Sieci neuronowe

Wstęp

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

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

Wykorzystywane biblioteki

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

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

Pierwszy duży framework do deep learningu, oraz obecnie najpopularniejszy, to

TensorFlow, wraz z wysokopoziomową nakładką Keras. Są jednak szanse, że Google (autorzy) będzie go powoli porzucać na rzecz ich nowego frameworka JAX (dyskusja, artykuł Business Insidera), który jest bardzo świeżym, ale ciekawym narzędziem.

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

Konfiguracja własnego komputera

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

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

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

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

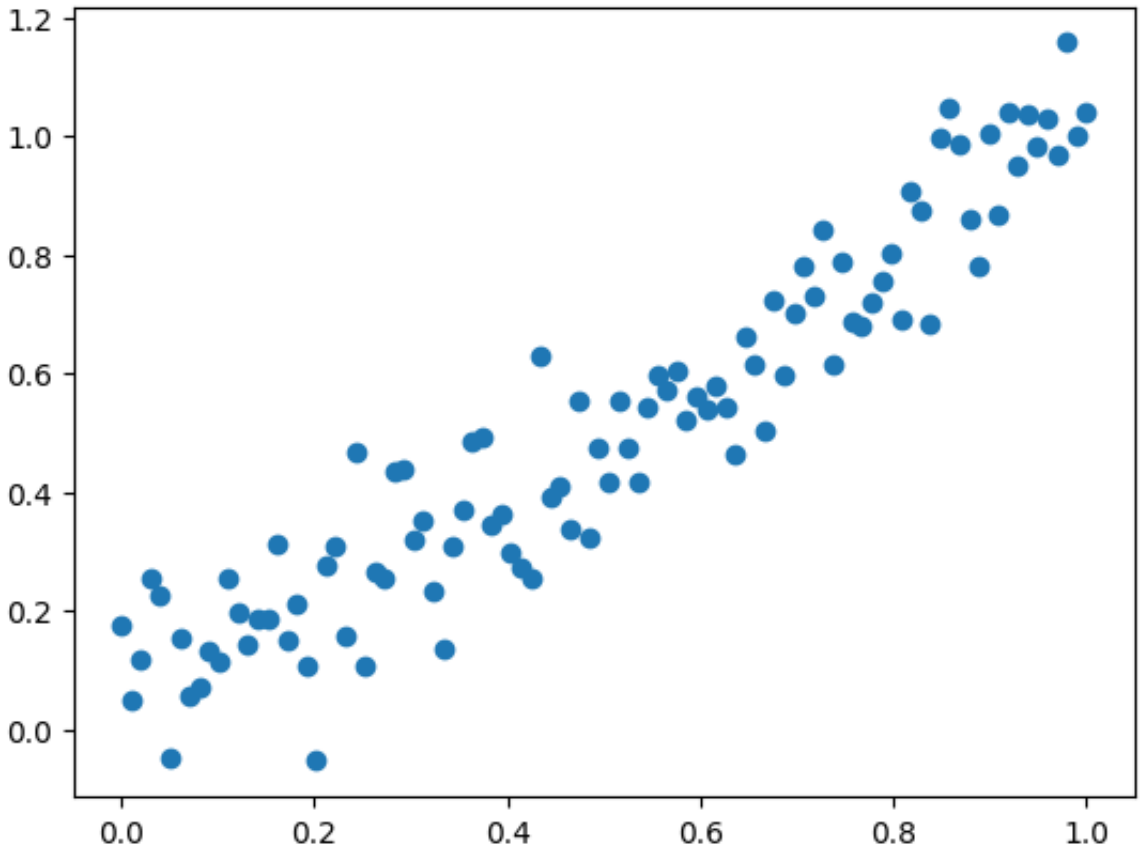
Wprowadzenie

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

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

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

In [ ]:
```
np.random.seed(0)

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

plt.scatter(x, y)
```

Out[ ]:
```
<matplotlib.collections.PathCollection at 0x176595b10>
```



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.

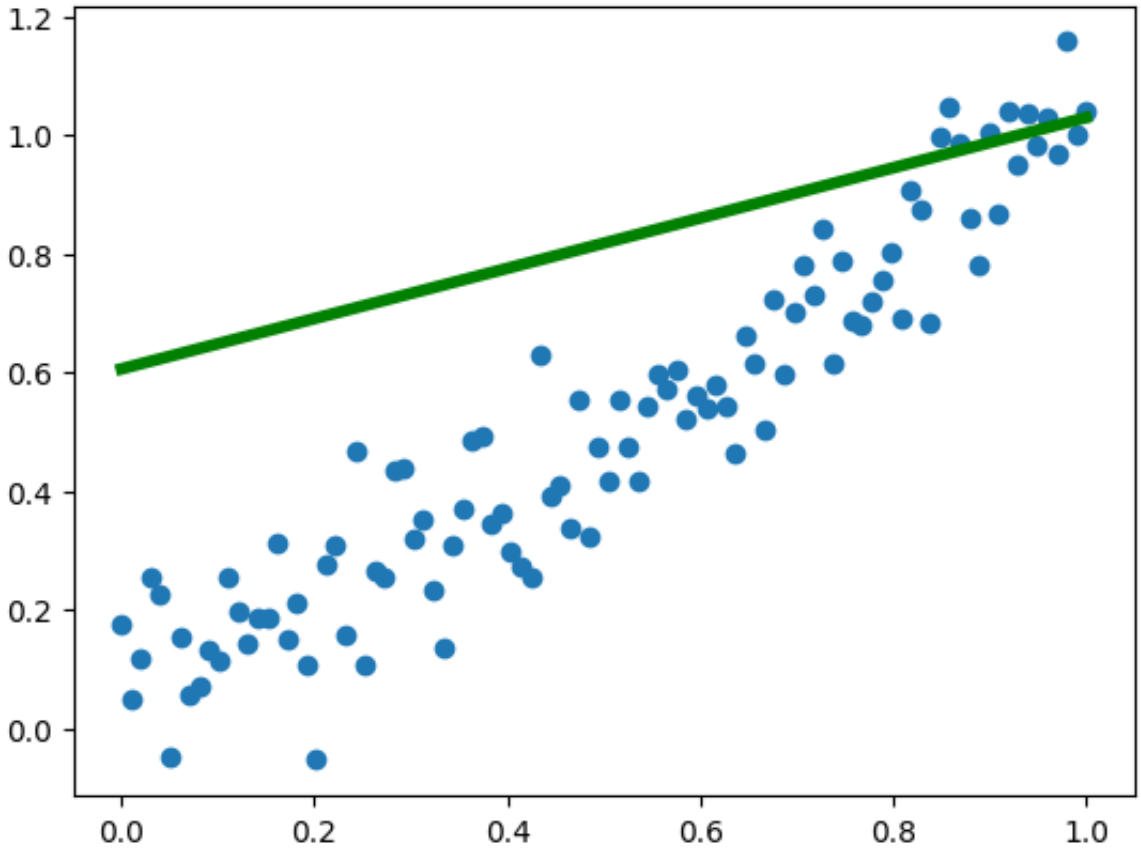
In [ ]:
```
def mse(y: np.ndarray, y_hat: np.ndarray) -> float:
    return np.sum((y - y_hat) ** 2) / len(y)
```

```
    raise NotImplementedError
```

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

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

```
MSE: 0.133
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1766c0c90>]
```



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

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

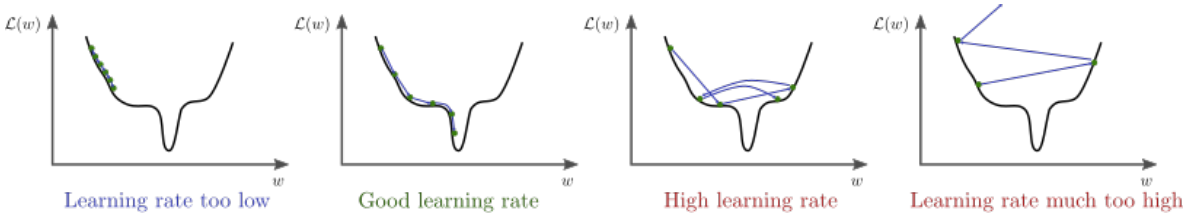
Stosując powyższe do optymalizacji, mamy:

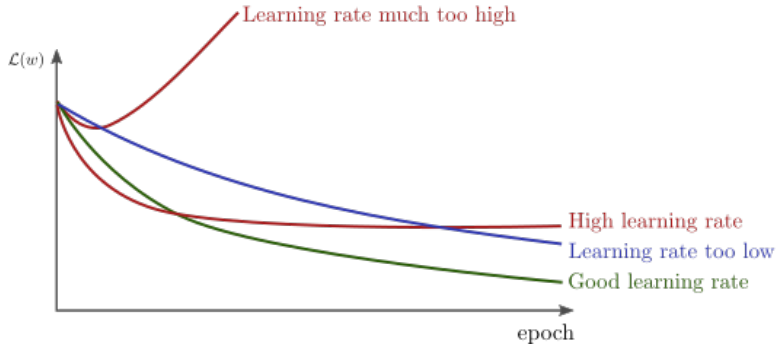
$$x_{t+1} = x_t - \alpha * \frac{f(x)}{dx}$$

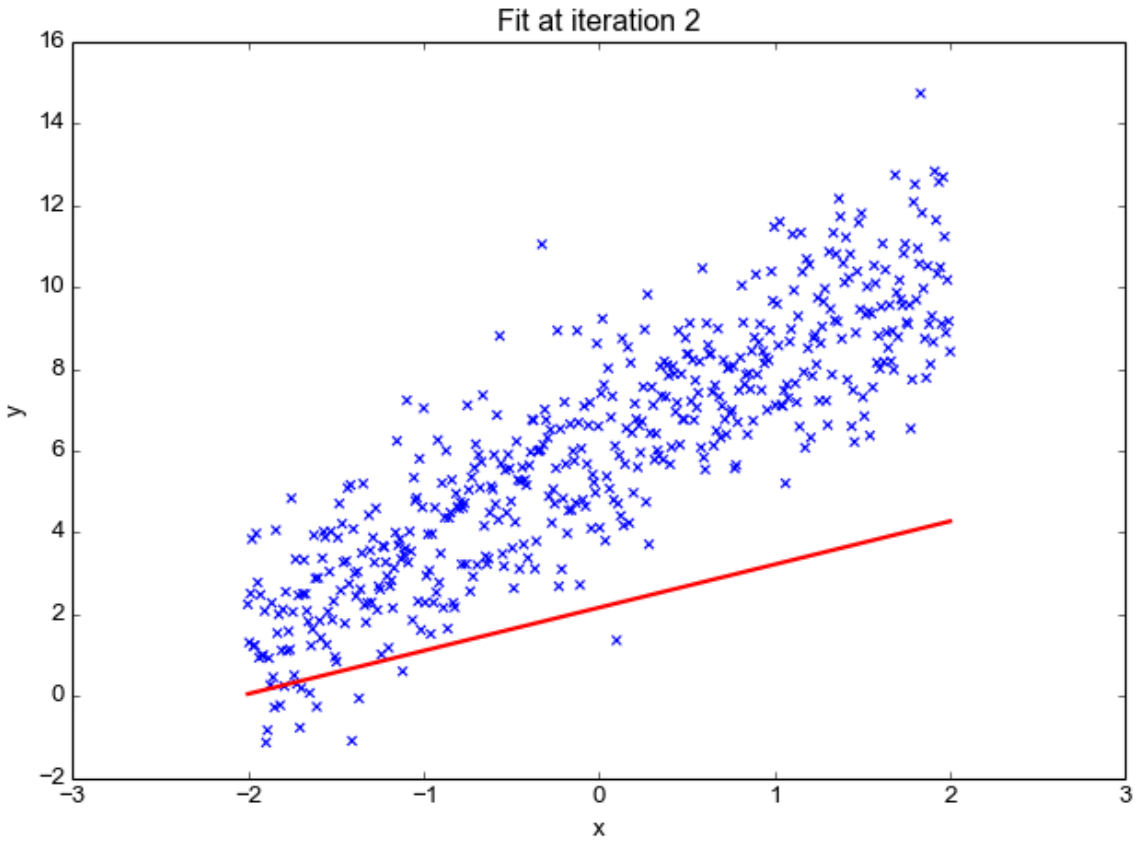
α to niewielka wartość (rzędu zwykle 10^{-5} - 10^{-2}), wprowadzona, aby trzymać się założenia o małej zmianie parametrów (ϵ). Nazywa się ją **stałą uczącą (learning rate)** i jest zwykle najważniejszym hiperparametrem podczas nauki sieci.

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:

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

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

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

Obliczenie pochodnych cząstkowych ze względu na każdy

Zadanie 2 (1.5 punkty)

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

In [ ]:
```
def optimize(
    x: np.ndarray, y: np.ndarray, a: float, b: float, learning_rate: floa
):
    y_hat = a * x + b
    errors = y - y_hat
    new_a = a + learning_rate * np.dot(errors, -x) * (-2/len(y))
    new_b = b + learning_rate * np.sum(errors) * (2/len(y))

    return new_a, new_b
```

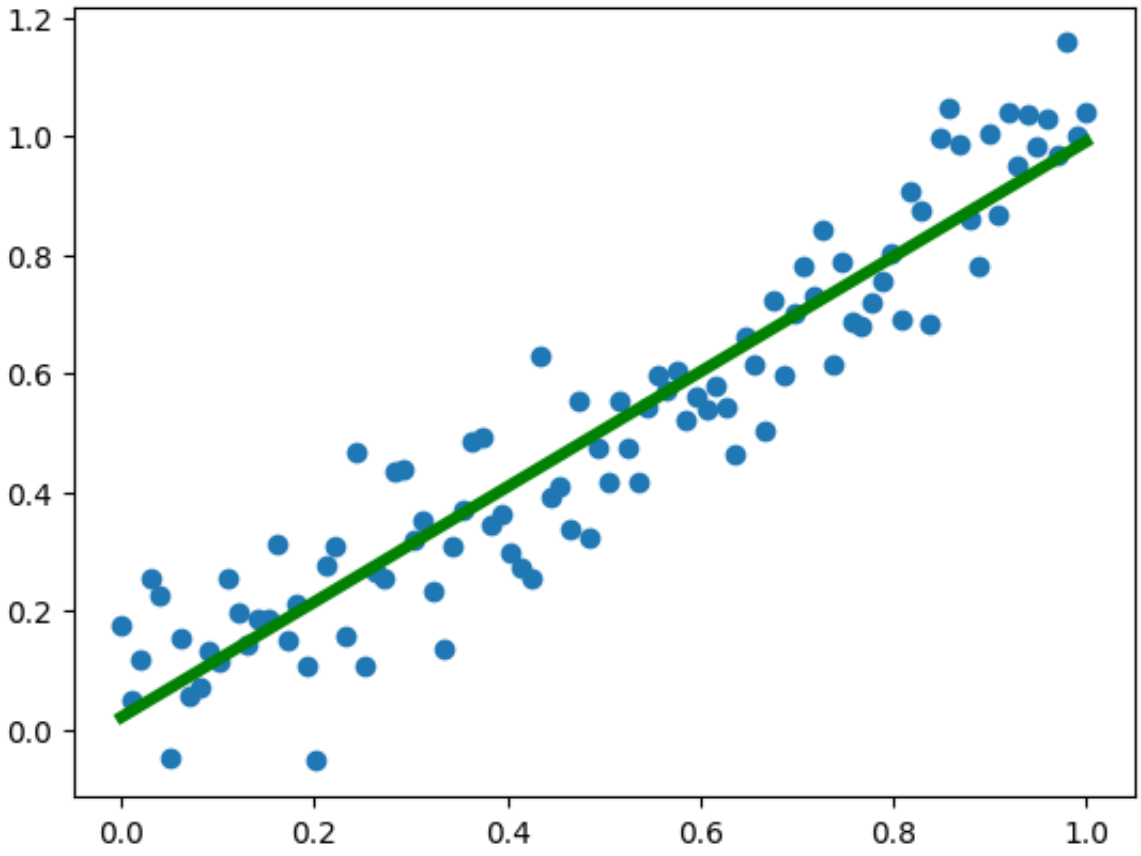
In [ ]:
```
for i in range(1000):
    loss = mse(y, a * x + b)
```

```
    a, b = optimize(x, y, a, b)
    if i % 100 == 0:
        print(f"step {i} loss: ", loss)

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

```
step 0 loss:  0.1330225119404028
step 100 loss:  0.012673197778527677
step 200 loss:  0.010257153540857817
step 300 loss:  0.0100948037549359
step 400 loss:  0.010083894412889118
step 500 loss:  0.010083161342973332
step 600 loss:  0.010083112083219709
step 700 loss:  0.010083108773135261
step 800 loss:  0.010083108550709076
step 900 loss:  0.01008310853576281
final loss: 0.010083108534760455
```

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

Out[ ]: `[<matplotlib.lines.Line2D at 0x1700098d0>]`



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

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

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

Wprowadzenie do PyTorcha

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

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

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

Najważniejsze moduły:

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

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

In [ ]:
```
ones = torch.ones(10)
noise = torch.ones(10) * torch.rand(10)

# elementwise sum
print(ones + noise)

# elementwise multiplication
```

```
print(ones * noise)

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

```
tensor([1.0290, 1.3804, 1.0296, 1.8940, 1.2498, 1.9640, 1.8721, 1.6344, 1.
9160,
        1.8826])
tensor([0.0290, 0.3804, 0.0296, 0.8940, 0.2498, 0.9640, 0.8721, 0.6344, 0.
9160,
        0.8826])
tensor(5.8519)
```

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

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

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

Out[ ]:
```
(tensor([0.7216], requires_grad=True), tensor([0.1598], requires_grad=Tr
ue))
```

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

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

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

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

Out[ ]:
```
tensor(0.0156, dtype=torch.float64, grad_fn=<MseLossBackward0>)
```

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

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

```
loss.backward()
```

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

```
tensor([-0.0276])
```

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

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

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

Out[ ]:
```
tensor([-0.0552])
```

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

In [ ]:
```
learning_rate = 0.1
for 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(0.0156, dtype=torch.float64, grad_fn=<MseLossBackward
0>)
step 100 loss:  tensor(0.0104, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 200 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 300 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 400 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 500 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 600 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 700 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 800 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
step 900 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackwa
rd0>)
final loss: tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0
>)
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 [ ]:
```python
# 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: 0.0685
step 100 loss: 0.0106
step 200 loss: 0.0101
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
>)
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 [ ]:
```
#!wget https://archive.ics.uci.edu/ml/machine-learning-databases/adult/ad
```

In [ ]:
```
import pandas as pd

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

"""
age: continuous.
workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-go
fnlwgt: continuous.
education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acd
education-num: continuous.
marital-status: Married-civ-spouse, Divorced, Never-married, Separated, W
occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-manage
relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Un
race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
sex: Female, Male.
capital-gain: continuous.
capital-loss: continuous.
hours-per-week: continuous.
native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Ge
"""

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

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

In [ ]:
```
# attribution: https://www.kaggle.com/code/royshih23/topic7-classificatio
df['education'].replace('Preschool', 'dropout',inplace=True)
```

```
df['education'].replace('10th', 'dropout',inplace=True)
df['education'].replace('11th', 'dropout',inplace=True)
df['education'].replace('12th', 'dropout',inplace=True)
df['education'].replace('1st-4th', 'dropout',inplace=True)
df['education'].replace('5th-6th', 'dropout',inplace=True)
df['education'].replace('7th-8th', 'dropout',inplace=True)
df['education'].replace('9th', 'dropout',inplace=True)
df['education'].replace('HS-Grad', 'HighGrad',inplace=True)
df['education'].replace('HS-grad', 'HighGrad',inplace=True)
df['education'].replace('Some-college', 'CommunityCollege',inplace=True)
df['education'].replace('Assoc-acdm', 'CommunityCollege',inplace=True)
df['education'].replace('Assoc-voc', 'CommunityCollege',inplace=True)
df['education'].replace('Bachelors', 'Bachelors',inplace=True)
df['education'].replace('Masters', 'Masters',inplace=True)
df['education'].replace('Prof-school', 'Masters',inplace=True)
df['education'].replace('Doctorate', 'Doctorate',inplace=True)

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

In [ ]:

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

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

train_valid_size = 0.2

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

continuous_cols = ['age', 'fnlwgt', 'education-num', 'capital-gain', 'cap
continuous_X_train = X_train[continuous_cols]
categorical_X_train = X_train.loc[:, ~X_train.columns.isin(continuous_col

continuous_X_valid = X_valid[continuous_cols]
```

```
categorical_X_valid = X_valid.loc[:, ~X_valid.columns.isin(continuous_col

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

categorical_encoder = OneHotEncoder(sparse=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=
X_valid = np.concatenate([continuous_X_valid, categorical_X_valid], axis=
X_test = np.concatenate([continuous_X_test, categorical_X_test], axis=1)

X_train.shape, y_train.shape
```

```
/Users/filipdziurdzia/Desktop/Studia/Semestr 5/PSI 2/Artificial-Intelligen
ce-AGH/PSI/lib/python3.11/site-packages/sklearn/preprocessing/_encoders.p
y:975: FutureWarning: `sparse` was renamed to `sparse_output` in version
1.2 and will be removed in 1.4. `sparse_output` is ignored unless you leav
e `sparse` to its default value.
  warnings.warn(
```

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

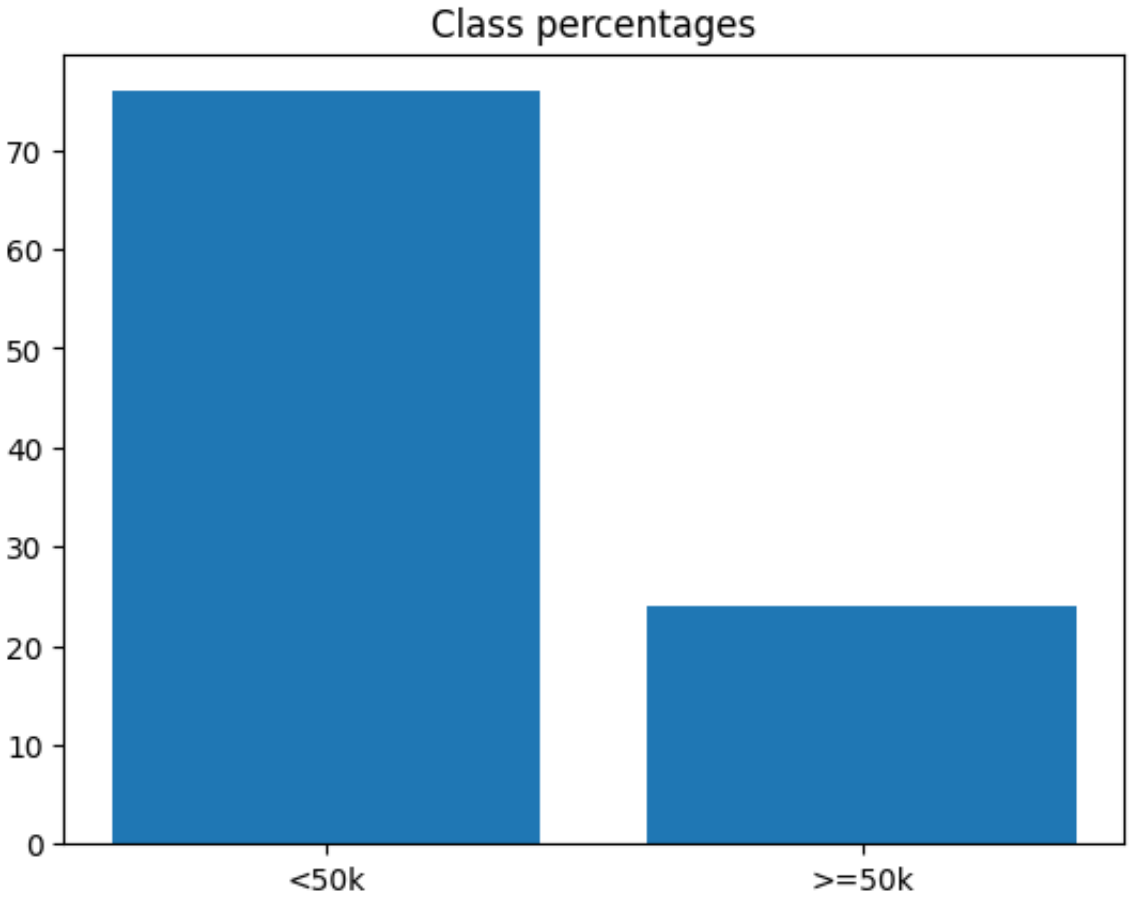
Uwaga co do typów - PyTorchu wszystko w sieci neuronowej musi być typu `float32`. W szczególności trzeba uważać na konwersje z Numpy'a, który używa domyślnie typu `float64`. Może ci się przydać metoda `.float()`.

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

In [ ]:
```
X_train = torch.from_numpy(X_train).float()
y_train = torch.from_numpy(y_train).float().unsqueeze(-1)

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

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

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

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

model = nn.Linear(108, 1)
activation = nn.Sigmoid()
optimizer = optim.SGD(model.parameters(), learning_rate)
loss_fn = nn.BCELoss()
best_loss = float("inf")

for i in range(3000):
    y = activation(model(X_train))

    loss = loss_fn(y, y_train)

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()

    if i % 300 == 0:
        if loss < best_loss:
            best_loss = loss
        print(f"step {i} loss: {loss.item():.4f}")



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

```
step 0 loss: 0.7004
step 300 loss: 0.6078
step 600 loss: 0.5540
step 900 loss: 0.5203
step 1200 loss: 0.4973
step 1500 loss: 0.4805
step 1800 loss: 0.4674
step 2100 loss: 0.4569
step 2400 loss: 0.4480
step 2700 loss: 0.4405
final loss: 0.4339
```

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

Dodatkowo podczas liczenia predykcji dobrze jest wyłączyć liczenie gradientów, bo

nie będą potrzebne, a oszczędza to czas i pamięć. Używa się do tego menadżera kontekstu `with torch.no_grad():`.

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

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

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

```
AUROC: 85.64%
```

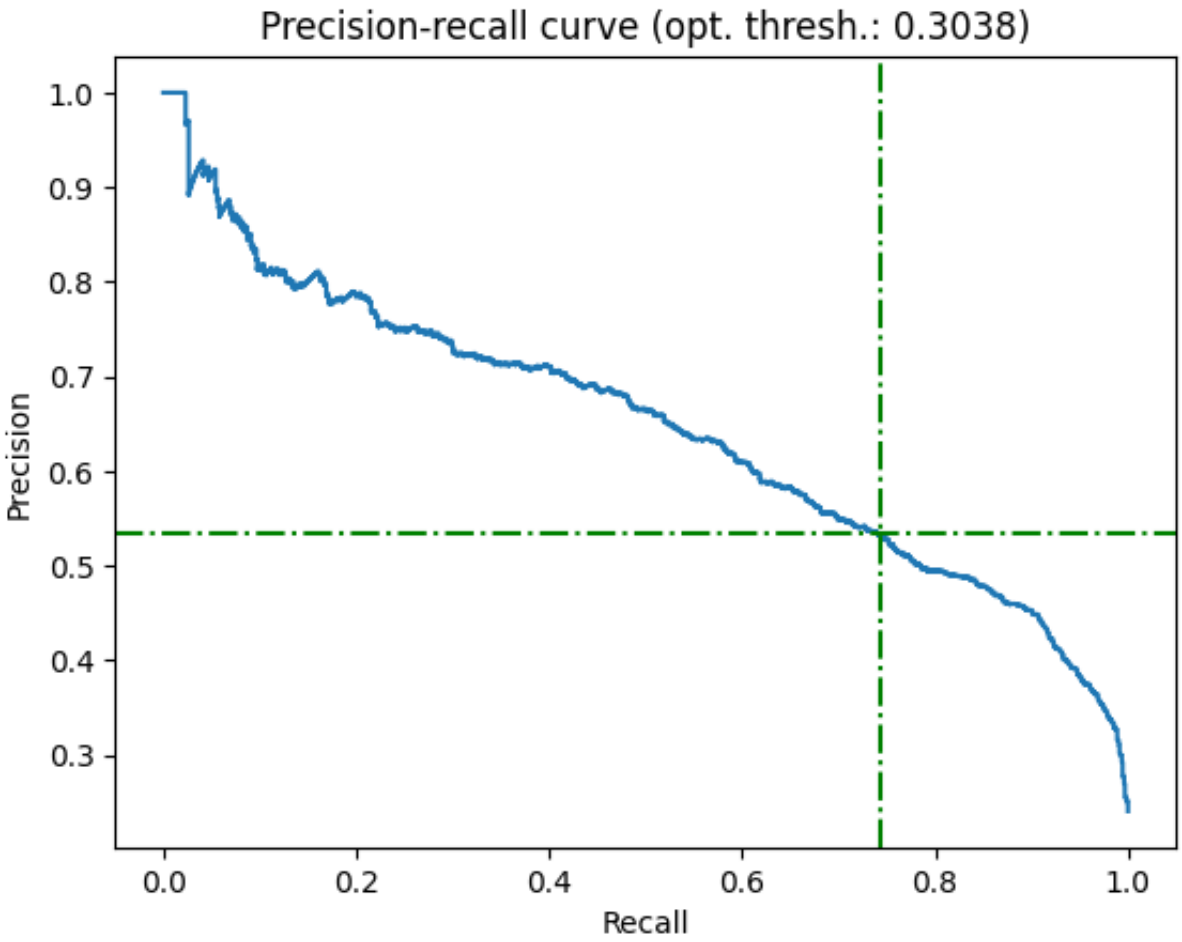
Jest to całkiem dobry wynik, a może być jeszcze lepszy. Sprawdźmy dla pewności jeszcze inne metryki: precyzję, recall oraz F1-score. Dodatkowo narysujemy krzywą precision-recall, czyli jak zmieniają się te metryki w zależności od przyjętego progu (threshold) prawdopodobieństwa, powyżej którego przyjmujemy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

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

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


def get_optimal_threshold(
    precisions: np.array,
    recalls: np.array,
    thresholds: np.array
) -> Tuple[int, float]:
    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_pr
    optimal_idx, optimal_threshold = get_optimal_threshold(precisions, re

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

```
    plt.axhline(precisions[optimal_idx], color="green", linestyle="-.")
    plt.show()
```

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

plot_precision_recall_curve(y_valid, y_pred_valid_score)
```



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

Sieci neuronowe

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

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

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

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

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

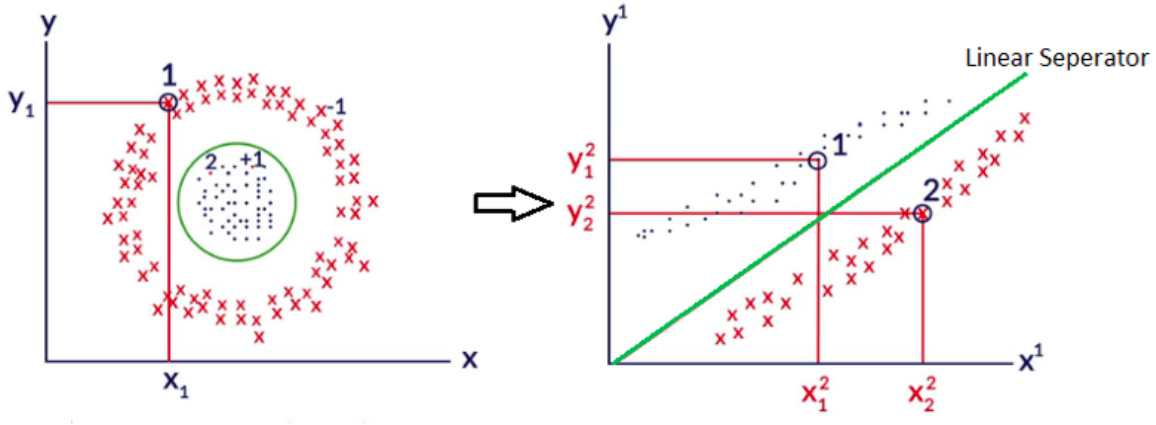
Dla zainteresowanych polecamy doskonałą książkę "Dive into Deep Learning", z implementacjami w PyTorchu, klasyczną książkę "Deep Learning Book", oraz ten filmik, jeśli zastanawiałeś/-aś się, czemu używamy deep learning, a nie naprzykład (wide?) learning. (aka. czemu staramy się budować głębokie sieci,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

Sieci MLP

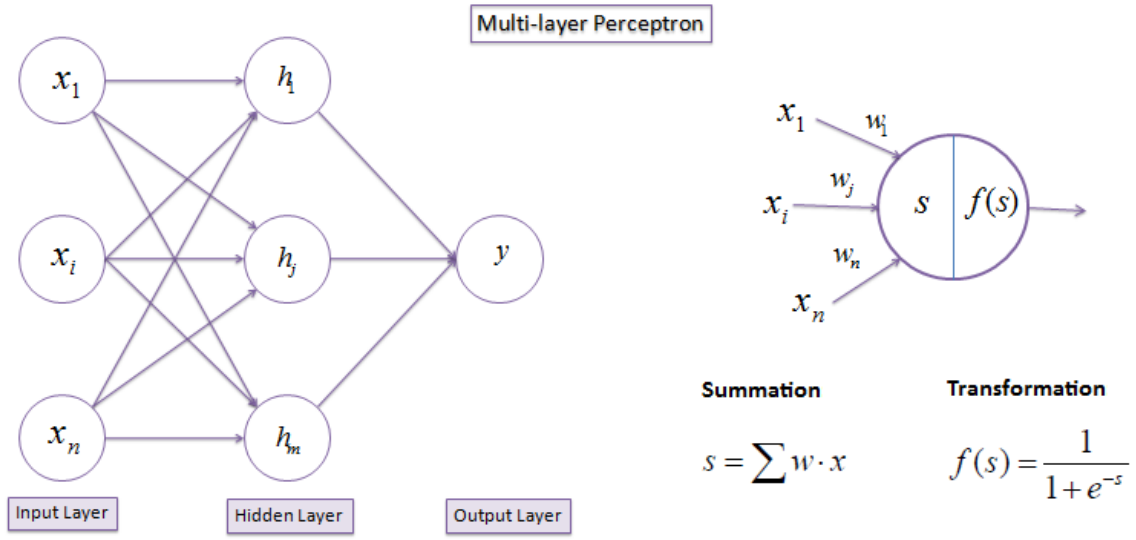
Dla przypomnienia, na wejściu mamy punkty ze zbioru treningowego, czyli d-wymiarowe wektory. W klasyfikacji chcemy znaleźć granicę decyzyjną, czyli krzywą, która oddzieli od siebie klasy. W wejściowej przestrzeni może być to trudne, bo chmury punktów z poszczególnych klas mogą być ze sobą dość pomieszane.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

Sieć MLP składa się z warstw. Każda z nich dokonuje nieliniowego przekształcenia przestrzeni (można o tym myśleć jak o składaniu przestrzeni jakąś prostą/łamaną), tak, aby w finalnej przestrzeni nasze punkty były możliwie liniowo separowalne.

Wtedy ostatnia warstwa z sigmoidą będzie potrafiła je rozdzielić od siebie.



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



Zapisane matematycznie MLP to:

$h_1 = f_1(x)$

$h_2 = f_2(h_1)$

$h_3 = f_3(h_2)$

$\ldots h_n = f_n(h_{n-1})$

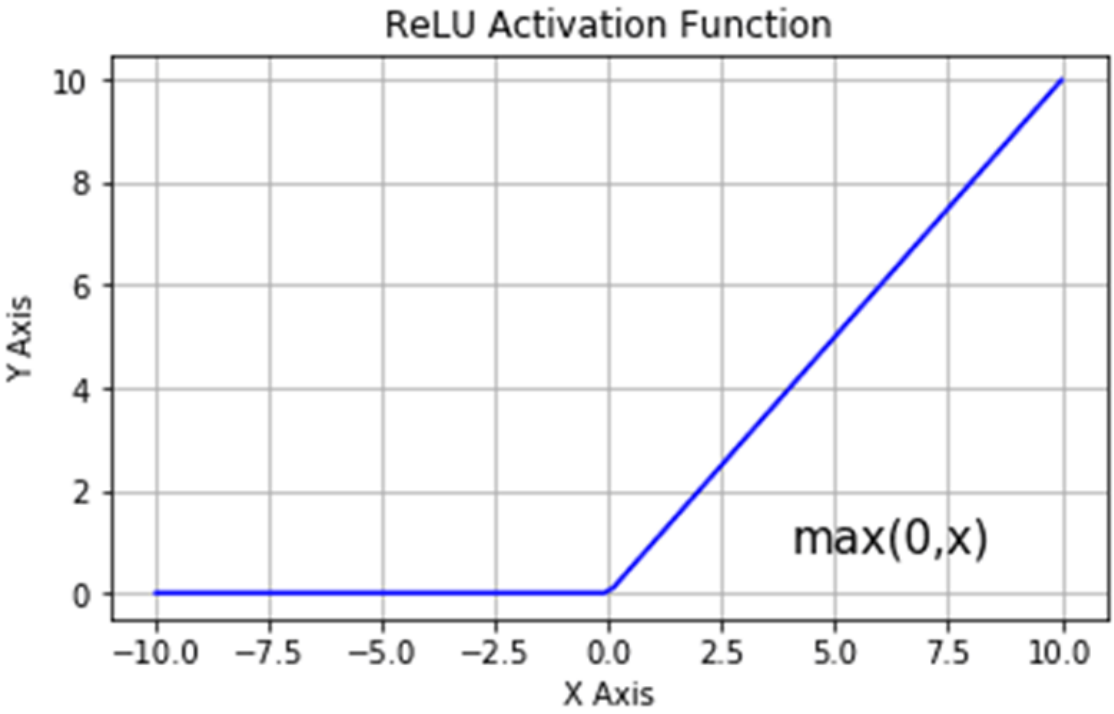
gdzie x to wejście f_i to funkcja aktywacji i-tej warstwy, a h_i to wyjście i-tej warstwy, nazywane **ukrytą reprezentacją (hidden representation)**, lub *latent representation*.

Nazwa bierze się z tego, że w środku sieci wyciągamy cechy i wzorce w danych, które nie są widoczne na pierwszy rzut oka na wejściu.

Załóżmy, że nie mamy funkcji aktywacji, czyli mamy aktywację liniową $f(x) = x$. Zobaczmy na początku sieci:
$h_1 = f_1(x) = xh_2 = f_2(f_1) = f_2(x) = x \dots h_n = f_n(f_{n-1}) = f_n(x) = x$ Jak widać, taka sieć niczego się nie nauczy. Wynika to z tego, że złożenie funkcji liniowych jest także funkcją liniową - patrz notatki z algebry :)

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

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



MLP w PyTorchu

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

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

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

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

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

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

Zadanie 4 (1 punkt)

Uzupełnij implementację 3-warstwowej sieci MLP. Użyj rozmiarów:

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

Użyj funkcji aktywacji ReLU.

Przydatne klasy:

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

In [ ]:
```python
from torch import sigmoid

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

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

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

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

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

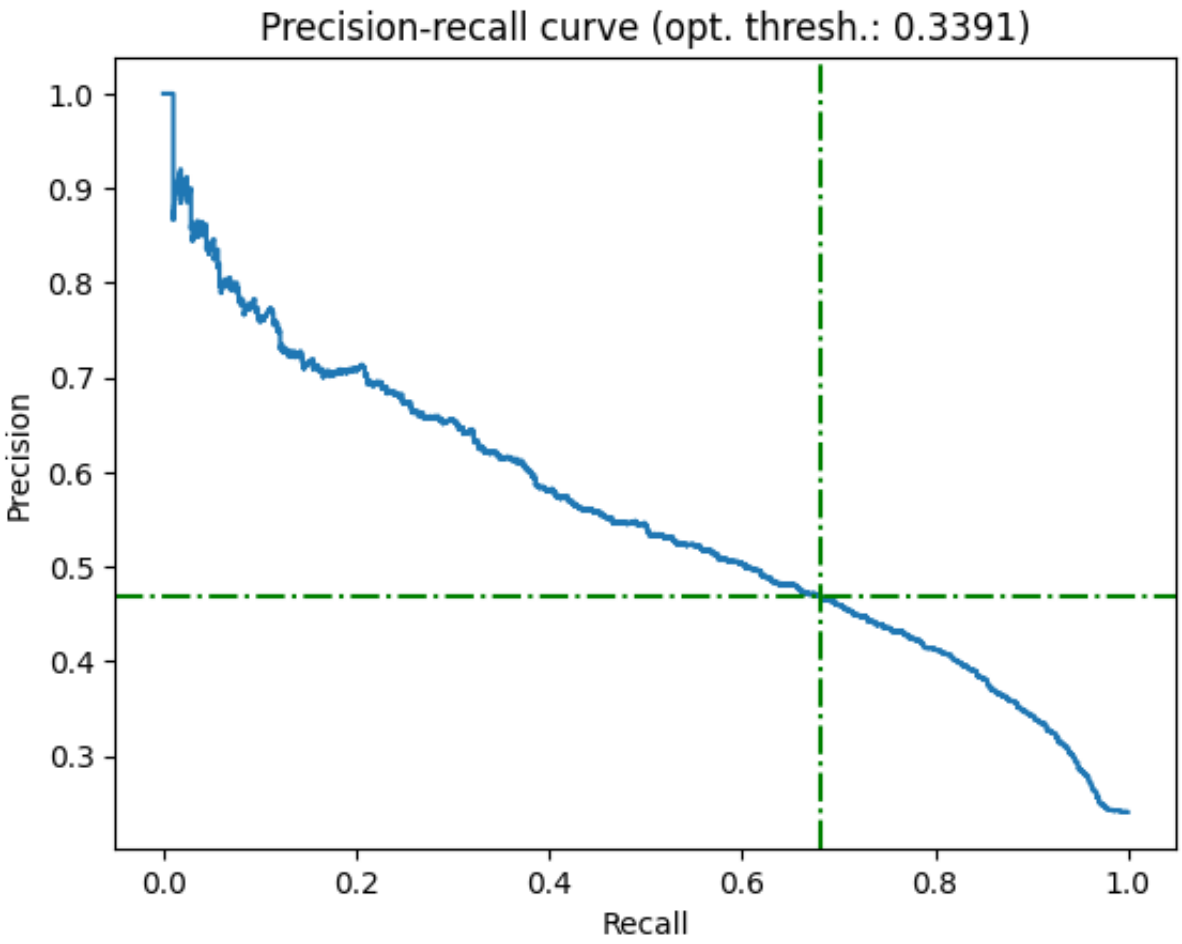
In [ ]:
```python
learning_rate = 1e-3
model = MLP(input_size=X_train.shape[1])
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# 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: 0.6888
Epoch 200 train loss: 0.6672
Epoch 400 train loss: 0.6487
Epoch 600 train loss: 0.6326
Epoch 800 train loss: 0.6184
Epoch 1000 train loss: 0.6059
Epoch 1200 train loss: 0.5947
Epoch 1400 train loss: 0.5847
Epoch 1600 train loss: 0.5758
Epoch 1800 train loss: 0.5678
final loss: 0.5607
```

In [ ]:
```
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: 78.23%
```



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 [ ]:
```
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_valid = model.predict_proba(X_valid)
        y_pred_test = model.predict_proba(X_test)

    auroc = roc_auc_score(y_test, y_pred_test)

    if threshold is None:
        precisions, recalls, thresholds = precision_recall_curve(y, y_pre
        _, threshold = get_optimal_threshold(precisions, recalls, thresho

    with torch.no_grad():
        y_pred = model.predict_proba(X)
        y_pred = (y_pred > threshold).float()

    loss = loss_fn(y_pred, y)
    f1 = f1_score(y, y_pred)
    precision = precision_score(y, y_pred)
    recall = recall_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 [ ]:
```
class RegularizedMLP(nn.Module):
    def __init__(self, input_size: int, dropout_p: float = 0.5):
        super().__init__()

        # implement me!
        self.mlp = nn.Sequential(
            nn.Linear(input_size, 256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, 128),
            nn.Dropout(0.5),
```

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

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

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

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

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

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

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

Na razie użyjemy jednak minibatch SGD.

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

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

In [ ]:
```
from torch.utils.data import Dataset

class MyDataset(Dataset):
```

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

        self.data = data
        self.y = y

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

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

Zadanie 7 (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 [ ]:
```
from copy import deepcopy

from torch.utils.data import DataLoader


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

early_stopping_patience = 4
```

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

```
loss_fn = torch.nn.BCEWithLogitsLoss()

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

steps_without_improvement = 0

best_val_loss = np.inf
best_model = None
best_threshold = None

for epoch_num in range(max_epochs):

    if steps_without_improvement >= early_stopping_patience:
        break

    model.train()

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

        optimizer.step()
        optimizer.zero_grad()


    # model evaluation, early stopping

    valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
    if valid_metrics["loss"] < best_val_loss:
        best_model = deepcopy(model)
        best_val_loss = valid_metrics["loss"]
        best_threshold = valid_metrics["optimal_threshold"]
        steps_without_improvement = 0
    else:
        steps_without_improvement += 1

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

```
/var/folders/qt/z4grrnl54m18jsyt2pvlxy5m0000gn/T/ipykernel_98676/223719159
2.py:9: RuntimeWarning: invalid value encountered in divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
/var/folders/qt/z4grrnl54m18jsyt2pvlxy5m0000gn/T/ipykernel_98676/223719159
2.py:9: RuntimeWarning: invalid value encountered in divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
Epoch 0 train loss: 0.6554, eval loss 0.965893030166626
Epoch 1 train loss: 0.6366, eval loss 0.8830918669700623
```

```
/var/folders/qt/z4grrnl54m18jsyt2pvlxy5m0000gn/T/ipykernel_98676/2237191592.py:9: RuntimeWarning: invalid value encountered in divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
/var/folders/qt/z4grrnl54m18jsyt2pvlxy5m0000gn/T/ipykernel_98676/2237191592.py:9: RuntimeWarning: invalid value encountered in divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
Epoch 2 train loss: 0.6261, eval loss 0.8812122344970703
Epoch 3 train loss: 0.6135, eval loss 0.8494081497192383
/var/folders/qt/z4grrnl54m18jsyt2pvlxy5m0000gn/T/ipykernel_98676/2237191592.py:9: RuntimeWarning: invalid value encountered in divide
  f1_scores = 2 * precisions * recalls / (precisions + recalls)
Epoch 4 train loss: 0.5987, eval loss 0.8162251114845276
Epoch 5 train loss: 0.5870, eval loss 0.7872433066368103
Epoch 6 train loss: 0.5758, eval loss 0.7892914414405823
Epoch 7 train loss: 0.5708, eval loss 0.7811291217803955
Epoch 8 train loss: 0.5607, eval loss 0.7685620188713074
Epoch 9 train loss: 0.5603, eval loss 0.7622537016868591
Epoch 10 train loss: 0.5579, eval loss 0.7658759951591492
Epoch 11 train loss: 0.5558, eval loss 0.758853554725647
Epoch 12 train loss: 0.5425, eval loss 0.7477027773857117
Epoch 13 train loss: 0.5421, eval loss 0.7432795763015747
Epoch 14 train loss: 0.5308, eval loss 0.7402428388595581
Epoch 15 train loss: 0.5219, eval loss 0.7369370460510254
Epoch 16 train loss: 0.5243, eval loss 0.7294609546661377
Epoch 17 train loss: 0.5220, eval loss 0.7347109317779541
Epoch 18 train loss: 0.5272, eval loss 0.7238904237747192
Epoch 19 train loss: 0.5210, eval loss 0.7264060974121094
Epoch 20 train loss: 0.5070, eval loss 0.7233114242553711
Epoch 21 train loss: 0.5186, eval loss 0.7152896523475647
Epoch 22 train loss: 0.5007, eval loss 0.7238036394119263
Epoch 23 train loss: 0.5076, eval loss 0.7237693071365356
Epoch 24 train loss: 0.4944, eval loss 0.7149754166603088
Epoch 25 train loss: 0.5098, eval loss 0.7286622524261475
Epoch 26 train loss: 0.4923, eval loss 0.7254174947738647
Epoch 27 train loss: 0.5057, eval loss 0.7111666798591614
Epoch 28 train loss: 0.4854, eval loss 0.7075187563896179
Epoch 29 train loss: 0.4811, eval loss 0.7034751176834106
Epoch 30 train loss: 0.4880, eval loss 0.7026880979537964
Epoch 31 train loss: 0.4842, eval loss 0.7020200490951538
Epoch 32 train loss: 0.4889, eval loss 0.7018741965293884
Epoch 33 train loss: 0.4753, eval loss 0.700545608997345
Epoch 34 train loss: 0.4688, eval loss 0.7030226588249207
Epoch 35 train loss: 0.4687, eval loss 0.69935542345047
Epoch 36 train loss: 0.4715, eval loss 0.7018978595733643
Epoch 37 train loss: 0.4616, eval loss 0.7010646462440491
Epoch 38 train loss: 0.4609, eval loss 0.7009188532829285
Epoch 39 train loss: 0.4605, eval loss 0.7007001042366028
```

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

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

```
print(f"Recall: {100 * test_metrics['recall']:.2f}%")
```

```
AUROC: 87.10%
F1: 62.98%
Precision: 57.88%
Recall: 69.07%
```

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

        # implement me!
        self.mlp = nn.Sequential(
            nn.Linear(input_size, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(128, 1),
        )

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

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

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

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

weights = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(y_train.cpu()),
    y=np.array(y_train.cpu().flatten())
)

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

early_stopping_patience = 4
```

In [ ]:
```python
model = NormalizingMLP(
    input_size=X_train.shape[1],
    dropout_p=dropout_p
)
optimizer = torch.optim.AdamW(
   model.parameters(),
   lr=learning_rate,
   weight_decay=l2_reg
)
```

```
loss_fn = torch.nn.BCEWithLogitsLoss(pos_weight=torch.from_numpy(weights))

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

steps_without_improvement = 0

best_val_loss = np.inf
best_model = None
best_threshold = None

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

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

        optimizer.step()
        optimizer.zero_grad()

    # model evaluation, early stopping
    # implement me!

    model.eval()
    valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
    if valid_metrics['loss'] < best_val_loss:
        best_model = deepcopy(model)
        best_val_loss = valid_metrics['loss']
        best_threshold = valid_metrics['optimal_threshold']
        steps_without_improvement = 0
    else:
        steps_without_improvement += 1

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

```
Epoch 0 train loss: 0.5916, eval loss 0.7919982075691223
Epoch 1 train loss: 0.5897, eval loss 0.7924771904945374
Epoch 2 train loss: 0.5704, eval loss 0.7861838936805725
Epoch 3 train loss: 0.5745, eval loss 0.7903118133544922
Epoch 4 train loss: 0.5782, eval loss 0.7915897369384766
Epoch 5 train loss: 0.6097, eval loss 0.7911564707756042
Epoch 6 train loss: 0.5687, eval loss 0.7914497256278992
Epoch 7 train loss: 0.5614, eval loss 0.7934533953666687
Epoch 8 train loss: 0.4526, eval loss 0.7899014949798584
Epoch 9 train loss: 0.5900, eval loss 0.7892511487007141
Epoch 10 train loss: 0.5307, eval loss 0.7891644835472107
Epoch 11 train loss: 0.4804, eval loss 0.786064863204956
Epoch 12 train loss: 0.5617, eval loss 0.787485659122467
Epoch 13 train loss: 0.4740, eval loss 0.7893909811973572
```

```
Epoch 14 train loss: 0.5598, eval loss 0.7883626818656921
Epoch 15 train loss: 0.4941, eval loss 0.7872895002365112
Epoch 16 train loss: 0.4477, eval loss 0.7888398170471191
Epoch 17 train loss: 0.5388, eval loss 0.7894777059555054
Epoch 18 train loss: 0.4395, eval loss 0.7901061773300171
Epoch 19 train loss: 0.4902, eval loss 0.7896291017532349
Epoch 20 train loss: 0.5197, eval loss 0.7869428992271423
Epoch 21 train loss: 0.4338, eval loss 0.7881036400794983
Epoch 22 train loss: 0.5045, eval loss 0.7912431955337524
Epoch 23 train loss: 0.5298, eval loss 0.7888054251670837
Epoch 24 train loss: 0.4588, eval loss 0.7884722352027893
Epoch 25 train loss: 0.5169, eval loss 0.7894244194030762
Epoch 26 train loss: 0.4583, eval loss 0.7896853685379028
Epoch 27 train loss: 0.4688, eval loss 0.7908443212509155
Epoch 28 train loss: 0.4239, eval loss 0.7889902591705322
Epoch 29 train loss: 0.4483, eval loss 0.7925658226013184
Epoch 30 train loss: 0.4131, eval loss 0.7891530394554138
Epoch 31 train loss: 0.4266, eval loss 0.7906480431556702
Epoch 32 train loss: 0.4422, eval loss 0.7899223566055298
Epoch 33 train loss: 0.4266, eval loss 0.7890986800193787
Epoch 34 train loss: 0.4267, eval loss 0.7916211485862732
Epoch 35 train loss: 0.4473, eval loss 0.7871360778808594
Epoch 36 train loss: 0.4837, eval loss 0.7889577150344849
Epoch 37 train loss: 0.4745, eval loss 0.7878302931785583
Epoch 38 train loss: 0.5155, eval loss 0.7864990830421448
Epoch 39 train loss: 0.4023, eval loss 0.7869124412536621
Epoch 40 train loss: 0.4563, eval loss 0.7870962619781494
Epoch 41 train loss: 0.3910, eval loss 0.7888283133506775
Epoch 42 train loss: 0.4209, eval loss 0.7876590490341187
Epoch 43 train loss: 0.3967, eval loss 0.7901281118392944
Epoch 44 train loss: 0.4507, eval loss 0.7895100116729736
Epoch 45 train loss: 0.4417, eval loss 0.7891309857368469
Epoch 46 train loss: 0.3468, eval loss 0.7886759042739868
Epoch 47 train loss: 0.4135, eval loss 0.7883303761482239
Epoch 48 train loss: 0.3639, eval loss 0.78890460729599
Epoch 49 train loss: 0.4058, eval loss 0.7895863056182861
Epoch 50 train loss: 0.4761, eval loss 0.7902815341949463
Epoch 51 train loss: 0.4018, eval loss 0.7882426381111145
Epoch 52 train loss: 0.4173, eval loss 0.7887625694274902
Epoch 53 train loss: 0.4293, eval loss 0.7902346849441528
Epoch 54 train loss: 0.4243, eval loss 0.7887730002403259
Epoch 55 train loss: 0.4272, eval loss 0.7919020056724548
Epoch 56 train loss: 0.3877, eval loss 0.7952930331230164
Epoch 57 train loss: 0.3997, eval loss 0.7893796563148499
Epoch 58 train loss: 0.4178, eval loss 0.7907556295394897
Epoch 59 train loss: 0.3915, eval loss 0.7893043756484985
Epoch 60 train loss: 0.3814, eval loss 0.7909851670265198
Epoch 61 train loss: 0.4636, eval loss 0.7921639680862427
Epoch 62 train loss: 0.3868, eval loss 0.7926734089851379
Epoch 63 train loss: 0.3763, eval loss 0.7931922674179077
Epoch 64 train loss: 0.4058, eval loss 0.7899109721183777
Epoch 65 train loss: 0.4360, eval loss 0.7873238325119019
Epoch 66 train loss: 0.4030, eval loss 0.7879826426506042
```

```
Epoch 67 train loss: 0.4315, eval loss 0.7886664271354675
Epoch 68 train loss: 0.4565, eval loss 0.7898681163787842
Epoch 69 train loss: 0.3438, eval loss 0.7910469174385071
Epoch 70 train loss: 0.4511, eval loss 0.7905269861221313
Epoch 71 train loss: 0.3685, eval loss 0.7907775044441223
Epoch 72 train loss: 0.4506, eval loss 0.7923591732978821
Epoch 73 train loss: 0.4235, eval loss 0.791893482208252
Epoch 74 train loss: 0.3941, eval loss 0.7940933108329773
Epoch 75 train loss: 0.3916, eval loss 0.7904621958732605
Epoch 76 train loss: 0.4225, eval loss 0.7935818433761597
Epoch 77 train loss: 0.3596, eval loss 0.7929219603538513
Epoch 78 train loss: 0.4029, eval loss 0.7880340218544006
Epoch 79 train loss: 0.3562, eval loss 0.7906804084777832
Epoch 80 train loss: 0.3721, eval loss 0.7931485176086426
Epoch 81 train loss: 0.3951, eval loss 0.7899632453918457
Epoch 82 train loss: 0.3083, eval loss 0.7915583848953247
Epoch 83 train loss: 0.3732, eval loss 0.7892710566520691
Epoch 84 train loss: 0.4480, eval loss 0.7898786067962646
Epoch 85 train loss: 0.3525, eval loss 0.7948492169380188
Epoch 86 train loss: 0.3930, eval loss 0.7934836745262146
Epoch 87 train loss: 0.4115, eval loss 0.7901793718338013
Epoch 88 train loss: 0.3607, eval loss 0.7874826788902283
Epoch 89 train loss: 0.3827, eval loss 0.7931475639343262
Epoch 90 train loss: 0.3703, eval loss 0.7892282009124756
Epoch 91 train loss: 0.3887, eval loss 0.7892147898674011
Epoch 92 train loss: 0.4838, eval loss 0.7944473624229431
Epoch 93 train loss: 0.4163, eval loss 0.7948128581047058
Epoch 94 train loss: 0.3273, eval loss 0.7954089641571045
Epoch 95 train loss: 0.3699, eval loss 0.7928980588912964
Epoch 96 train loss: 0.3454, eval loss 0.7912203073501587
Epoch 97 train loss: 0.3461, eval loss 0.7912496328353882
Epoch 98 train loss: 0.4260, eval loss 0.7890613675117493
Epoch 99 train loss: 0.3716, eval loss 0.7917152643203735
Epoch 100 train loss: 0.3255, eval loss 0.7939378023147583
Epoch 101 train loss: 0.4046, eval loss 0.7904155254364014
Epoch 102 train loss: 0.3805, eval loss 0.7972704768180847
Epoch 103 train loss: 0.3352, eval loss 0.7959290146827698
Epoch 104 train loss: 0.3681, eval loss 0.7933522462844849
Epoch 105 train loss: 0.3698, eval loss 0.7932959198951721
Epoch 106 train loss: 0.4243, eval loss 0.7937520742416382
Epoch 107 train loss: 0.3899, eval loss 0.797487735748291
Epoch 108 train loss: 0.3336, eval loss 0.7948671579360962
Epoch 109 train loss: 0.3589, eval loss 0.7950813174247742
Epoch 110 train loss: 0.3997, eval loss 0.7944861650466919
Epoch 111 train loss: 0.3346, eval loss 0.7935025691986084
Epoch 112 train loss: 0.3645, eval loss 0.7947919964790344
Epoch 113 train loss: 0.3931, eval loss 0.7937710881233215
Epoch 114 train loss: 0.3872, eval loss 0.7935110330581665
Epoch 115 train loss: 0.3523, eval loss 0.791551411151886
Epoch 116 train loss: 0.3735, eval loss 0.7924503087997437
Epoch 117 train loss: 0.2967, eval loss 0.796695351600647
Epoch 118 train loss: 0.3731, eval loss 0.7957859635353088
Epoch 119 train loss: 0.4001, eval loss 0.7952868938446045
```

```
Epoch 120 train loss: 0.3802, eval loss 0.7948536276817322
Epoch 121 train loss: 0.4921, eval loss 0.7957651019096375
Epoch 122 train loss: 0.3596, eval loss 0.7941396236419678
Epoch 123 train loss: 0.3653, eval loss 0.7975180149078369
Epoch 124 train loss: 0.3589, eval loss 0.793056845664978
Epoch 125 train loss: 0.3603, eval loss 0.7932196259498596
Epoch 126 train loss: 0.3845, eval loss 0.7951375246047974
Epoch 127 train loss: 0.3683, eval loss 0.796013593673706
Epoch 128 train loss: 0.3942, eval loss 0.796654462814331
Epoch 129 train loss: 0.3952, eval loss 0.7966316342353821
Epoch 130 train loss: 0.3472, eval loss 0.7943348288536072
Epoch 131 train loss: 0.3134, eval loss 0.795635461807251
Epoch 132 train loss: 0.3884, eval loss 0.796783983707428
Epoch 133 train loss: 0.3341, eval loss 0.7959822416305542
Epoch 134 train loss: 0.3272, eval loss 0.7963078022003174
Epoch 135 train loss: 0.3579, eval loss 0.7947586178779602
Epoch 136 train loss: 0.3915, eval loss 0.7958737015724182
Epoch 137 train loss: 0.3217, eval loss 0.7981256246566772
Epoch 138 train loss: 0.3418, eval loss 0.7962640523910522
Epoch 139 train loss: 0.3223, eval loss 0.7968268394470215
Epoch 140 train loss: 0.3835, eval loss 0.7988731265068054
Epoch 141 train loss: 0.3586, eval loss 0.7980598211288452
Epoch 142 train loss: 0.3534, eval loss 0.7951669692993164
Epoch 143 train loss: 0.4689, eval loss 0.7956011295318604
Epoch 144 train loss: 0.3749, eval loss 0.7969323396682739
Epoch 145 train loss: 0.3193, eval loss 0.7953402400016785
Epoch 146 train loss: 0.3488, eval loss 0.8005822896957397
Epoch 147 train loss: 0.4796, eval loss 0.7957420945167542
Epoch 148 train loss: 0.3264, eval loss 0.7986645102500916
Epoch 149 train loss: 0.3622, eval loss 0.7962611317634583
Epoch 150 train loss: 0.3687, eval loss 0.7994139790534973
Epoch 151 train loss: 0.3849, eval loss 0.7951250672340393
Epoch 152 train loss: 0.3903, eval loss 0.7954812049865723
Epoch 153 train loss: 0.3549, eval loss 0.7946051955223083
Epoch 154 train loss: 0.2947, eval loss 0.7979494333267212
Epoch 155 train loss: 0.2973, eval loss 0.7944965958595276
Epoch 156 train loss: 0.3259, eval loss 0.7944632768630981
Epoch 157 train loss: 0.4072, eval loss 0.7941396236419678
Epoch 158 train loss: 0.3559, eval loss 0.7916236519813538
Epoch 159 train loss: 0.3389, eval loss 0.7947452068328857
Epoch 160 train loss: 0.3621, eval loss 0.7946908473968506
Epoch 161 train loss: 0.4607, eval loss 0.7947022914886475
Epoch 162 train loss: 0.3223, eval loss 0.7982426285743713
Epoch 163 train loss: 0.3266, eval loss 0.7967904806137085
Epoch 164 train loss: 0.3310, eval loss 0.7989463210105896
Epoch 165 train loss: 0.3319, eval loss 0.7974094152450562
Epoch 166 train loss: 0.3073, eval loss 0.7962850332260132
Epoch 167 train loss: 0.3363, eval loss 0.7976704239845276
Epoch 168 train loss: 0.3197, eval loss 0.7972457408905029
Epoch 169 train loss: 0.3646, eval loss 0.7968478202819824
Epoch 170 train loss: 0.3459, eval loss 0.7944204807281494
Epoch 171 train loss: 0.3535, eval loss 0.7961296439170837
Epoch 172 train loss: 0.3571, eval loss 0.7958468198776245
```

```
Epoch 173 train loss: 0.3093, eval loss 0.8020200133323669
Epoch 174 train loss: 0.3358, eval loss 0.7948277592658997
Epoch 175 train loss: 0.3458, eval loss 0.795380175113678
Epoch 176 train loss: 0.3347, eval loss 0.7970076203346252
Epoch 177 train loss: 0.3346, eval loss 0.7967486381530762
Epoch 178 train loss: 0.3899, eval loss 0.7955772876739502
Epoch 179 train loss: 0.3157, eval loss 0.7945279479026794
Epoch 180 train loss: 0.3502, eval loss 0.7967132329940796
Epoch 181 train loss: 0.3066, eval loss 0.7959228754043579
Epoch 182 train loss: 0.3324, eval loss 0.7954164147377014
Epoch 183 train loss: 0.3661, eval loss 0.7968343496322632
Epoch 184 train loss: 0.3291, eval loss 0.7968666553497314
Epoch 185 train loss: 0.3398, eval loss 0.7966819405555725
Epoch 186 train loss: 0.3013, eval loss 0.7951242327690125
Epoch 187 train loss: 0.2963, eval loss 0.7917978763580322
Epoch 188 train loss: 0.3301, eval loss 0.794330894947052
Epoch 189 train loss: 0.3685, eval loss 0.7951983213424683
Epoch 190 train loss: 0.3218, eval loss 0.7978407144546509
Epoch 191 train loss: 0.3350, eval loss 0.7940843105316162
Epoch 192 train loss: 0.3326, eval loss 0.796920895576477
Epoch 193 train loss: 0.3927, eval loss 0.796626627445221
Epoch 194 train loss: 0.2899, eval loss 0.7976245880126953
Epoch 195 train loss: 0.3587, eval loss 0.7985244989395142
Epoch 196 train loss: 0.3546, eval loss 0.7962819337844849
Epoch 197 train loss: 0.4689, eval loss 0.7940948009490967
Epoch 198 train loss: 0.2950, eval loss 0.7936834692955017
Epoch 199 train loss: 0.3199, eval loss 0.7963675856590271
Epoch 200 train loss: 0.3920, eval loss 0.7956838011741638
Epoch 201 train loss: 0.3403, eval loss 0.7967561483383179
Epoch 202 train loss: 0.3660, eval loss 0.7981102466583252
Epoch 203 train loss: 0.4042, eval loss 0.8017925024032593
Epoch 204 train loss: 0.3599, eval loss 0.7972989678382874
Epoch 205 train loss: 0.3242, eval loss 0.795261025428772
Epoch 206 train loss: 0.2985, eval loss 0.7967007756233215
Epoch 207 train loss: 0.2934, eval loss 0.7988870739936829
Epoch 208 train loss: 0.2983, eval loss 0.7976445555686951
Epoch 209 train loss: 0.3474, eval loss 0.7981520295143127
Epoch 210 train loss: 0.3001, eval loss 0.7976883053779602
Epoch 211 train loss: 0.2585, eval loss 0.799181342124939
Epoch 212 train loss: 0.3071, eval loss 0.7970379590988159
Epoch 213 train loss: 0.4153, eval loss 0.7989014983177185
Epoch 214 train loss: 0.3432, eval loss 0.7993013858795166
Epoch 215 train loss: 0.3274, eval loss 0.7993555665016174
Epoch 216 train loss: 0.3834, eval loss 0.7999203205108643
Epoch 217 train loss: 0.3134, eval loss 0.7998307347297668
Epoch 218 train loss: 0.2945, eval loss 0.7987701296806335
Epoch 219 train loss: 0.3471, eval loss 0.7977721095085144
Epoch 220 train loss: 0.3058, eval loss 0.8006744384765625
Epoch 221 train loss: 0.3172, eval loss 0.7973940372467041
Epoch 222 train loss: 0.3470, eval loss 0.7983804941177368
Epoch 223 train loss: 0.2941, eval loss 0.7986311316490173
Epoch 224 train loss: 0.3090, eval loss 0.7999727129936218
Epoch 225 train loss: 0.2900, eval loss 0.7991176247596741
```

```
Epoch 226 train loss: 0.3264, eval loss 0.7962580323219299
Epoch 227 train loss: 0.3828, eval loss 0.8024532794952393
Epoch 228 train loss: 0.3645, eval loss 0.8017705678939819
Epoch 229 train loss: 0.3370, eval loss 0.8044663071632385
Epoch 230 train loss: 0.2654, eval loss 0.7992794513702393
Epoch 231 train loss: 0.3235, eval loss 0.8012830018997192
Epoch 232 train loss: 0.3130, eval loss 0.8025494813919067
Epoch 233 train loss: 0.3366, eval loss 0.8030474185943604
Epoch 234 train loss: 0.3180, eval loss 0.7996898889541626
Epoch 235 train loss: 0.2868, eval loss 0.7989996075630188
Epoch 236 train loss: 0.3962, eval loss 0.8001888394355774
Epoch 237 train loss: 0.2779, eval loss 0.7980015873908997
Epoch 238 train loss: 0.2839, eval loss 0.8020399212837219
Epoch 239 train loss: 0.3338, eval loss 0.7998736500740051
Epoch 240 train loss: 0.2981, eval loss 0.7987377047538757
Epoch 241 train loss: 0.3724, eval loss 0.8008592128753662
Epoch 242 train loss: 0.2850, eval loss 0.798888087272644
Epoch 243 train loss: 0.2557, eval loss 0.7994955778121948
Epoch 244 train loss: 0.2967, eval loss 0.7979807257652283
Epoch 245 train loss: 0.2969, eval loss 0.7980349659919739
Epoch 246 train loss: 0.3018, eval loss 0.8011733889579773
Epoch 247 train loss: 0.2969, eval loss 0.801584780216217
Epoch 248 train loss: 0.3250, eval loss 0.8027665019035339
Epoch 249 train loss: 0.3082, eval loss 0.7997879981994629
Epoch 250 train loss: 0.2732, eval loss 0.8035768270492554
Epoch 251 train loss: 0.2602, eval loss 0.7992032766342163
Epoch 252 train loss: 0.2940, eval loss 0.8000468611717224
Epoch 253 train loss: 0.3235, eval loss 0.8039683103561401
Epoch 254 train loss: 0.3151, eval loss 0.7966246604919434
Epoch 255 train loss: 0.2766, eval loss 0.7986290454864502
Epoch 256 train loss: 0.3140, eval loss 0.7999040484428406
Epoch 257 train loss: 0.3200, eval loss 0.805113673210144
Epoch 258 train loss: 0.2880, eval loss 0.8000678420066833
Epoch 259 train loss: 0.3630, eval loss 0.8052870035171509
Epoch 260 train loss: 0.3096, eval loss 0.7993956208229065
Epoch 261 train loss: 0.2402, eval loss 0.801108717918396
Epoch 262 train loss: 0.3433, eval loss 0.7978273630142212
Epoch 263 train loss: 0.2807, eval loss 0.8001335263252258
Epoch 264 train loss: 0.3202, eval loss 0.7974492907524109
Epoch 265 train loss: 0.2779, eval loss 0.8023646473884583
Epoch 266 train loss: 0.2631, eval loss 0.7997326850891113
Epoch 267 train loss: 0.3044, eval loss 0.8042994141578674
Epoch 268 train loss: 0.3347, eval loss 0.8008028864860535
Epoch 269 train loss: 0.2683, eval loss 0.8025902509689331
Epoch 270 train loss: 0.3227, eval loss 0.8028398156166077
Epoch 271 train loss: 0.3369, eval loss 0.8012770414352417
Epoch 272 train loss: 0.4552, eval loss 0.8017436861991882
Epoch 273 train loss: 0.2408, eval loss 0.800152599811554
Epoch 274 train loss: 0.3176, eval loss 0.8028700947761536
Epoch 275 train loss: 0.2440, eval loss 0.7996659874916077
Epoch 276 train loss: 0.3623, eval loss 0.7979757189750671
Epoch 277 train loss: 0.2842, eval loss 0.8005878329277039
Epoch 278 train loss: 0.2741, eval loss 0.8014857172966003
```

```
Epoch 279 train loss: 0.3524, eval loss 0.8009438514709473
Epoch 280 train loss: 0.3875, eval loss 0.8011255860328674
Epoch 281 train loss: 0.3379, eval loss 0.8046345710754395
Epoch 282 train loss: 0.3065, eval loss 0.8017874360084534
Epoch 283 train loss: 0.2581, eval loss 0.8036415576934814
Epoch 284 train loss: 0.3561, eval loss 0.8061400651931763
Epoch 285 train loss: 0.3774, eval loss 0.800325870513916
Epoch 286 train loss: 0.3220, eval loss 0.800272524356842
Epoch 287 train loss: 0.2724, eval loss 0.8021789193153381
Epoch 288 train loss: 0.2618, eval loss 0.7979452610015869
Epoch 289 train loss: 0.2962, eval loss 0.7995166182518005
Epoch 290 train loss: 0.2828, eval loss 0.8024284839630127
Epoch 291 train loss: 0.2659, eval loss 0.8007172346115112
Epoch 292 train loss: 0.2993, eval loss 0.8094015121459961
Epoch 293 train loss: 0.2479, eval loss 0.8026455044746399
Epoch 294 train loss: 0.3081, eval loss 0.800327718257904
Epoch 295 train loss: 0.2676, eval loss 0.8042691349983215
Epoch 296 train loss: 0.3826, eval loss 0.804906964302063
Epoch 297 train loss: 0.2588, eval loss 0.8032177686691284
Epoch 298 train loss: 0.3081, eval loss 0.7972596287727356
Epoch 299 train loss: 0.3087, eval loss 0.7987984418869019
```

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

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

```
AUROC: 90.71%
F1: 69.18%
Precision: 66.51%
Recall: 72.07%
```

Pytania kontrolne (1 punkt)

1. Wymień 4 najważniejsze twoim zdaniem hiperparametry sieci neuronowej.

   A. epochs - ilość epok w treningu, liczba przepuszczenia danych przez model
   B. liczba warstw sieci
   C. learning_rate - współczynnik uczenia, wskazuje jak mocno aktualizować wagi podczas uczenia
   D. batch_size - liczba danych uzywanych podczas jednej aktualizacji

2. Czy widzisz jakiś problem w użyciu regularyzacji L1 w treningu sieci neuronowych? Czy dropout może twoim zdaniem stanowić alternatywę dla tego rodzaju regularyzacji?

   L1 moze prowadzić do zerowania wag co prowadzi do zaniku informacji. Dropout jest lepszym rozwiązaniem, ponieważ nie zeruje wag, a jedynie wyłącza neurony,

co prowadzi do zmniejszenia zależności między neuronami.

3. Czy użycie innej metryki do wczesnego stopu da taki sam model końcowy? Czemu?

   Uzycie innej metryki wczesnego stopu wpłynie na model końcowy, poniewaz będzie nagradzać lub karać różne zachowania modelu, co prowadzi do innego modelu końcowego.

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, weigh

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

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

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

        optimizer.step()
        optimizer.zero_grad()

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

```
        step_counter += 1

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

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

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/Users/filipdziurdzia/Desktop/Studia/Semestr 5/PSI 2/Artificial-Intelligence-AGH/lab3/lab_3.ipynb Cell 86 line 6
      <a href='vscode-notebook-cell:/Users/filipdziurdzia/Desktop/Studia/Semestr%205/PSI%202/Artificial-Intelligence-AGH/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=0'>1</a> import time
      <a href='vscode-notebook-cell:/Users/filipdziurdzia/Desktop/Studia/Semestr%205/PSI%202/Artificial-Intelligence-AGH/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=2'>3</a> model = NormalizingMLP(
      <a href='vscode-notebook-cell:/Users/filipdziurdzia/Desktop/Studia/Semestr%205/PSI%202/Artificial-Intelligence-AGH/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=3'>4</a>     input_size=X_train.shape[1],
      <a href='vscode-notebook-cell:/Users/filipdziurdzia/Desktop/Studia/Semestr%205/PSI%202/Artificial-Intelligence-AGH/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=4'>5</a>     dropout_p=dropout_p
----> <a href='vscode-notebook-cell:/Users/filipdziurdzia/Desktop/Studia/Semestr%205/PSI%202/Artificial-Intelligence-AGH/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=5'>6</a> ).to('cuda')
      <a href='vscode-notebook-cell:/Users/filipdziurdzia/Desktop/Studia/Semestr%205/PSI%202/Artificial-Intelligence-AGH/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=7'>8</a> optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate, weight_decay=1e-4)
     <a href='vscode-notebook-cell:/Users/filipdziurdzia/Desktop/Studia/Semestr%205/PSI%202/Artificial-Intelligence-AGH/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=9'>10</a> # note that we are using loss function with sigmoid built in

File ~/Desktop/Studia/Semestr 5/PSI 2/Artificial-Intelligence-AGH/PSI/lib/python3.11/site-packages/torch/nn/modules/module.py:1160, in Module.to(self, *args, **kwargs)
   1156         return t.to(device, dtype if t.is_floating_point() or t.is_complex() else None,
   1157                     non_blocking, memory_format=convert_to_format)
   1158     return t.to(device, dtype if t.is_floating_point() or t.is_complex() else None, non_blocking)
-> 1160 return self._apply(convert)

File ~/Desktop/Studia/Semestr 5/PSI 2/Artificial-Intelligence-AGH/PSI/lib/python3.11/site-packages/torch/nn/modules/module.py:810, in Module._apply(self, fn, recurse)
    808 if recurse:
    809     for module in self.children():
--> 810         module._apply(fn)
```

```
    812 def compute_should_use_set_data(tensor, tensor_applied):
    813     if torch._has_compatible_shallow_copy_type(tensor, tensor_appl
ied):
    814         # If the new tensor has compatible tensor type as the exis
ting tensor,
    815         # the current behavior is to change the tensor in-place us
ing `.data =`,
   (...)
    820         # global flag to let the user control whether they want th
e future
    821         # behavior of overwriting the existing tensor or not.

File ~/Desktop/Studia/Semestr 5/PSI 2/Artificial-Intelligence-AGH/PSI/lib/
python3.11/site-packages/torch/nn/modules/module.py:810, in Module._apply(
self, fn, recurse)
    808 if recurse:
    809     for module in self.children():
--> 810         module._apply(fn)
    812 def compute_should_use_set_data(tensor, tensor_applied):
    813     if torch._has_compatible_shallow_copy_type(tensor, tensor_appl
ied):
    814         # If the new tensor has compatible tensor type as the exis
ting tensor,
    815         # the current behavior is to change the tensor in-place us
ing `.data =`,
   (...)
    820         # global flag to let the user control whether they want th
e future
    821         # behavior of overwriting the existing tensor or not.

File ~/Desktop/Studia/Semestr 5/PSI 2/Artificial-Intelligence-AGH/PSI/lib/
python3.11/site-packages/torch/nn/modules/module.py:833, in Module._apply(
self, fn, recurse)
    829 # Tensors stored in modules are graph leaves, and we don't want to
    830 # track autograd history of `param_applied`, so we have to use
    831 # `with torch.no_grad():`
    832 with torch.no_grad():
--> 833     param_applied = fn(param)
    834 should_use_set_data = compute_should_use_set_data(param, param_app
lied)
    835 if should_use_set_data:

File ~/Desktop/Studia/Semestr 5/PSI 2/Artificial-Intelligence-AGH/PSI/lib/
python3.11/site-packages/torch/nn/modules/module.py:1158, in Module.to.<lo
cals>.convert(t)
   1155 if convert_to_format is not None and t.dim() in (4, 5):
   1156     return t.to(device, dtype if t.is_floating_point() or t.is_com
plex() else None,
   1157                 non_blocking, memory_format=convert_to_format)
-> 1158 return t.to(device, dtype if t.is_floating_point() or t.is_complex
() else None, non_blocking)

File ~/Desktop/Studia/Semestr 5/PSI 2/Artificial-Intelligence-AGH/PSI/lib/
```

```
python3.11/site-packages/torch/cuda/__init__.py:289, in _lazy_init()
    284     raise RuntimeError(
    285         "Cannot re-initialize CUDA in forked subprocess. To use CU
DA with "
    286         "multiprocessing, you must use the 'spawn' start method"
    287     )
    288 if not hasattr(torch._C, "_cuda_getDeviceCount"):
--> 289     raise AssertionError("Torch not compiled with CUDA enabled")
    290 if _cudart is None:
    291     raise AssertionError(
    292         "libcudart functions unavailable. It looks like you have a
broken build?"
    293     )

AssertionError: Torch not compiled with CUDA enabled
```

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

Dla zainteresowanych polecamy tę serie artykułów

Zadanie dla chętnych

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

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

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

Przeszukaj wybrane zakresy dla hiperparametrów:

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

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

Przydatne materiały:

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

In [ ]: