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 [1]:

```
# for conda users
%conda install -y matplotlib pandas pytorch torchvision torchaudio -c pytorch -c conda-f
orge
```

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

## Package Plan ##

  environment location: C:\Users\Wojtek\anaconda3

  added / updated specs:
    - matplotlib
    - pandas
    - pytorch
    - torchaudio
    - torchvision


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    conda-22.9.0               |   py39hcbf5309_2         985 KB  conda-forge
    libuv-1.44.2               |       h8ffe710_0         362 KB  conda-forge
    pytorch-1.13.0             |      py3.9_cpu_0       138.2 MB  pytorch
    pytorch-mutex-1.0          |              cpu           3 KB  pytorch
    torchaudio-0.13.0          |         py39_cpu         4.5 MB  pytorch
    torchvision-0.14.0         |         py39_cpu         6.3 MB  pytorch
    ------------------------------------------------------------
                                           Total:       150.3 MB

The following NEW packages will be INSTALLED:

  libuv              conda-forge/win-64::libuv-1.44.2-h8ffe710_0 None
  pytorch            pytorch/win-64::pytorch-1.13.0-py3.9_cpu_0 None
  pytorch-mutex      pytorch/noarch::pytorch-mutex-1.0-cpu None
  torchaudio         pytorch/win-64::torchaudio-0.13.0-py39_cpu None
  torchvision        pytorch/win-64::torchvision-0.14.0-py39_cpu None

The following packages will be UPDATED:

  conda                               22.9.0-py39hcbf5309_1 --> 22.9.0-py39hcbf5309_2 No
ne



Downloading and Extracting Packages

pytorch-1.13.0       | 138.2 MB  |            |   0%
pytorch-1.13.0       | 138.2 MB  |            |   0%
pytorch-1.13.0       | 138.2 MB  |            |   0%
pytorch-1.13.0       | 138.2 MB  | 1          |   1%
pytorch-1.13.0       | 138.2 MB  | 1          |   2%
pytorch-1.13.0       | 138.2 MB  | 2          |   2%
pytorch-1.13.0       | 138.2 MB  | 2          |   3%
pytorch-1.13.0       | 138.2 MB  | 3          |   4%
pytorch-1.13.0       | 138.2 MB  | 4          |   4%
pytorch-1.13.0       | 138.2 MB  | 5          |   5%
pytorch-1.13.0       | 138.2 MB  | 5          |   6%
pytorch-1.13.0       | 138.2 MB  | 6          |   7%
pytorch-1.13.0       | 138.2 MB  | 7          |   7%
pytorch-1.13.0       | 138.2 MB  | 8          |   9%
pytorch-1.13.0       | 138.2 MB  | 9          |  10%
pytorch-1.13.0       | 138.2 MB  | #1         |  11%
pytorch-1.13.0       | 138.2 MB  | #2         |  12%
pytorch-1.13.0       | 138.2 MB  | #3         |  14%
pytorch-1.13.0       | 138.2 MB  | #4         |  15%
pytorch-1.13.0       | 138.2 MB  | #6         |  16%
pytorch-1.13.0       | 138.2 MB  | #7         |  18%
```

```
pytorch-1.13.0       | 138.2 MB  | #8         |  19%
pytorch-1.13.0       | 138.2 MB  | ##         |  20%
pytorch-1.13.0       | 138.2 MB  | ##1        |  21%
pytorch-1.13.0       | 138.2 MB  | ##2        |  23%
pytorch-1.13.0       | 138.2 MB  | ##3        |  24%
pytorch-1.13.0       | 138.2 MB  | ##5        |  25%
pytorch-1.13.0       | 138.2 MB  | ##6        |  26%
pytorch-1.13.0       | 138.2 MB  | ##7        |  27%
pytorch-1.13.0       | 138.2 MB  | ##8        |  29%
pytorch-1.13.0       | 138.2 MB  | ##9        |  30%
pytorch-1.13.0       | 138.2 MB  | ###        |  31%
pytorch-1.13.0       | 138.2 MB  | ###1       |  32%
pytorch-1.13.0       | 138.2 MB  | ###2       |  33%
pytorch-1.13.0       | 138.2 MB  | ###3       |  34%
pytorch-1.13.0       | 138.2 MB  | ###4       |  35%
pytorch-1.13.0       | 138.2 MB  | ###5       |  36%
pytorch-1.13.0       | 138.2 MB  | ###6       |  37%
pytorch-1.13.0       | 138.2 MB  | ###8       |  38%
pytorch-1.13.0       | 138.2 MB  | ###9       |  39%
pytorch-1.13.0       | 138.2 MB  | ####       |  41%
pytorch-1.13.0       | 138.2 MB  | ####1      |  42%
pytorch-1.13.0       | 138.2 MB  | ####3      |  43%
pytorch-1.13.0       | 138.2 MB  | ####4      |  44%
pytorch-1.13.0       | 138.2 MB  | ####5      |  45%
pytorch-1.13.0       | 138.2 MB  | ####6      |  46%
pytorch-1.13.0       | 138.2 MB  | ####7      |  47%
pytorch-1.13.0       | 138.2 MB  | ####8      |  48%
pytorch-1.13.0       | 138.2 MB  | ####9      |  50%
pytorch-1.13.0       | 138.2 MB  | #####      |  51%
pytorch-1.13.0       | 138.2 MB  | #####2     |  52%
pytorch-1.13.0       | 138.2 MB  | #####3     |  53%
pytorch-1.13.0       | 138.2 MB  | #####4     |  55%
pytorch-1.13.0       | 138.2 MB  | #####6     |  56%
pytorch-1.13.0       | 138.2 MB  | #####7     |  58%
pytorch-1.13.0       | 138.2 MB  | #####8     |  59%
pytorch-1.13.0       | 138.2 MB  | ######     |  60%
pytorch-1.13.0       | 138.2 MB  | ######1    |  61%
pytorch-1.13.0       | 138.2 MB  | ######2    |  63%
pytorch-1.13.0       | 138.2 MB  | ######4    |  64%
pytorch-1.13.0       | 138.2 MB  | ######5    |  65%
pytorch-1.13.0       | 138.2 MB  | ######6    |  67%
pytorch-1.13.0       | 138.2 MB  | ######7    |  68%
pytorch-1.13.0       | 138.2 MB  | ######9    |  69%
pytorch-1.13.0       | 138.2 MB  | #######    |  71%
pytorch-1.13.0       | 138.2 MB  | #######1   |  72%
pytorch-1.13.0       | 138.2 MB  | #######3   |  73%
pytorch-1.13.0       | 138.2 MB  | #######4   |  75%
pytorch-1.13.0       | 138.2 MB  | #######5   |  76%
pytorch-1.13.0       | 138.2 MB  | #######7   |  77%
pytorch-1.13.0       | 138.2 MB  | #######8   |  79%
pytorch-1.13.0       | 138.2 MB  | #######9   |  80%
pytorch-1.13.0       | 138.2 MB  | ########1  |  81%
pytorch-1.13.0       | 138.2 MB  | ########2  |  83%
pytorch-1.13.0       | 138.2 MB  | ########4  |  84%
pytorch-1.13.0       | 138.2 MB  | ########5  |  85%
pytorch-1.13.0       | 138.2 MB  | ########6  |  87%
pytorch-1.13.0       | 138.2 MB  | ########8  |  88%
pytorch-1.13.0       | 138.2 MB  | ########9  |  89%
pytorch-1.13.0       | 138.2 MB  | #########  |  91%
pytorch-1.13.0       | 138.2 MB  | #########2 |  92%
pytorch-1.13.0       | 138.2 MB  | #########3 |  93%
pytorch-1.13.0       | 138.2 MB  | #########4 |  95%
pytorch-1.13.0       | 138.2 MB  | #########6 |  96%
pytorch-1.13.0       | 138.2 MB  | #########7 |  97%
pytorch-1.13.0       | 138.2 MB  | #########8 |  99%
pytorch-1.13.0       | 138.2 MB  | ########## | 100%

libuv-1.44.2         | 362 KB    |            |   0%
libuv-1.44.2         | 362 KB    | 4          |   4%
libuv-1.44.2         | 362 KB    | ########## | 100%

pytorch-mutex-1.0    | 3 KB      |            |   0%
```

```
pytorch-mutex-1.0     | 3 KB      | ########## | 100%
pytorch-mutex-1.0     | 3 KB      | ########## | 100%

torchaudio-0.13.0     | 4.5 MB    |            |   0%
torchaudio-0.13.0     | 4.5 MB    | ##4        |  24%
torchaudio-0.13.0     | 4.5 MB    | ######5    |  65%
torchaudio-0.13.0     | 4.5 MB    | ########## | 100%
torchaudio-0.13.0     | 4.5 MB    | ########## | 100%

conda-22.9.0          | 985 KB    |            |   0%
conda-22.9.0          | 985 KB    | ########## | 100%
conda-22.9.0          | 985 KB    | ########## | 100%

torchvision-0.14.0    | 6.3 MB    |            |   0%
torchvision-0.14.0    | 6.3 MB    | 6          |   6%
torchvision-0.14.0    | 6.3 MB    | ##6        |  26%
torchvision-0.14.0    | 6.3 MB    | ####7      |  47%
torchvision-0.14.0    | 6.3 MB    | ######4    |  64%
torchvision-0.14.0    | 6.3 MB    | ########5  |  86%
torchvision-0.14.0    | 6.3 MB    | ########## | 100%
Preparing transaction: ...working... done
Verifying transaction: ...working... done
Executing transaction: ...working... done
Retrieving notices: ...working... done
```

Wprowadzenie

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

In [2]:

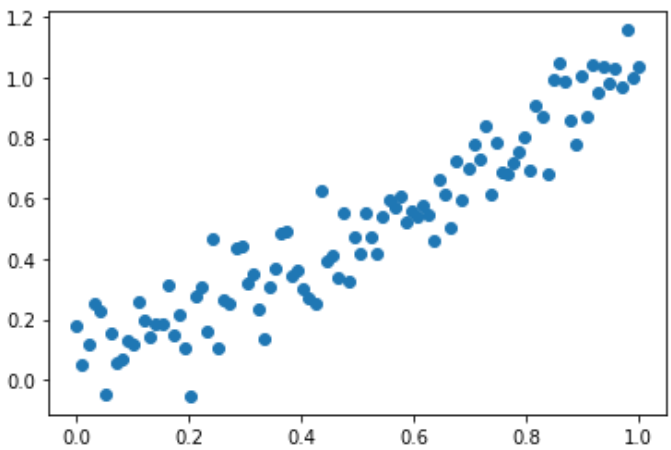
```
from typing import Tuple, Dict

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

In [3]:

```
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)
```

Out[3]:

```
<matplotlib.collections.PathCollection at 0x29503600a30>
```



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

Zadanie 1 (0.5 punkt)

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

In [7]:

```
def mse(y: np.ndarray, y_hat: np.ndarray) -> float:
    return np.mean((y - y_hat) ** 2)
```

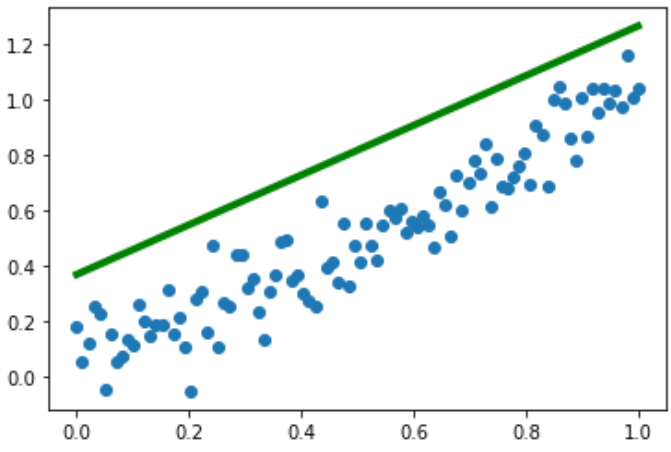
In [14]:

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

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

```
MSE: 0.107
```

Out[14]:

```
[<matplotlib.lines.Line2D at 0x295054cf2e0>]
```



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

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

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

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

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

Przyglądając się temu równaniu widzimy, że:

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

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

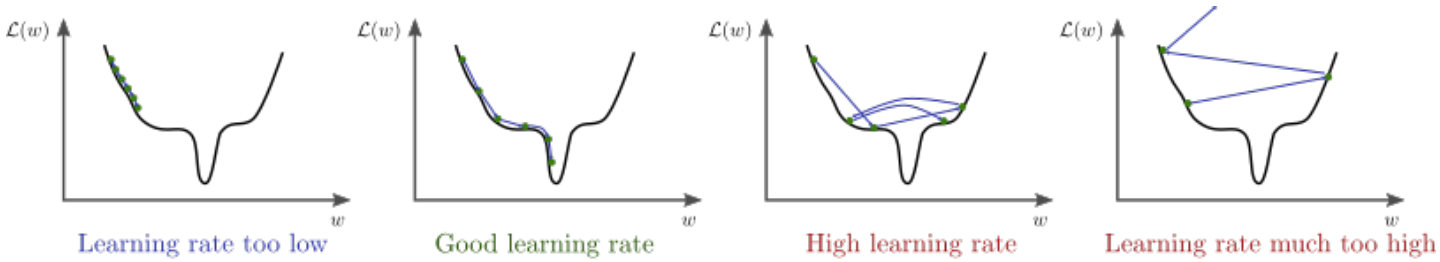
Stosując powyższe do optymalizacji, mamy:

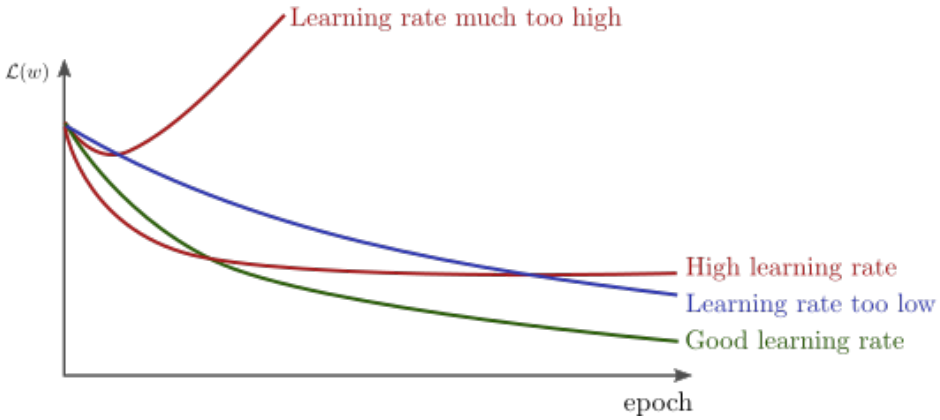
$$x_{t+1} = x_t - \alpha * \frac{f(x)}{dx}$$

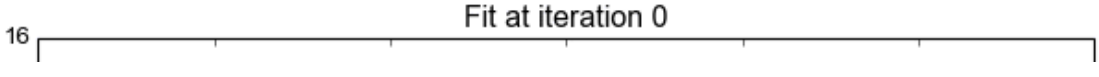
α to niewielka wartość (rzędu zwykle 10^{-5} - 10^{-2}), wprowadzona, aby trzymać się założenia o małej zmianie parametrów (ϵ). Nazywa się ją stałą uczącą (learning rate) i jest zwykle najważniejszym hiperparametrem podczas nauki sieci.

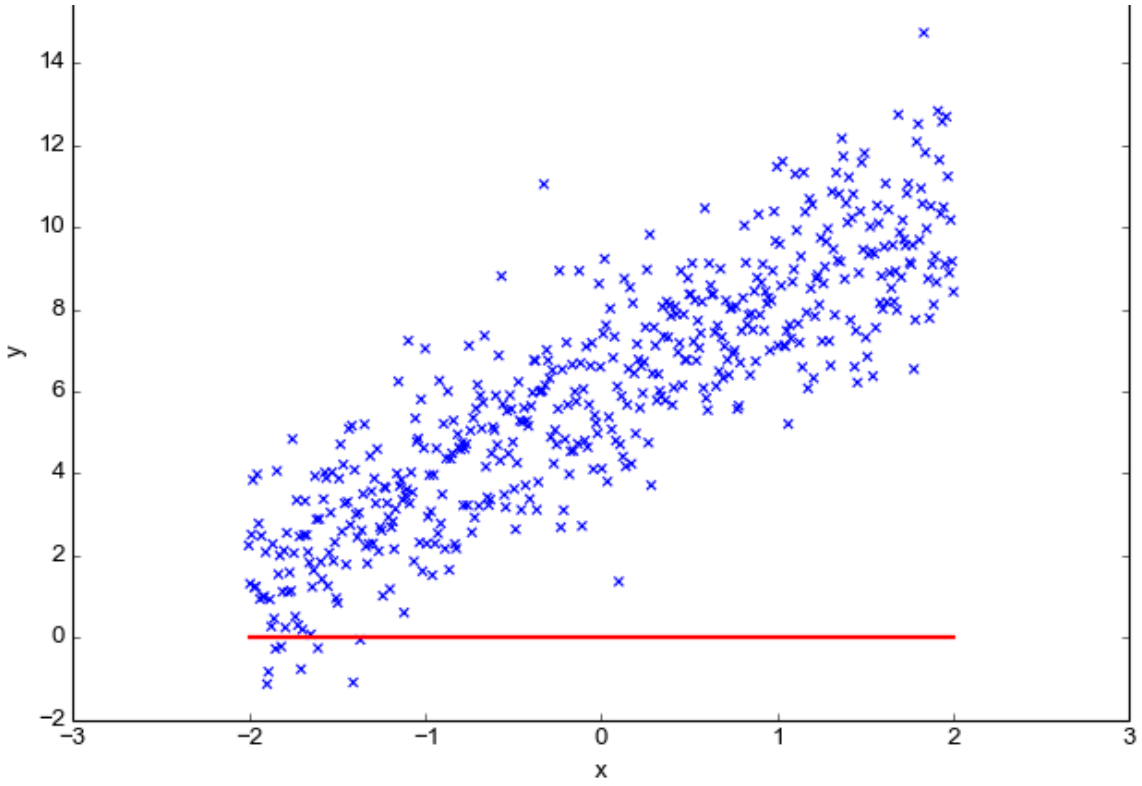
Metoda ta zakłada, że używamy całego zbioru danych do aktualizacji parametrów w każdym kroku, co nazywa się po prostu GD (od *gradient descent*) albo *full batch GD*. Wtedy każdy krok optymalizacji nazywa się epoką (epoch).

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









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

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

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

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

Pełna aktualizacja to zatem:

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

$$
* (-x)\Bigg)
$$

$$
b' = b + \alpha
$$

$$
* \Bigg(\frac{-2}{N}
$$

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

$$
* (-1)\Bigg)
$$

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

Obliczenie pochodnych cząstkowych ze względu na każdy

Zadanie 2 (1.5 punkty)

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

In [15]:

```
def optimize(
    x: np.ndarray, y: np.ndarray, a: float, b: float, learning_rate: float = 0.1
):
    y_hat = a * x + b
    errors = y - y_hat
    a_grad = -2 * np.mean(errors * x)
    b_grad = -2 * np.mean(errors)
    a -= learning_rate * a_grad
    b -= learning_rate * b_grad
    return a, b
```

In [16]:

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

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

```
step 0 loss:  0.1065548816766127
step 100 loss:  0.01031887809653938
step 200 loss:  0.010098951430812353
step 300 loss:  0.01008417312231013
step 400 loss:  0.010083180071278938
step 500 loss:  0.010083113341696835
step 600 loss:  0.010083108857700555
step 700 loss:  0.010083108556391574
step 800 loss:  0.010083108536144657
step 900 loss:  0.01008310853478413
final loss: 0.010083108534692886
```

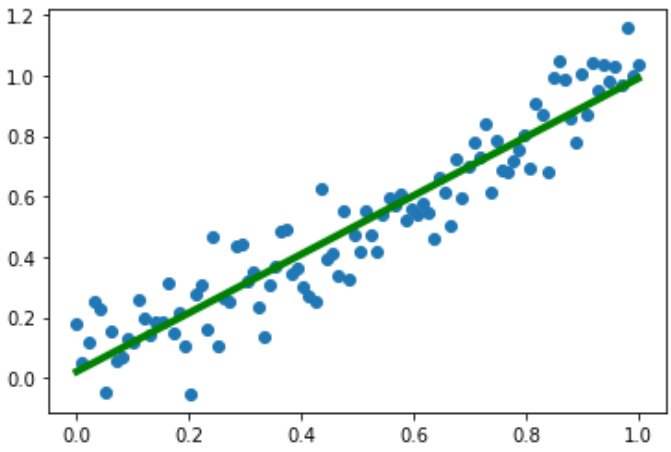
In [17]:

```
plt.scatter(x, y)
```

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

Out[17]:

```
[<matplotlib.lines.Line2D at 0x295055326a0>]
```



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 [18]:

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

In [19]:

```
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([1.6359, 1.2581, 1.1945, 1.1868, 1.4237, 1.0729, 1.6517, 1.5732, 1.7320,
        1.0632])
tensor([0.6359, 0.2581, 0.1945, 0.1868, 0.4237, 0.0729, 0.6517, 0.5732, 0.7320,
        0.0632])
tensor(3.7919)
```

In [20]:

```
# 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 [21]:

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

Out[21]:

```
(tensor([0.4552], requires_grad=True), tensor([0.4250], requires_grad=True))
```

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

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

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

In [22]:

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

Out[22]:

```
tensor(0.0541, 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 [23]:

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

In [26]:

```
print(a.grad)
```

```
tensor([0.0590])
```

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 [34]:

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

Out[34]:

```
tensor([0.5311])
```

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

In [35]:

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

    # compute gradients
    loss.backward()

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

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

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

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

```
step 0 loss:  tensor(0.0541, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 100 loss:  tensor(0.0115, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 200 loss:  tensor(0.0102, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 300 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 400 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 500 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 600 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 700 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 800 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 900 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
final loss: tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
```

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

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

In [36]:

```
# initialization
learning_rate = 0.1
a = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)
optimizer = torch.optim.SGD([a, b], lr=learning_rate)
best_loss = float("inf")

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

    # loss calculation
    loss = mse(y, y_hat)

    # backpropagation
    loss.backward()

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

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

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

```
step 0 loss: 0.4777
step 100 loss: 0.0117
step 200 loss: 0.0102
step 300 loss: 0.0101
step 400 loss: 0.0101
step 500 loss: 0.0101
step 600 loss: 0.0101
step 700 loss: 0.0101
step 800 loss: 0.0101
step 900 loss: 0.0101
final loss: tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
```

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

Funkcją kosztu zwykle jest entropia krzyżowa (cross-entropy), stosowana też w klasycznej regresji logistycznej. Co ważne, sieci neuronowe, nawet tak proste, uczą się szybciej i stabilniej, gdy dane na wejściu (a przynajmniej zmienne numeryczne) są ustandaryzowane (standardized). Operacja ta polega na odjęciu średniej i podzieleniu przez odchylenie standardowe (tzw. *Z-score transformation*).

Uwaga - PyTorch wymaga tensora klas będącego liczbami zmiennoprzecinkowymi!

Zbiór danych

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

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

In [38]:

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

```
UsageError: Line magic function `%wget` not found.
```

In [39]:

```
import pandas as pd

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

"""
age: continuous.
workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
fnlwgt: continuous.
education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool.
education-num: continuous.
marital-status: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse.
occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
sex: Female, Male.
capital-gain: continuous.
capital-loss: continuous.
hours-per-week: continuous.
native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands.
"""

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

Out[39]:

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

In [40]:

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

```
df['education'].replace('Assoc-acdm', 'CommunityCollege',inplace=True)
df['education'].replace('Assoc-voc', 'CommunityCollege',inplace=True)
df['education'].replace('Bachelors', 'Bachelors',inplace=True)
df['education'].replace('Masters', 'Masters',inplace=True)
df['education'].replace('Prof-school', 'Masters',inplace=True)
df['education'].replace('Doctorate', 'Doctorate',inplace=True)

df['marital-status'].replace('Never-married', 'NotMarried',inplace=True)
df['marital-status'].replace(['Married-AF-spouse'], 'Married',inplace=True)
df['marital-status'].replace(['Married-civ-spouse'], 'Married',inplace=True)
df['marital-status'].replace(['Married-spouse-absent'], 'NotMarried',inplace=True)
df['marital-status'].replace(['Separated'], 'Separated',inplace=True)
df['marital-status'].replace(['Divorced'], 'Separated',inplace=True)
df['marital-status'].replace(['Widowed'], 'Widowed',inplace=True)
```

In [41]:

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

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

train_valid_size = 0.2

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

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

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

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

categorical_encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
continuous_scaler = StandardScaler() #MinMaxScaler(feature_range=(-1, 1))

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

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

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

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

X_train.shape, y_train.shape
```

Out[41]:

```
((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()`.

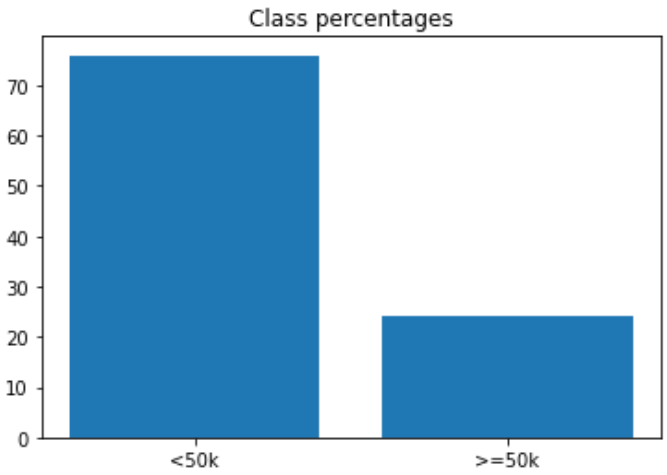
In [42]:

```
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 [43]:

```
import matplotlib.pyplot as plt

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

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



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

Zadanie 3 (1 punkt)

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

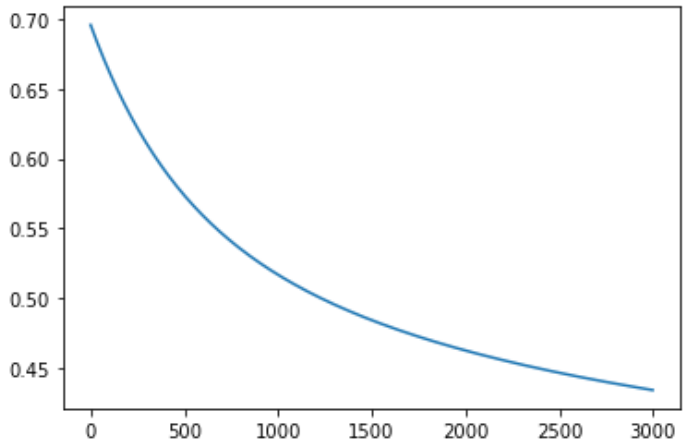
Sama sieć składa się z 2 elementów:

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

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

In [54]:

```
learning_rate = 1e-3

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

epochs = 3000
losses = []

for epoch in range(epochs):
    y_pred = activation(model(X_train))
    loss = criterion(y_pred, y_train)
    losses.append(loss.item())
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

plt.plot(losses)
plt.show()


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




```
final loss: 0.4341
```

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 [55]:

```
from sklearn.metrics import precision_recall_curve, precision_recall_fscore_support, roc_
auc_score

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

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

```
AUROC: 86.03%
```

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).

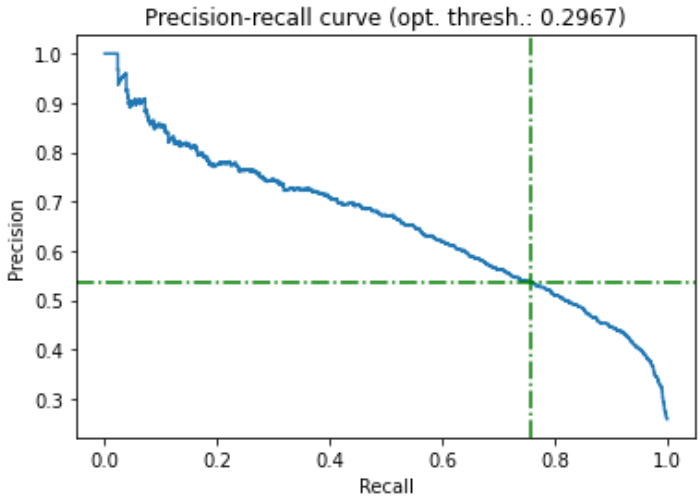
Poniżej zaimplementowano także funkcję `get_optimal_threshold()`, która sprawdza, dla którego progu uzyskujemy maksymalny F1-score, i zwraca indeks oraz wartość optymalnego progu. Przyda ci się ona w dalszej części laboratorium.

In [56]:

```
from sklearn.metrics import PrecisionRecallDisplay


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

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

    return optimal_idx, optimal_threshold


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

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

In [57]:

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

plot_precision_recall_curve(y_valid, y_pred_valid_score)
```



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

Sieci neuronowe

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

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

1. Perceptrony wielowarstwowe (*MultiLayer Perceptron*, MLP) - najbardziej podobne do powyższego opisu, niezbędne do klasyfikacji i regresji
2. Konwolucyjne (*Convolutional Neural Networks*, CNNs) - do przetwarzania danych z zależnościami przestrzennymi, np. obrazów czy dźwięku
3. Rekurencyjne (*Recurrent Neural Networks*, RNNs) - do przetwarzania danych z zależnościami sekwencyjnymi, np. szeregi czasowe, oraz kiedyś do języka naturalnego
4. Transformacyjne (*Transformers*), oparte o mechanizm atencji (*attention*) - do przetwarzania języka naturalnego (NLP), z którego wyparły RNNs, a coraz częściej także do wszelkich innych danych, np. obrazów, dźwięku
5. Grafowe (*Graph Neural Networks*, GNNS) - do przetwarzania grafów

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

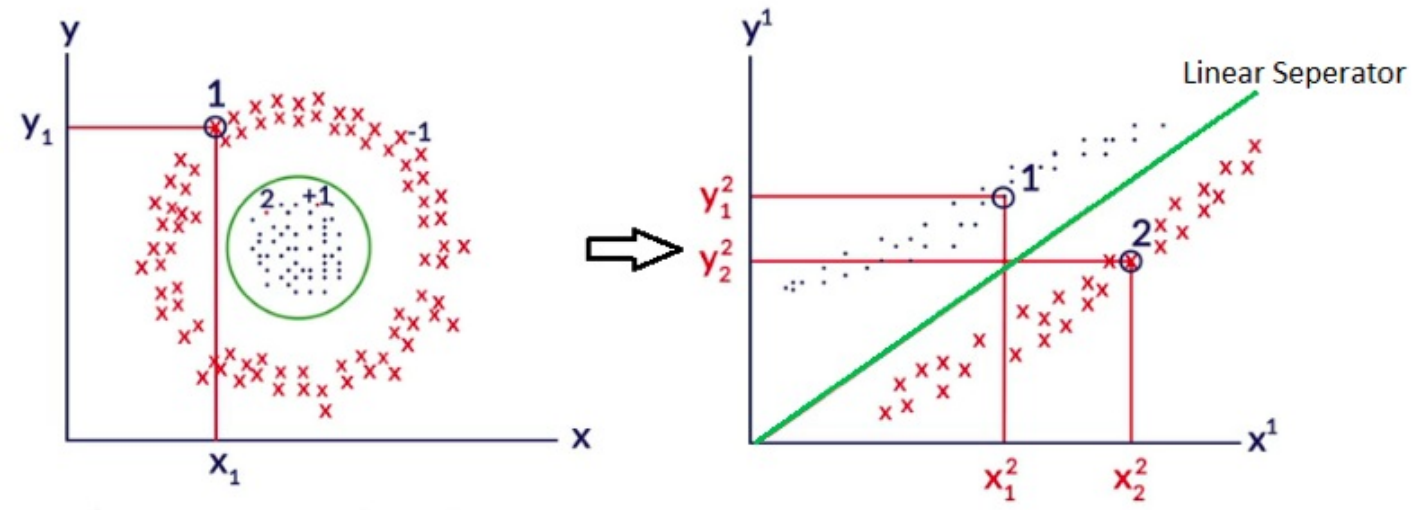
Dodatkowo, pomimo prostoty MLP są bardzo potężne - udowodniono, że perceptrony (ich powszechna nazwa) są uniwersalnym aproksymatorem, będącym w stanie przybliżyć dowolną funkcję z odpowiednio małym błędem, zakładając wystarczającą wielkość warstw sieci. Szczególne ich wersje potrafią nawet reprezentować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

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

Sieci MLP

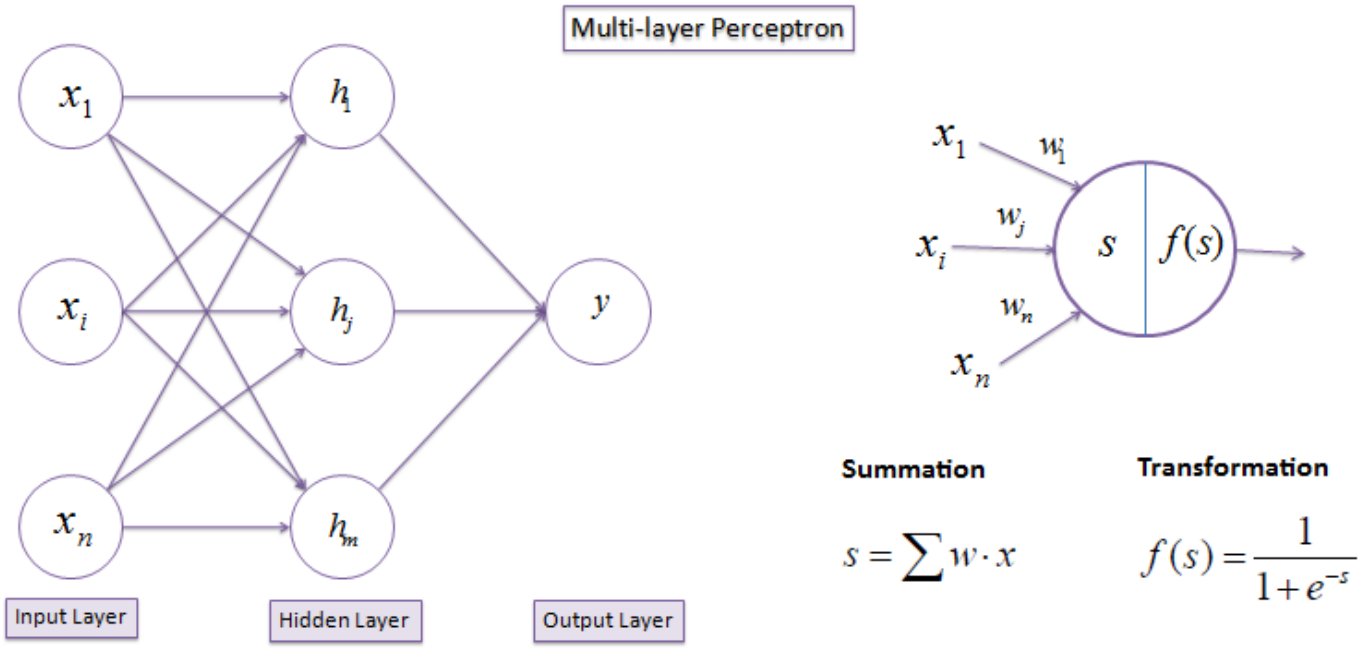
Dla przypomnienia, na wejściu mamy punkty ze zbioru treningowego, czyli d-wymiarowe wektory. W klasyfikacji chcemy znaleźć granicę decyzyjną, czyli krzywą, która oddzieli od siebie klasy. W wejściowej przestrzeni może być to trudne, bo chmury punktów z poszczególnych klas mogą być ze sobą dość pomieszane.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

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



Poszczególne neurony składają się z iloczynu skalarnego wejść z wagami neuronu, oraz nieliniowej funkcji

aktywacji. W PyTorchu są to osobne obiekty - `nn.Linear` oraz np. `nn.Sigmoid`. Funkcja aktywacji przyjmuje wynik iloczynu skalarnego i przekształca go, aby sprawdzić, jak mocno reaguje neuron na dane wejście. Musi być nieliniowa z dwóch powodów. Po pierwsze, tylko nieliniowe przekształcenia są na tyle potężne, żeby umożliwić liniową separację danych w ostatniej warstwie. Po drugie, liniowe przekształcenia zwyczajnie nie działają. Aby zrozumieć czemu, trzeba zobaczyć, co matematycznie oznacza sieć MLP.



Zapisane matematycznie MLP to: $h_1 = f_1(x)$ gdzie x to wejście f_i to funkcja aktywacji i-tej warstwy, a h_i to

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

wyjście i-tej warstwy, nazywane **ukrytą reprezentacją (hidden representation)**, lub *latent representation*. Nazwa bierze się z tego, że w środku sieci wyciągamy cechy i wzorce w danych, które nie są widoczne na pierwszy rzut oka na wejściu.

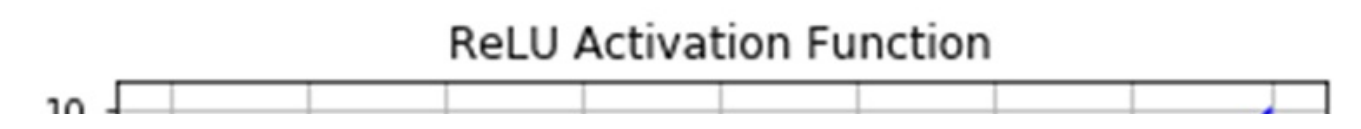
Załóżmy, że nie mamy funkcji aktywacji, czyli mamy aktywację liniową $f(x) = x$. Zobaczmy na początku sieci:

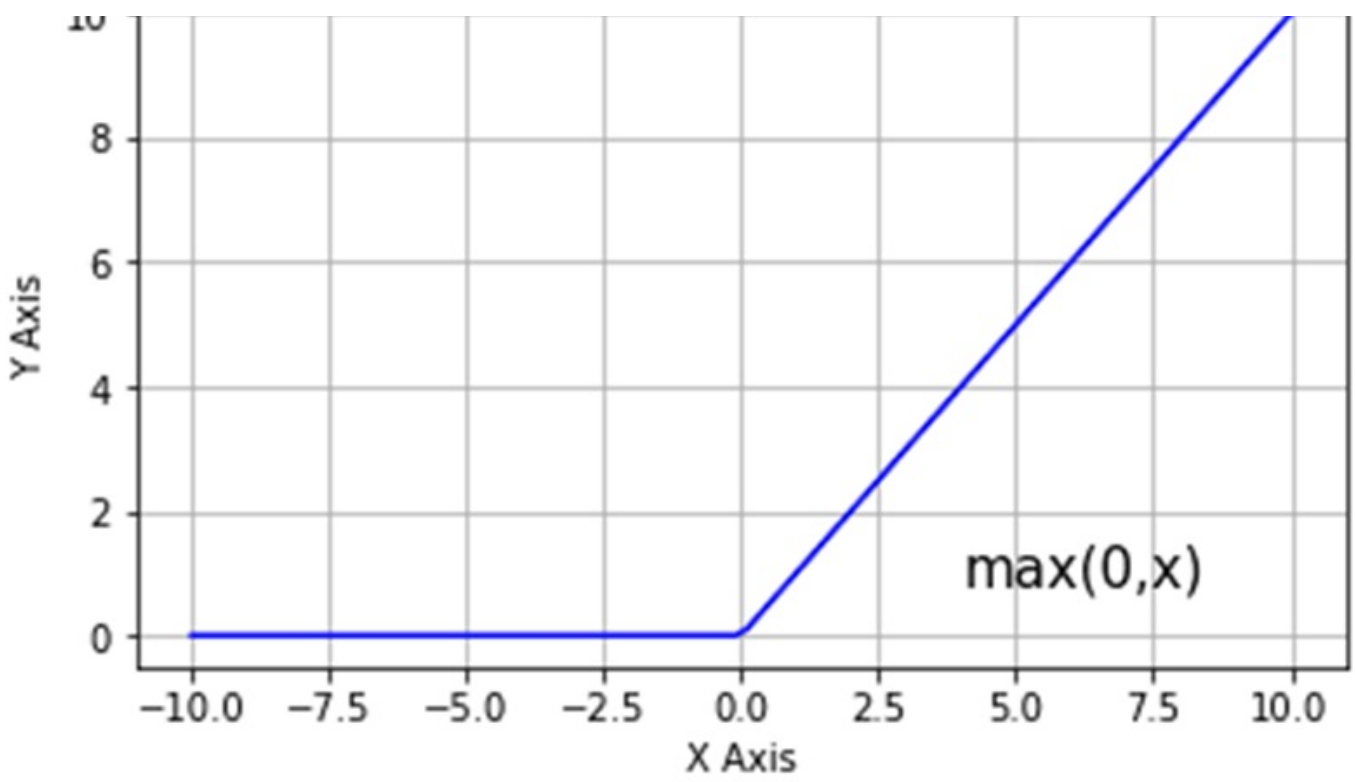
$$
\begin{aligned}
h_1 &= f_1(x) \\
&= x h_2 \\
&= f_2(f_1) \\
&= f_2(x) = x. \\
&.. h_n \\
&= f_n(f_{n-1}) \\
&= f_n(x) = x
\end{aligned}
$$

Jak widać, taka sieć niczego się nie nauczy. Wynika to z tego, że złożenie funkcji liniowych jest także funkcją liniową - patrz notatki z algebry :)

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

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





MLP w PyTorchu

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

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

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

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

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

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

Zadanie 4 (1 punkt)

Uzupełnij implementację 3-warstwowej sieci MLP. Użyj rozmiarów:

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

Użyj funkcji aktywacji ReLU.

Przydatne klasy:

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

In [60]:

```
from torch import sigmoid

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

        self.model = 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.model(x)

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

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

In [61]:

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

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

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

    optimizer.step()
    optimizer.zero_grad()

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

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

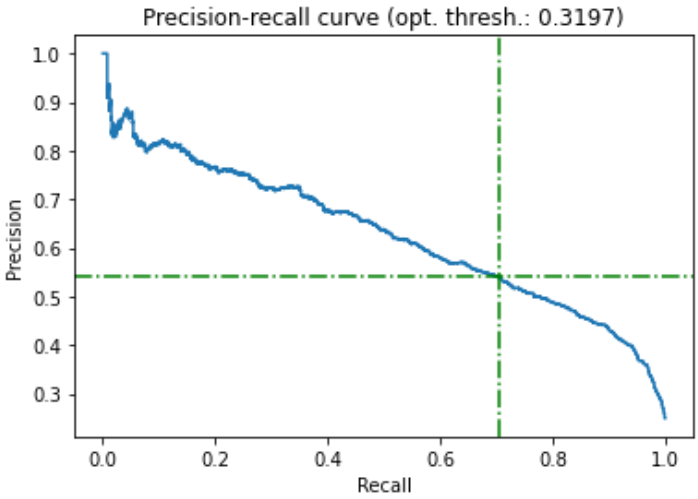
```
Epoch 0 train loss: 0.6691
Epoch 200 train loss: 0.6476
Epoch 400 train loss: 0.6292
Epoch 600 train loss: 0.6132
Epoch 800 train loss: 0.5991
```

```
Epoch 1000 train loss: 0.5865
Epoch 1200 train loss: 0.5754
Epoch 1400 train loss: 0.5654
Epoch 1600 train loss: 0.5565
Epoch 1800 train loss: 0.5485
final loss: 0.5412
```

In [62]:

```
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: 84.40%
```



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

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

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

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

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

Niektóre implementacje uwzględniają tzw. *grace period*, czyli gwarantowaną minimalną liczbę epok, przez którą będziemy trenować sieć, niezależnie od wybranej cierpliwości.

Dodatkowo ryzyko przeuczenia można zmniejszyć, używając mniejszej stałej uczącej.

Zadanie 5 (1 punkt)

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

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

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

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

In [63]:

```
from typing import Optional

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

def evaluate_model(
    model: nn.Module,
    X: torch.Tensor,
    y: torch.Tensor,
    loss_fn: nn.Module,
    threshold: Optional[float]= None
) -> Dict[str, float]:
    # implement me!
    model.eval()
    with torch.no_grad():
        y_pred = model(X)
        loss = loss_fn(y_pred, y)
        y_pred_score = sigmoid(y_pred)
        auroc = roc_auc_score(y, y_pred_score)
        optimal_idx, optimal_threshold = get_optimal_threshold(*precision_recall_curve(y
, y_pred_score))
        y_pred_hard = (y_pred_score >= optimal_threshold).float()
        precision = precision_score(y, y_pred_hard)
        recall = recall_score(y, y_pred_hard)
        f1 = f1_score(y, y_pred_hard)


    auroc = roc_auc_score(y, y_pred_score)

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

    y_pred = (y_pred_score >= threshold).float()

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

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

```
    return results
```

Zadanie 6 (1 punkt)

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

In [64]:

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

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

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

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

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

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

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

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

Na razie użyjemy jednak minibatch SGD.

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

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

In [65]:

```
from torch.utils.data import Dataset

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

```
        self.data = data
        self.y = y

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

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

Zadanie 7 (2 punkty)

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

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

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

In [66]:

```
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 [69]:

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

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

steps_without_improvement = 0

best_val_loss = np.inf
best_model = None
best_threshold = None

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

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

```
        optimizer.zero_grad()


    # model evaluation, early stopping
    model.eval()
    val_results = evaluate_model(model, X_valid, y_valid, loss_fn)
    val_loss = val_results["loss"]

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_model = deepcopy(model)
        best_threshold = val_results["optimal_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_results['loss
']}")
```

```
Epoch 0 train loss: 0.6694, eval loss 0.664940357208252
Epoch 1 train loss: 0.6493, eval loss 0.6505984663963318
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
Epoch 2 train loss: 0.6363, eval loss 0.6379115581512451
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
Epoch 3 train loss: 0.6220, eval loss 0.626559853553772
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
Epoch 4 train loss: 0.6130, eval loss 0.6164289116859436
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
Epoch 5 train loss: 0.6074, eval loss 0.6073062419891357
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
Epoch 6 train loss: 0.5952, eval loss 0.5990767478942871
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
C:\Users\Wojtek\AppData\Local\Temp\ipykernel_16912\2237191592.py:9: RuntimeWarning: inval
```

```
id value encountered in true_divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
```

```
Epoch 7 train loss: 0.5842, eval loss 0.5916579961776733
Epoch 8 train loss: 0.5791, eval loss 0.5848808884620667
Epoch 9 train loss: 0.5757, eval loss 0.5787349343299866
Epoch 10 train loss: 0.5645, eval loss 0.5731366276741028
Epoch 11 train loss: 0.5559, eval loss 0.567969024181366
Epoch 12 train loss: 0.5582, eval loss 0.5631903409957886
Epoch 13 train loss: 0.5540, eval loss 0.5587365627288818
Epoch 14 train loss: 0.5487, eval loss 0.5545583963394165
Epoch 15 train loss: 0.5494, eval loss 0.5506676435470581
Epoch 16 train loss: 0.5506, eval loss 0.5469855666160583
Epoch 17 train loss: 0.5321, eval loss 0.5434700846672058
Epoch 18 train loss: 0.5333, eval loss 0.5400855541229248
Epoch 19 train loss: 0.5379, eval loss 0.5368154048919678
Epoch 20 train loss: 0.5261, eval loss 0.5336123108863831
Epoch 21 train loss: 0.5315, eval loss 0.5304904580116272
Epoch 22 train loss: 0.5345, eval loss 0.5273909568786621
Epoch 23 train loss: 0.5131, eval loss 0.5243391990661621
Epoch 24 train loss: 0.5203, eval loss 0.521281361579895
Epoch 25 train loss: 0.5054, eval loss 0.5182325839996338
Epoch 26 train loss: 0.5145, eval loss 0.5152115821838379
Epoch 27 train loss: 0.5045, eval loss 0.5121763348579407
Epoch 28 train loss: 0.4923, eval loss 0.5091046690940857
Epoch 29 train loss: 0.4952, eval loss 0.5060411691665649
Epoch 30 train loss: 0.5084, eval loss 0.5029376149177551
Epoch 31 train loss: 0.4920, eval loss 0.49982497096061707
Epoch 32 train loss: 0.5037, eval loss 0.49665841460227966
Epoch 33 train loss: 0.4907, eval loss 0.4935014545917511
Epoch 34 train loss: 0.4946, eval loss 0.49028724431991577
Epoch 35 train loss: 0.4953, eval loss 0.4870961606502533
Epoch 36 train loss: 0.4827, eval loss 0.483864963054657
Epoch 37 train loss: 0.4772, eval loss 0.48063069581985474
Epoch 38 train loss: 0.4804, eval loss 0.47734329104423523
Epoch 39 train loss: 0.4767, eval loss 0.4740937352180481
Epoch 40 train loss: 0.4688, eval loss 0.4708177447319031
Epoch 41 train loss: 0.4804, eval loss 0.4675653874874115
Epoch 42 train loss: 0.4767, eval loss 0.46430519223213196
Epoch 43 train loss: 0.4510, eval loss 0.46105465292930603
Epoch 44 train loss: 0.4505, eval loss 0.4578195810317993
Epoch 45 train loss: 0.4732, eval loss 0.4546118676662445
Epoch 46 train loss: 0.4670, eval loss 0.45143288373947144
Epoch 47 train loss: 0.4720, eval loss 0.448257178068161
Epoch 48 train loss: 0.4388, eval loss 0.44510215520858765
Epoch 49 train loss: 0.4543, eval loss 0.4420180916786194
Epoch 50 train loss: 0.4491, eval loss 0.43899479508399963
Epoch 51 train loss: 0.4573, eval loss 0.43598490953445435
Epoch 52 train loss: 0.4633, eval loss 0.4330379068851471
Epoch 53 train loss: 0.4531, eval loss 0.4301515221595764
Epoch 54 train loss: 0.4231, eval loss 0.42729127407073975
Epoch 55 train loss: 0.4151, eval loss 0.42450931668281555
Epoch 56 train loss: 0.4404, eval loss 0.4217766523361206
Epoch 57 train loss: 0.4446, eval loss 0.4190785288881073
Epoch 58 train loss: 0.4477, eval loss 0.4164709150791168
Epoch 59 train loss: 0.4129, eval loss 0.4139097332954407
Epoch 60 train loss: 0.4190, eval loss 0.4114264249801636
Epoch 61 train loss: 0.4497, eval loss 0.4089726507663727
Epoch 62 train loss: 0.4159, eval loss 0.40660202503204346
Epoch 63 train loss: 0.4372, eval loss 0.4042811691761017
Epoch 64 train loss: 0.4402, eval loss 0.40202251076698303
Epoch 65 train loss: 0.4022, eval loss 0.39984551072120667
Epoch 66 train loss: 0.4311, eval loss 0.3977136015892029
Epoch 67 train loss: 0.4255, eval loss 0.39564046263694763
Epoch 68 train loss: 0.4027, eval loss 0.39366284012794495
Epoch 69 train loss: 0.4477, eval loss 0.391778826713562
Epoch 70 train loss: 0.4045, eval loss 0.3899168372154236
Epoch 71 train loss: 0.4045, eval loss 0.38812291622161865
Epoch 72 train loss: 0.4088, eval loss 0.3863869905471802
Epoch 73 train loss: 0.4297, eval loss 0.38471952080726624
Epoch 74 train loss: 0.4245, eval loss 0.38314902782440186
Epoch 75 train loss: 0.4199, eval loss 0.3815879821777344
Epoch 76 train loss: 0.4165, eval loss 0.38011443614959717
```

Epoch 77 train loss: 0.4204, eval loss 0.37872564792633057
Epoch 78 train loss: 0.4136, eval loss 0.37734806537628174
Epoch 79 train loss: 0.4177, eval loss 0.37606778740882874
Epoch 80 train loss: 0.4048, eval loss 0.37481144070625305
Epoch 81 train loss: 0.4126, eval loss 0.3736100196838379
Epoch 82 train loss: 0.4007, eval loss 0.37247395515441895
Epoch 83 train loss: 0.3966, eval loss 0.37140074372291565
Epoch 84 train loss: 0.3975, eval loss 0.37037134170532227
Epoch 85 train loss: 0.3978, eval loss 0.3693805932998657
Epoch 86 train loss: 0.4124, eval loss 0.36842623353004456
Epoch 87 train loss: 0.3948, eval loss 0.3675374686717987
Epoch 88 train loss: 0.4112, eval loss 0.36664825677871704
Epoch 89 train loss: 0.4117, eval loss 0.36585044860839844
Epoch 90 train loss: 0.4099, eval loss 0.3650493025779724
Epoch 91 train loss: 0.4092, eval loss 0.364305704832077
Epoch 92 train loss: 0.4069, eval loss 0.3635961413383484
Epoch 93 train loss: 0.3836, eval loss 0.3628889322280884
Epoch 94 train loss: 0.3728, eval loss 0.3622134029865265
Epoch 95 train loss: 0.4062, eval loss 0.36156585812568665
Epoch 96 train loss: 0.3909, eval loss 0.3609558641910553
Epoch 97 train loss: 0.3751, eval loss 0.360365629196167
Epoch 98 train loss: 0.3755, eval loss 0.35980328917503357
Epoch 99 train loss: 0.4130, eval loss 0.35925862193107605
Epoch 100 train loss: 0.4294, eval loss 0.3587338626384735
Epoch 101 train loss: 0.4167, eval loss 0.35823968052864075
Epoch 102 train loss: 0.3777, eval loss 0.3577571511268616
Epoch 103 train loss: 0.3994, eval loss 0.3572736382484436
Epoch 104 train loss: 0.3956, eval loss 0.3568008542060852
Epoch 105 train loss: 0.4015, eval loss 0.3563409447669983
Epoch 106 train loss: 0.3873, eval loss 0.35590603947639465
Epoch 107 train loss: 0.3915, eval loss 0.3554862141609192
Epoch 108 train loss: 0.3853, eval loss 0.3550875782966614
Epoch 109 train loss: 0.3679, eval loss 0.35468828678131104
Epoch 110 train loss: 0.3989, eval loss 0.3543097674846649
Epoch 111 train loss: 0.3779, eval loss 0.3539489805698395
Epoch 112 train loss: 0.3892, eval loss 0.3535953462123871
Epoch 113 train loss: 0.3710, eval loss 0.35325542092323303
Epoch 114 train loss: 0.4222, eval loss 0.35290905833244324
Epoch 115 train loss: 0.3903, eval loss 0.35256028175354004
Epoch 116 train loss: 0.3676, eval loss 0.3522440195083618
Epoch 117 train loss: 0.4105, eval loss 0.351917028427124
Epoch 118 train loss: 0.4038, eval loss 0.3516141176223755
Epoch 119 train loss: 0.4258, eval loss 0.35128968954086304
Epoch 120 train loss: 0.3998, eval loss 0.35098424553871155
Epoch 121 train loss: 0.3933, eval loss 0.35068774223327637
Epoch 122 train loss: 0.3732, eval loss 0.3503929674625397
Epoch 123 train loss: 0.4024, eval loss 0.3501039445400238
Epoch 124 train loss: 0.3857, eval loss 0.3498329222202301
Epoch 125 train loss: 0.3958, eval loss 0.34956443309783936
Epoch 126 train loss: 0.3990, eval loss 0.34929022192955017
Epoch 127 train loss: 0.3821, eval loss 0.34902700781822205
Epoch 128 train loss: 0.3997, eval loss 0.34876593947410583
Epoch 129 train loss: 0.3844, eval loss 0.34851202368736267
Epoch 130 train loss: 0.3982, eval loss 0.3482547402381897
Epoch 131 train loss: 0.3716, eval loss 0.34799641370773315
Epoch 132 train loss: 0.3747, eval loss 0.347747802734375
Epoch 133 train loss: 0.3547, eval loss 0.34752142429351807
Epoch 134 train loss: 0.4109, eval loss 0.3472699224948883
Epoch 135 train loss: 0.3806, eval loss 0.34703272581100464
Epoch 136 train loss: 0.3692, eval loss 0.3468078374862671
Epoch 137 train loss: 0.3896, eval loss 0.3465659022331238
Epoch 138 train loss: 0.4135, eval loss 0.34633466601371765
Epoch 139 train loss: 0.3750, eval loss 0.3461158275604248
Epoch 140 train loss: 0.3898, eval loss 0.3458978235721588
Epoch 141 train loss: 0.3830, eval loss 0.3456861972808838
Epoch 142 train loss: 0.3827, eval loss 0.34546151757240295
Epoch 143 train loss: 0.3775, eval loss 0.345233678817749
Epoch 144 train loss: 0.3899, eval loss 0.3450206220149994
Epoch 145 train loss: 0.3883, eval loss 0.3447941541671753
Epoch 146 train loss: 0.4016, eval loss 0.3445945084095001
Epoch 147 train loss: 0.3857, eval loss 0.34438037872314453
Epoch 148 train loss: 0.3504, eval loss 0.34417515993118286

Epoch 149 train loss: 0.3625, eval loss 0.34398579597473145
Epoch 150 train loss: 0.3709, eval loss 0.3437887728214264
Epoch 151 train loss: 0.3741, eval loss 0.3435795307159424
Epoch 152 train loss: 0.3738, eval loss 0.34338271617889404
Epoch 153 train loss: 0.3854, eval loss 0.3431810140609741
Epoch 154 train loss: 0.3818, eval loss 0.34300005435943604
Epoch 155 train loss: 0.3944, eval loss 0.342800110578537
Epoch 156 train loss: 0.3559, eval loss 0.342597097158432
Epoch 157 train loss: 0.3994, eval loss 0.3424147963523865
Epoch 158 train loss: 0.4160, eval loss 0.34223049879074097
Epoch 159 train loss: 0.4268, eval loss 0.34203970432281494
Epoch 160 train loss: 0.3988, eval loss 0.3418656885623932
Epoch 161 train loss: 0.3608, eval loss 0.3416884243488312
Epoch 162 train loss: 0.3651, eval loss 0.3415330648422241
Epoch 163 train loss: 0.3466, eval loss 0.3413618803024292
Epoch 164 train loss: 0.4202, eval loss 0.3411884307861328
Epoch 165 train loss: 0.3660, eval loss 0.3410263657569885
Epoch 166 train loss: 0.3601, eval loss 0.34085556864738464
Epoch 167 train loss: 0.3899, eval loss 0.3406793773174286
Epoch 168 train loss: 0.4083, eval loss 0.34051430225372314
Epoch 169 train loss: 0.4133, eval loss 0.34036222100257874
Epoch 170 train loss: 0.3613, eval loss 0.34020736813545227
Epoch 171 train loss: 0.3751, eval loss 0.3400559723377228
Epoch 172 train loss: 0.4230, eval loss 0.33987870812416077
Epoch 173 train loss: 0.3865, eval loss 0.33972036838531494
Epoch 174 train loss: 0.3739, eval loss 0.3395599126815796
Epoch 175 train loss: 0.3992, eval loss 0.33941197395324707
Epoch 176 train loss: 0.3804, eval loss 0.3392731845378876
Epoch 177 train loss: 0.3988, eval loss 0.3391100764274597
Epoch 178 train loss: 0.4055, eval loss 0.3389631509780884
Epoch 179 train loss: 0.3777, eval loss 0.33879366517066956
Epoch 180 train loss: 0.3925, eval loss 0.33863767981529236
Epoch 181 train loss: 0.3531, eval loss 0.3384931981563568
Epoch 182 train loss: 0.4020, eval loss 0.3383474051952362
Epoch 183 train loss: 0.3936, eval loss 0.3382017910480499
Epoch 184 train loss: 0.3839, eval loss 0.33805811405181885
Epoch 185 train loss: 0.3741, eval loss 0.33793163299560547
Epoch 186 train loss: 0.3760, eval loss 0.3377910852432251
Epoch 187 train loss: 0.3775, eval loss 0.33764949440956116
Epoch 188 train loss: 0.3479, eval loss 0.3375139534473419
Epoch 189 train loss: 0.3895, eval loss 0.3373764157295227
Epoch 190 train loss: 0.3658, eval loss 0.33724895119667053
Epoch 191 train loss: 0.3741, eval loss 0.33711427450180054
Epoch 192 train loss: 0.3799, eval loss 0.3369746804237366
Epoch 193 train loss: 0.3798, eval loss 0.33683380484580994
Epoch 194 train loss: 0.3957, eval loss 0.33670976758003235
Epoch 195 train loss: 0.3676, eval loss 0.3365667760372162
Epoch 196 train loss: 0.3707, eval loss 0.33644041419029236
Epoch 197 train loss: 0.3741, eval loss 0.33630549907684326
Epoch 198 train loss: 0.3705, eval loss 0.33618247509002686
Epoch 199 train loss: 0.3951, eval loss 0.3360620439052582
Epoch 200 train loss: 0.3534, eval loss 0.3359230160713196
Epoch 201 train loss: 0.3450, eval loss 0.33578982949256897
Epoch 202 train loss: 0.3474, eval loss 0.3356732130050659
Epoch 203 train loss: 0.3819, eval loss 0.3355666399002075
Epoch 204 train loss: 0.3994, eval loss 0.3354455530643463
Epoch 205 train loss: 0.3889, eval loss 0.3353370428085327
Epoch 206 train loss: 0.3647, eval loss 0.3352084159851074
Epoch 207 train loss: 0.3664, eval loss 0.33507975935935974
Epoch 208 train loss: 0.3924, eval loss 0.33495089411735535
Epoch 209 train loss: 0.3419, eval loss 0.334837943315506
Epoch 210 train loss: 0.3733, eval loss 0.3347371816635132
Epoch 211 train loss: 0.4235, eval loss 0.3346240222454071
Epoch 212 train loss: 0.3600, eval loss 0.3345092833042145
Epoch 213 train loss: 0.3619, eval loss 0.334389328956604
Epoch 214 train loss: 0.3833, eval loss 0.3342684507369995
Epoch 215 train loss: 0.3736, eval loss 0.334170401096344
Epoch 216 train loss: 0.3995, eval loss 0.3340386748313904
Epoch 217 train loss: 0.3559, eval loss 0.3339283764362335
Epoch 218 train loss: 0.3602, eval loss 0.33381858468055725
Epoch 219 train loss: 0.3779, eval loss 0.3337070643901825
Epoch 220 train loss: 0.3720, eval loss 0.3336089253425598

Epoch 221 train loss: 0.3462, eval loss 0.3334963619709015
Epoch 222 train loss: 0.3865, eval loss 0.3333815932273865
Epoch 223 train loss: 0.3417, eval loss 0.333270788192749
Epoch 224 train loss: 0.3870, eval loss 0.33315491676330566
Epoch 225 train loss: 0.3629, eval loss 0.33303940296173096
Epoch 226 train loss: 0.3899, eval loss 0.3329316973686218
Epoch 227 train loss: 0.3743, eval loss 0.33282479643821716
Epoch 228 train loss: 0.3706, eval loss 0.3327169418334961
Epoch 229 train loss: 0.3483, eval loss 0.3326067328453064
Epoch 230 train loss: 0.3903, eval loss 0.33251726627349854
Epoch 231 train loss: 0.3623, eval loss 0.332420289516449
Epoch 232 train loss: 0.3859, eval loss 0.33231741189956665
Epoch 233 train loss: 0.4100, eval loss 0.33221718668937683
Epoch 234 train loss: 0.3805, eval loss 0.33212390542030334
Epoch 235 train loss: 0.3618, eval loss 0.3320276737213135
Epoch 236 train loss: 0.3603, eval loss 0.3319481313228607
Epoch 237 train loss: 0.3671, eval loss 0.33183130621910095
Epoch 238 train loss: 0.3702, eval loss 0.3317405581474304
Epoch 239 train loss: 0.3798, eval loss 0.3316442370414734
Epoch 240 train loss: 0.3547, eval loss 0.3315690755844116
Epoch 241 train loss: 0.3795, eval loss 0.3314697742462158
Epoch 242 train loss: 0.3429, eval loss 0.33138546347618103
Epoch 243 train loss: 0.3822, eval loss 0.3312944173812866
Epoch 244 train loss: 0.3787, eval loss 0.3312092423439026
Epoch 245 train loss: 0.3957, eval loss 0.3311122953891754
Epoch 246 train loss: 0.3800, eval loss 0.33102381229400635
Epoch 247 train loss: 0.4180, eval loss 0.3309314250946045
Epoch 248 train loss: 0.3626, eval loss 0.33084428310394287
Epoch 249 train loss: 0.3540, eval loss 0.3307413160800934
Epoch 250 train loss: 0.3759, eval loss 0.33066070079803467
Epoch 251 train loss: 0.3702, eval loss 0.33055904507637024
Epoch 252 train loss: 0.3886, eval loss 0.33048123121261597
Epoch 253 train loss: 0.3964, eval loss 0.3303948640823364
Epoch 254 train loss: 0.3671, eval loss 0.3303113579750061
Epoch 255 train loss: 0.3878, eval loss 0.33023083209991455
Epoch 256 train loss: 0.3562, eval loss 0.33016014099121094
Epoch 257 train loss: 0.3969, eval loss 0.33006227016448975
Epoch 258 train loss: 0.3826, eval loss 0.3299647271633148
Epoch 259 train loss: 0.3640, eval loss 0.329883337020874
Epoch 260 train loss: 0.3707, eval loss 0.3297949731349945
Epoch 261 train loss: 0.3494, eval loss 0.3297130763530731
Epoch 262 train loss: 0.3795, eval loss 0.329633504152298
Epoch 263 train loss: 0.3722, eval loss 0.329550176858902
Epoch 264 train loss: 0.3792, eval loss 0.3294758200645447
Epoch 265 train loss: 0.3082, eval loss 0.329389750957489
Epoch 266 train loss: 0.3732, eval loss 0.329303503036499
Epoch 267 train loss: 0.3819, eval loss 0.3292332887649536
Epoch 268 train loss: 0.3548, eval loss 0.3291490972042084
Epoch 269 train loss: 0.3874, eval loss 0.32906854152679443
Epoch 270 train loss: 0.3531, eval loss 0.32899942994117737
Epoch 271 train loss: 0.3615, eval loss 0.3289247751235962
Epoch 272 train loss: 0.3574, eval loss 0.3288585841655731
Epoch 273 train loss: 0.3507, eval loss 0.3287869393825531
Epoch 274 train loss: 0.4019, eval loss 0.32870930433273315
Epoch 275 train loss: 0.3517, eval loss 0.3286377787590027
Epoch 276 train loss: 0.3892, eval loss 0.32856690883636475
Epoch 277 train loss: 0.3737, eval loss 0.3284960389137268
Epoch 278 train loss: 0.3686, eval loss 0.32842835783958435
Epoch 279 train loss: 0.3699, eval loss 0.32835525274276733
Epoch 280 train loss: 0.3660, eval loss 0.328294962644577
Epoch 281 train loss: 0.3812, eval loss 0.3282063305377796
Epoch 282 train loss: 0.3883, eval loss 0.32813915610313416
Epoch 283 train loss: 0.3646, eval loss 0.3280741870403290
Epoch 284 train loss: 0.3613, eval loss 0.3280070424079895
Epoch 285 train loss: 0.3837, eval loss 0.32792630791664124
Epoch 286 train loss: 0.3714, eval loss 0.3278515040874481
Epoch 287 train loss: 0.3655, eval loss 0.3277795612812042
Epoch 288 train loss: 0.3626, eval loss 0.3277006447315216
Epoch 289 train loss: 0.3634, eval loss 0.32765406370162964
Epoch 290 train loss: 0.3777, eval loss 0.32757893204689026
Epoch 291 train loss: 0.3901, eval loss 0.32750219106674194
Epoch 292 train loss: 0.3492, eval loss 0.32744354009628296

```
Epoch 293 train loss: 0.3586, eval loss 0.3273720145225525
Epoch 294 train loss: 0.3494, eval loss 0.32732370495796204
Epoch 295 train loss: 0.3501, eval loss 0.3272482752799988
Epoch 296 train loss: 0.3859, eval loss 0.32718101143836975
Epoch 297 train loss: 0.3783, eval loss 0.32711970806121826
Epoch 298 train loss: 0.3579, eval loss 0.3270455300807953
Epoch 299 train loss: 0.3630, eval loss 0.32700270414352417
```

In [70]:

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

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

```
AUROC: 90.19%
F1: 68.49%
Precision: 61.85%
Recall: 76.72%
```

Wyniki wyglądają już dużo lepiej.

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

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

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

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

Zadanie 8 (1 punkt)

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

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

Zamień używany optymalizator na `AdamW`.

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

In [71]:

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

        self.mlp = nn.Sequential(
            nn.Linear(input_size, 256),
```

```
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(128, 1),
        )


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

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

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

In [73]:

```
from sklearn.utils.class_weight import compute_class_weight

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

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

early_stopping_patience = 4
```

In [74]:

```
model = NormalizingMLP(
    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(pos_weight=torch.from_numpy(weights)[1])

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

steps_without_improvement = 0

best_val_loss = np.inf
best_model = None
best_threshold = None

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

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

```
        optimizer.step()
        optimizer.zero_grad()


    # model evaluation, early stopping



    model.eval()
    valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
    if valid_metrics["loss"] < best_val_loss:
        best_val_loss = valid_metrics["loss"]
        best_model = deepcopy(model)
        best_threshold = valid_metrics["optimal_threshold"]
        steps_without_improvement = 0
    else:
        steps_without_improvement += 1
        if steps_without_improvement == early_stopping_patience:
            print("Early stopping!")
            break

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

```
Epoch 0 train loss: 0.8406, eval loss 0.8040788173675537
Epoch 1 train loss: 0.7469, eval loss 0.7321597337722778
Epoch 2 train loss: 0.7420, eval loss 0.6863046288490295
Epoch 3 train loss: 0.6645, eval loss 0.6527690291404724
Epoch 4 train loss: 0.6048, eval loss 0.6257001757621765
Epoch 5 train loss: 0.6277, eval loss 0.6058278679847717
Epoch 6 train loss: 0.6410, eval loss 0.5903467535972595
Epoch 7 train loss: 0.5975, eval loss 0.5787750482559204
Epoch 8 train loss: 0.6100, eval loss 0.5673083066940308
Epoch 9 train loss: 0.5723, eval loss 0.5599688291549683
Epoch 10 train loss: 0.6281, eval loss 0.5527128577232361
Epoch 11 train loss: 0.6142, eval loss 0.5477914214134216
Epoch 12 train loss: 0.5740, eval loss 0.5421823263168335
Epoch 13 train loss: 0.6111, eval loss 0.5386484861373901
Epoch 14 train loss: 0.6372, eval loss 0.5356246829032898
Epoch 15 train loss: 0.5839, eval loss 0.5321337580680847
Epoch 16 train loss: 0.5706, eval loss 0.5286152958869934
Epoch 17 train loss: 0.5559, eval loss 0.5258557796478271
Epoch 18 train loss: 0.5291, eval loss 0.5236740112304688
Epoch 19 train loss: 0.5936, eval loss 0.5222853422164917
Epoch 20 train loss: 0.6024, eval loss 0.5202939510345459
Epoch 21 train loss: 0.5364, eval loss 0.5183731317520142
Epoch 22 train loss: 0.5989, eval loss 0.5166724920272827
Epoch 23 train loss: 0.5723, eval loss 0.5150096416473389
Epoch 24 train loss: 0.6511, eval loss 0.5142176747322083
Epoch 25 train loss: 0.5669, eval loss 0.5126816034317017
Epoch 26 train loss: 0.6213, eval loss 0.5109835863113403
Epoch 27 train loss: 0.5559, eval loss 0.5091724395751953
Epoch 28 train loss: 0.5520, eval loss 0.5091718435287476
Epoch 29 train loss: 0.6363, eval loss 0.507881224155426
Epoch 30 train loss: 0.5675, eval loss 0.5078728199005127
Epoch 31 train loss: 0.5616, eval loss 0.5061127543449402
Epoch 32 train loss: 0.5405, eval loss 0.5049129128456116
Epoch 33 train loss: 0.5752, eval loss 0.5043767094612122
Epoch 34 train loss: 0.5797, eval loss 0.5036065578460693
Epoch 35 train loss: 0.5321, eval loss 0.5032432079315186
Epoch 36 train loss: 0.5928, eval loss 0.5019406080245972
Epoch 37 train loss: 0.5912, eval loss 0.501747190952301
Epoch 38 train loss: 0.5870, eval loss 0.5007941126823425
Epoch 39 train loss: 0.5718, eval loss 0.4995657503604889
Epoch 40 train loss: 0.5795, eval loss 0.4996967315673828
Epoch 41 train loss: 0.5707, eval loss 0.49919629096984863
Epoch 42 train loss: 0.5240, eval loss 0.4976585805416107
Epoch 43 train loss: 0.6126, eval loss 0.49802565574645996
Epoch 44 train loss: 0.5173, eval loss 0.4973483681678772
Epoch 45 train loss: 0.5940, eval loss 0.4968610405921936
Epoch 46 train loss: 0.5910, eval loss 0.4963463842868805
Epoch 47 train loss: 0.5248, eval loss 0.495995432138443
```

```
Epoch 48 train loss: 0.5847, eval loss 0.4955858290195465
Epoch 49 train loss: 0.5874, eval loss 0.49528443813323975
Epoch 50 train loss: 0.5592, eval loss 0.49478673934936523
Epoch 51 train loss: 0.5830, eval loss 0.49458199739456177
Epoch 52 train loss: 0.6339, eval loss 0.4936836063861847
Epoch 53 train loss: 0.5837, eval loss 0.49379780888557434
Epoch 54 train loss: 0.5681, eval loss 0.4931337535381317
Epoch 55 train loss: 0.5192, eval loss 0.49242597818374634
Epoch 56 train loss: 0.5673, eval loss 0.49217504262924194
Epoch 57 train loss: 0.4822, eval loss 0.49189192056655884
Epoch 58 train loss: 0.5137, eval loss 0.4918053150177002
Epoch 59 train loss: 0.4898, eval loss 0.4912315905094147
Epoch 60 train loss: 0.5170, eval loss 0.4916132688522339
Epoch 61 train loss: 0.5406, eval loss 0.49042442440986633
Epoch 62 train loss: 0.5248, eval loss 0.4910409152507782
Epoch 63 train loss: 0.6165, eval loss 0.4899245500564575
Epoch 64 train loss: 0.5138, eval loss 0.4896976351737976
Epoch 65 train loss: 0.5529, eval loss 0.4899546205997467
Epoch 66 train loss: 0.5000, eval loss 0.4888480603694916
Epoch 67 train loss: 0.5549, eval loss 0.4893757998943329
Epoch 68 train loss: 0.5635, eval loss 0.4886835515499115
Epoch 69 train loss: 0.5534, eval loss 0.4885668158531189
Epoch 70 train loss: 0.5719, eval loss 0.4881678521633148
Epoch 71 train loss: 0.5796, eval loss 0.4881201386451721
Epoch 72 train loss: 0.5750, eval loss 0.4876443147659302
Epoch 73 train loss: 0.5767, eval loss 0.4873618483543396
Epoch 74 train loss: 0.6000, eval loss 0.48728635907173157
Epoch 75 train loss: 0.5724, eval loss 0.4870719313621521
Epoch 76 train loss: 0.5509, eval loss 0.4870348572731018
Epoch 77 train loss: 0.5415, eval loss 0.486562579870224
Epoch 78 train loss: 0.5816, eval loss 0.4863249957561493
Epoch 79 train loss: 0.5436, eval loss 0.48645979166030884
Epoch 80 train loss: 0.5370, eval loss 0.4864244759082794
Epoch 81 train loss: 0.5302, eval loss 0.48572495579719543
Epoch 82 train loss: 0.5644, eval loss 0.4852605164051056
Epoch 83 train loss: 0.5285, eval loss 0.4854362905025482
Epoch 84 train loss: 0.5558, eval loss 0.4853038489818573
Epoch 85 train loss: 0.5635, eval loss 0.48496633768081665
Epoch 86 train loss: 0.6232, eval loss 0.4846828281879425
Epoch 87 train loss: 0.5623, eval loss 0.4846654236316681
Epoch 88 train loss: 0.6004, eval loss 0.4844622015953064
Epoch 89 train loss: 0.5559, eval loss 0.48437801003456116
Epoch 90 train loss: 0.6100, eval loss 0.4842328727245331
Epoch 91 train loss: 0.5367, eval loss 0.4841471314430237
Epoch 92 train loss: 0.5315, eval loss 0.4841289818286896
Epoch 93 train loss: 0.4777, eval loss 0.4837525188922882
Epoch 94 train loss: 0.5133, eval loss 0.483860045671463
Epoch 95 train loss: 0.5779, eval loss 0.48386552929878235
Epoch 96 train loss: 0.5253, eval loss 0.4837500751018524
Epoch 97 train loss: 0.5826, eval loss 0.48360174894332886
Epoch 98 train loss: 0.5198, eval loss 0.4829147458076477
Epoch 99 train loss: 0.6016, eval loss 0.4829179346561432
Epoch 100 train loss: 0.6439, eval loss 0.48280462622642517
Epoch 101 train loss: 0.5759, eval loss 0.48278090357780457
Epoch 102 train loss: 0.5578, eval loss 0.48274821043014526
Epoch 103 train loss: 0.5868, eval loss 0.4828287661075592
Epoch 104 train loss: 0.5576, eval loss 0.4822773635387420 7
Epoch 105 train loss: 0.6026, eval loss 0.4822211265563965
Epoch 106 train loss: 0.4990, eval loss 0.4823896884918213
Epoch 107 train loss: 0.6128, eval loss 0.48225468397140503
Epoch 108 train loss: 0.5556, eval loss 0.4820731282234192
Epoch 109 train loss: 0.5022, eval loss 0.48167186975479126
Epoch 110 train loss: 0.5185, eval loss 0.48150694370269775
Epoch 111 train loss: 0.5833, eval loss 0.4814373850822449
Epoch 112 train loss: 0.4770, eval loss 0.4816868305206299
Epoch 113 train loss: 0.4980, eval loss 0.480984628200531
Epoch 114 train loss: 0.5282, eval loss 0.48161566257476807
Epoch 115 train loss: 0.5697, eval loss 0.4808390736579895
Epoch 116 train loss: 0.6307, eval loss 0.48124048113822937
Epoch 117 train loss: 0.5829, eval loss 0.48085975646972656
Epoch 118 train loss: 0.5423, eval loss 0.48054617643356323
Epoch 119 train loss: 0.5492, eval loss 0.4811133444309234 6
```

```
Epoch 120 train loss: 0.5084, eval loss 0.4808557629585266
Epoch 121 train loss: 0.4946, eval loss 0.4807373583316803
Early stopping!
```

In [75]:

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

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

```
AUROC: 90.57%
F1: 69.26%
Precision: 64.38%
Recall: 74.94%
```

Pytania kontrolne (1 punkt)

1. **Wymień 4 najważniejsze twoim zdaniem hiperparametry sieci neuronowej. A: liczba warstw, liczba neuronów w warstwach, funkcja aktywacji, funkcja kosztu**
2. **Czy widzisz jakiś problem w użyciu regularyzacji L1 w treningu sieci neuronowych? Czy dropout może twoim zdaniem stanowić alternatywę dla tego rodzaju regularyzacji? A: L1 regularyzuje wagi, a nie aktywacje, więc nie jest to najlepszy sposób regularyzacji. Dropout może być alternatywą, ale nie jest to najlepszy sposób regularyzacji.**
3. **Czy użycie innej metryki do wczesnego stopu da taki sam model końcowy? Czemu? A: Nie, ponieważ wczesne stopowanie jest związane z funkcją kosztu, a nie z metryką.**

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

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

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

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

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

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

        optimizer.step()
        optimizer.zero_grad()
```

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

        step_counter += 1

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

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

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

Dla zainteresowanych polecamy tę serie artykułów

Zadanie dla chętnych

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

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

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

Przeszukaj wybrane zakresy dla hiperparametrów:

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

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

Przydatne materiały:

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

In [ ]:

```
```