eval loss 0.8018491864204407
Epoch 86 train loss: 0.4026, eval loss 0.8015285730361938
Epoch 87 train loss: 0.4216, eval loss 0.802093505859375
Epoch 88 train loss: 0.3317, eval loss 0.8017180562019348
Epoch 89 train loss: 0.3427, eval loss 0.801907479763031
Epoch 90 train loss: 0.3775, eval loss 0.8019225597381592
Epoch 91 train loss: 0.3888, eval loss 0.8012410402297974
Epoch 92 train loss: 0.3309, eval loss 0.7998663783073425
Epoch 93 train loss: 0.4071, eval loss 0.8012180924415588
Epoch 94 train loss: 0.4039, eval loss 0.8018346428871155
Epoch 95 train loss: 0.3545, eval loss 0.8015927076339722
Epoch 96 train loss: 0.3796, eval loss 0.8011771440505981
Epoch 97 train loss: 0.3809, eval loss 0.8007342219352722
Epoch 98 train loss: 0.3318, eval loss 0.8010013103485107
Epoch 99 train loss: 0.3297, eval loss 0.8010680079460144
Epoch 100 train loss: 0.3483, eval loss 0.8020141124725342
Epoch 101 train loss: 0.3450, eval loss 0.8004631400108337
Epoch 102 train loss: 0.3648, eval loss 0.7996147871017456
Epoch 103 train loss: 0.3282, eval loss 0.8008513450622559
Epoch 104 train loss: 0.3634, eval loss 0.800872802734375
Epoch 105 train loss: 0.3907, eval loss 0.8008595108985901
Epoch 106 train loss: 0.3820, eval loss 0.8000925183296204
Epoch 107 train loss: 0.3427, eval loss 0.8007583618164062
Epoch 108 train loss: 0.3580, eval loss 0.7992900013923645
Epoch 109 train loss: 0.3847, eval loss 0.8004890084266663
Epoch 110 train loss: 0.3508, eval loss 0.7999377846717834
Epoch 111 train loss: 0.3220, eval loss 0.7998749613761902
Epoch 112 train loss: 0.3128, eval loss 0.8000549077987671
Epoch 113 train loss: 0.3627, eval loss 0.8004485368728638
Epoch 114 train loss: 0.3767, eval loss 0.7995209693908691
Epoch 115 train loss: 0.3876, eval loss 0.8006343245506287
Epoch 116 train loss: 0.3662, eval loss 0.8001945614814758
Epoch 117 train loss: 0.3202, eval loss 0.8006505966186523
Epoch 118 train loss: 0.3819, eval loss 0.8001911640167236
Epoch 119 train loss: 0.3441, eval loss 0.8002772927284241
```

```
Epoch 120 train loss: 0.3740, eval loss 0.8009080290794373
Epoch 121 train loss: 0.3392, eval loss 0.7998436093330383
Epoch 122 train loss: 0.3926, eval loss 0.8011487126350403
Epoch 123 train loss: 0.3287, eval loss 0.8010822534561157
Epoch 124 train loss: 0.3336, eval loss 0.801827073097229
Epoch 125 train loss: 0.3979, eval loss 0.8013080358505249
Epoch 126 train loss: 0.3103, eval loss 0.7999158501625061
Epoch 127 train loss: 0.3830, eval loss 0.801666796207428
Epoch 128 train loss: 0.3636, eval loss 0.8013674020767212
Epoch 129 train loss: 0.3414, eval loss 0.8014193773269653
Epoch 130 train loss: 0.3270, eval loss 0.8015238642692566
Epoch 131 train loss: 0.3603, eval loss 0.8002179861068726
Epoch 132 train loss: 0.3710, eval loss 0.8011668920516968
Epoch 133 train loss: 0.3536, eval loss 0.802046537399292
Epoch 134 train loss: 0.3354, eval loss 0.8006907105445862
Epoch 135 train loss: 0.3685, eval loss 0.8009732961654663
Epoch 136 train loss: 0.4145, eval loss 0.8007218241691589
Epoch 137 train loss: 0.3316, eval loss 0.799526035785675
Epoch 138 train loss: 0.2873, eval loss 0.8003090023994446
Epoch 139 train loss: 0.3583, eval loss 0.8009865880012512
Epoch 140 train loss: 0.3602, eval loss 0.799116313457489
Epoch 141 train loss: 0.3174, eval loss 0.7994464635848999
Epoch 142 train loss: 0.3688, eval loss 0.8002268075942993
Epoch 143 train loss: 0.4758, eval loss 0.7996047735214233
Epoch 144 train loss: 0.3427, eval loss 0.8009233474731445
Epoch 145 train loss: 0.3214, eval loss 0.800198495388031
Epoch 146 train loss: 0.3848, eval loss 0.8004689812660217
Epoch 147 train loss: 0.3264, eval loss 0.8002051711082458
Epoch 148 train loss: 0.4095, eval loss 0.8004466891288757
Epoch 149 train loss: 0.3416, eval loss 0.8021256923675537
Epoch 150 train loss: 0.3293, eval loss 0.800346851348877
Epoch 151 train loss: 0.2685, eval loss 0.799700915813446
Epoch 152 train loss: 0.3507, eval loss 0.8000146150588989
Epoch 153 train loss: 0.3411, eval loss 0.8001171350479126
Epoch 154 train loss: 0.2903, eval loss 0.8008460402488708
Epoch 155 train loss: 0.3195, eval loss 0.8007574677467346
Epoch 156 train loss: 0.3471, eval loss 0.7996404767036438
Epoch 157 train loss: 0.3288, eval loss 0.8006081581115723
Epoch 158 train loss: 0.3314, eval loss 0.8012731671333313
Epoch 159 train loss: 0.3030, eval loss 0.8005839586257935
Epoch 160 train loss: 0.3322, eval loss 0.800290048122406
Epoch 161 train loss: 0.3141, eval loss 0.8001722693443298
Epoch 162 train loss: 0.2898, eval loss 0.8024561405181885
Epoch 163 train loss: 0.3354, eval loss 0.8001542687416077
Epoch 164 train loss: 0.3392, eval loss 0.799476683139801
Epoch 165 train loss: 0.3062, eval loss 0.7999184727668762
Epoch 166 train loss: 0.4333, eval loss 0.8002081513404846
Epoch 167 train loss: 0.3359, eval loss 0.7993507981300354
Epoch 168 train loss: 0.3158, eval loss 0.8010748624801636
Epoch 169 train loss: 0.3385, eval loss 0.800112247467041
Epoch 170 train loss: 0.3143, eval loss 0.8002913594245911
Epoch 171 train loss: 0.3700, eval loss 0.8003752827644348
Epoch 172 train loss: 0.3301, eval loss 0.8019357919692993
Epoch 173 train loss: 0.3488, eval loss 0.8010488152503967
Epoch 174 train loss: 0.3131, eval loss 0.7999782562255859
Epoch 175 train loss: 0.2952, eval loss 0.7999871373176575
Epoch 176 train loss: 0.3521, eval loss 0.8007956147193909
Epoch 177 train loss: 0.3027, eval loss 0.8010517358779907
Epoch 178 train loss: 0.3707, eval loss 0.8015604615211487
Epoch 179 train loss: 0.3281, eval loss 0.8008806705474854
```

```
Epoch 180 train loss: 0.3904, eval loss 0.800868809223175
Epoch 181 train loss: 0.3657, eval loss 0.8010715246200562
Epoch 182 train loss: 0.3094, eval loss 0.8002661466598511
Epoch 183 train loss: 0.3047, eval loss 0.7996224761009216
Epoch 184 train loss: 0.3250, eval loss 0.8000989556312561
Epoch 185 train loss: 0.2801, eval loss 0.800008237361908
Epoch 186 train loss: 0.2652, eval loss 0.8011844158172607
Epoch 187 train loss: 0.4133, eval loss 0.7990203499794006
Epoch 188 train loss: 0.2948, eval loss 0.8004269599914551
Epoch 189 train loss: 0.3363, eval loss 0.8000019788742065
Epoch 190 train loss: 0.3682, eval loss 0.8013685941696167
Epoch 191 train loss: 0.2885, eval loss 0.800675094127655
Epoch 192 train loss: 0.2689, eval loss 0.7999104857444763
Epoch 193 train loss: 0.3068, eval loss 0.8003748059272766
Epoch 194 train loss: 0.3106, eval loss 0.7998701930046082
Epoch 195 train loss: 0.3393, eval loss 0.8006613850593567
Epoch 196 train loss: 0.3144, eval loss 0.8001955151557922
Epoch 197 train loss: 0.3390, eval loss 0.7996159195899963
Epoch 198 train loss: 0.3127, eval loss 0.8002064824104309
Epoch 199 train loss: 0.3056, eval loss 0.8003863096237183
Epoch 200 train loss: 0.3147, eval loss 0.8002702593803406
Epoch 201 train loss: 0.3564, eval loss 0.7996562123298645
Epoch 202 train loss: 0.3086, eval loss 0.8002240061759949
Epoch 203 train loss: 0.3120, eval loss 0.8016101121902466
Epoch 204 train loss: 0.2952, eval loss 0.8009035587310791
Epoch 205 train loss: 0.3125, eval loss 0.801316499710083
Epoch 206 train loss: 0.3075, eval loss 0.7998898029327393
Epoch 207 train loss: 0.2947, eval loss 0.8006877303123474
Epoch 208 train loss: 0.3149, eval loss 0.8004685044288635
Epoch 209 train loss: 0.3458, eval loss 0.7998011112213135
Epoch 210 train loss: 0.3095, eval loss 0.8005020618438721
Epoch 211 train loss: 0.3305, eval loss 0.8015560507774353
Epoch 212 train loss: 0.3342, eval loss 0.8006479144096375
Epoch 213 train loss: 0.2555, eval loss 0.8008430004119873
Epoch 214 train loss: 0.3222, eval loss 0.8003252744674683
Epoch 215 train loss: 0.3160, eval loss 0.7998379468917847
Epoch 216 train loss: 0.3006, eval loss 0.7998182773590088
Epoch 217 train loss: 0.2740, eval loss 0.7991926670074463
Epoch 218 train loss: 0.3647, eval loss 0.8004252314567566
Epoch 219 train loss: 0.3221, eval loss 0.7997629046440125
Epoch 220 train loss: 0.3439, eval loss 0.7998244762420654
Epoch 221 train loss: 0.3759, eval loss 0.8010115027427673
Epoch 222 train loss: 0.3112, eval loss 0.8003992438316345
Epoch 223 train loss: 0.3140, eval loss 0.8001256585121155
Epoch 224 train loss: 0.2726, eval loss 0.7999143600463867
Epoch 225 train loss: 0.2829, eval loss 0.8009320497512817
Epoch 226 train loss: 0.3552, eval loss 0.799959123134613
Epoch 227 train loss: 0.3059, eval loss 0.7991829514503479
Epoch 228 train loss: 0.3002, eval loss 0.7997986078262329
Epoch 229 train loss: 0.3758, eval loss 0.8002955913543701
Epoch 230 train loss: 0.3178, eval loss 0.8008468747138977
Epoch 231 train loss: 0.2457, eval loss 0.7990339994430542
Epoch 232 train loss: 0.2806, eval loss 0.7991597056388855
Epoch 233 train loss: 0.3512, eval loss 0.8000616431236267
Epoch 234 train loss: 0.3161, eval loss 0.7996057868003845
Epoch 235 train loss: 0.2714, eval loss 0.7997676730155945
Epoch 236 train loss: 0.3350, eval loss 0.8002623915672302
Epoch 237 train loss: 0.2993, eval loss 0.8002688884735107
Epoch 238 train loss: 0.2658, eval loss 0.7999210953712463
Epoch 239 train loss: 0.3165, eval loss 0.7995321750640869
```

```
Epoch 240 train loss: 0.2411, eval loss 0.8001236915588379
Epoch 241 train loss: 0.3401, eval loss 0.7999554872512817
Epoch 242 train loss: 0.3038, eval loss 0.7995222210884094
Epoch 243 train loss: 0.3194, eval loss 0.8011464476585388
Epoch 244 train loss: 0.2988, eval loss 0.7995145320892334
Epoch 245 train loss: 0.3381, eval loss 0.7996189594268799
Epoch 246 train loss: 0.2529, eval loss 0.8000696301460266
Epoch 247 train loss: 0.3435, eval loss 0.801456868648529
Epoch 248 train loss: 0.3640, eval loss 0.8001944422721863
Epoch 249 train loss: 0.2579, eval loss 0.7998297214508057
Epoch 250 train loss: 0.2952, eval loss 0.7997647523880005
Epoch 251 train loss: 0.3520, eval loss 0.7995056509971619
Epoch 252 train loss: 0.2580, eval loss 0.8001390099525452
Epoch 253 train loss: 0.2804, eval loss 0.8008912801742554
Epoch 254 train loss: 0.3247, eval loss 0.7999724745750427
Epoch 255 train loss: 0.3048, eval loss 0.7994735836982727
Epoch 256 train loss: 0.2610, eval loss 0.7999934554100037
Epoch 257 train loss: 0.2858, eval loss 0.799811065196991
Epoch 258 train loss: 0.3579, eval loss 0.8001388907432556
Epoch 259 train loss: 0.3041, eval loss 0.8001155853271484
Epoch 260 train loss: 0.3263, eval loss 0.8003233075141907
Epoch 261 train loss: 0.2787, eval loss 0.7998843789100647
Epoch 262 train loss: 0.2923, eval loss 0.7992346882820129
Epoch 263 train loss: 0.2834, eval loss 0.799951434135437
Epoch 264 train loss: 0.2987, eval loss 0.7997220158576965
Epoch 265 train loss: 0.3402, eval loss 0.8000121712684631
Epoch 266 train loss: 0.2871, eval loss 0.8011218309402466
Epoch 267 train loss: 0.3385, eval loss 0.799760639667511
Epoch 268 train loss: 0.3115, eval loss 0.7992111444473267
Epoch 269 train loss: 0.3854, eval loss 0.7994728088378906
Epoch 270 train loss: 0.3119, eval loss 0.7996938228607178
Epoch 271 train loss: 0.3344, eval loss 0.7989160418510437
Epoch 272 train loss: 0.3195, eval loss 0.7992215752601624
Epoch 273 train loss: 0.2881, eval loss 0.8001205921173096
Epoch 274 train loss: 0.3170, eval loss 0.8000084161758423
Epoch 275 train loss: 0.2945, eval loss 0.7999417185783386
Epoch 276 train loss: 0.2811, eval loss 0.7998498678207397
Epoch 277 train loss: 0.2943, eval loss 0.7990300059318542
Epoch 278 train loss: 0.2801, eval loss 0.7985392808914185
Epoch 279 train loss: 0.2904, eval loss 0.7998002767562866
Epoch 280 train loss: 0.3425, eval loss 0.8003681302070618
Epoch 281 train loss: 0.3675, eval loss 0.799590528011322
Epoch 282 train loss: 0.2624, eval loss 0.8000451922416687
Epoch 283 train loss: 0.2652, eval loss 0.8009237051010132
Epoch 284 train loss: 0.2869, eval loss 0.7993877530097961
Epoch 285 train loss: 0.3085, eval loss 0.8001387119293213
Epoch 286 train loss: 0.3031, eval loss 0.7992757558822632
Epoch 287 train loss: 0.2561, eval loss 0.7994285821914673
Epoch 288 train loss: 0.2811, eval loss 0.7990698218345642
Epoch 289 train loss: 0.2320, eval loss 0.7988684177398682
Epoch 290 train loss: 0.3230, eval loss 0.7995150685310364
Epoch 291 train loss: 0.3787, eval loss 0.8005328178405762
Epoch 292 train loss: 0.2927, eval loss 0.7991448044776917
Epoch 293 train loss: 0.2654, eval loss 0.7987827658653259
Epoch 294 train loss: 0.3236, eval loss 0.7991222143173218
Epoch 295 train loss: 0.2860, eval loss 0.7997121214866638
Epoch 296 train loss: 0.4241, eval loss 0.7994574308395386
Epoch 297 train loss: 0.2732, eval loss 0.8002296090126038
Epoch 298 train loss: 0.2435, eval loss 0.7991071939468384
Epoch 299 train loss: 0.3844, eval loss 0.799307107925415
```

In [ ]:
```
test_metrics = evaluate_model(best_model, X_test, y_test, loss_fn, best_t

print(f"AUROC: {100 * test_metrics['auroc']:.2f}%")
print(f"F1: {100 * test_metrics['f1_score']:.2f}%")
print(f"Precision: {100 * test_metrics['precision_score']:.2f}%")
print(f"Recall: {100 * test_metrics['recall_score']:.2f}%")
```

```
AUROC: 89.22%
F1: 66.39%
Precision: 62.65%
Recall: 70.60%
```

## 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):

In [ ]:
```
import time


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

        self.mlp = nn.Sequential(
            nn.Linear(input_size, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(256, 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, threshold: float = 0.5):
        y_pred_score = self.predict_proba(x)
        return (y_pred_score > threshold).to(torch.int32)


model = CudaMLP(X_train.shape[1]).to('cuda')

optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate, weigh

# note that we are using loss function with sigmoid built in
loss_fn = torch.nn.BCEWithLogitsLoss(pos_weight=torch.from_numpy(weights)

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_from_eval = time.time()

        step_counter += 1

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

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

Co prawda ten model nie będzie tak dobry jak ten z laboratorium, ale zwróć uwagę, o ile jest większy, a przy tym szybszy.

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

In [ ]: