Sieci neuronowe

Wstęp

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

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

Wykorzystywane biblioteki

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

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

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

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

Konfiguracja własnego komputera

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

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

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

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

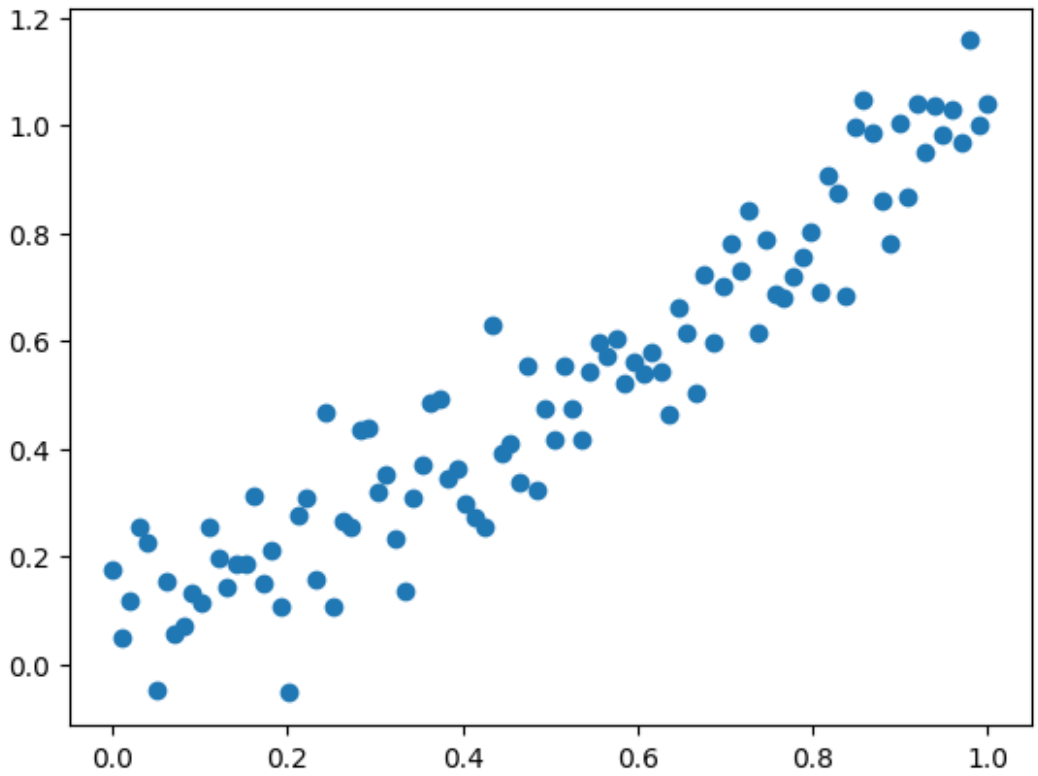
```
/bin/bash: line 1: conda: command not found
```

Wprowadzenie

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

```
from typing import Tuple, Dict

import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)

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

plt.scatter(x, y)
```

```
<matplotlib.collections.PathCollection at 0x7b942d781e10>
```



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

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

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

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

Zadanie 1 (0.5 punkt)

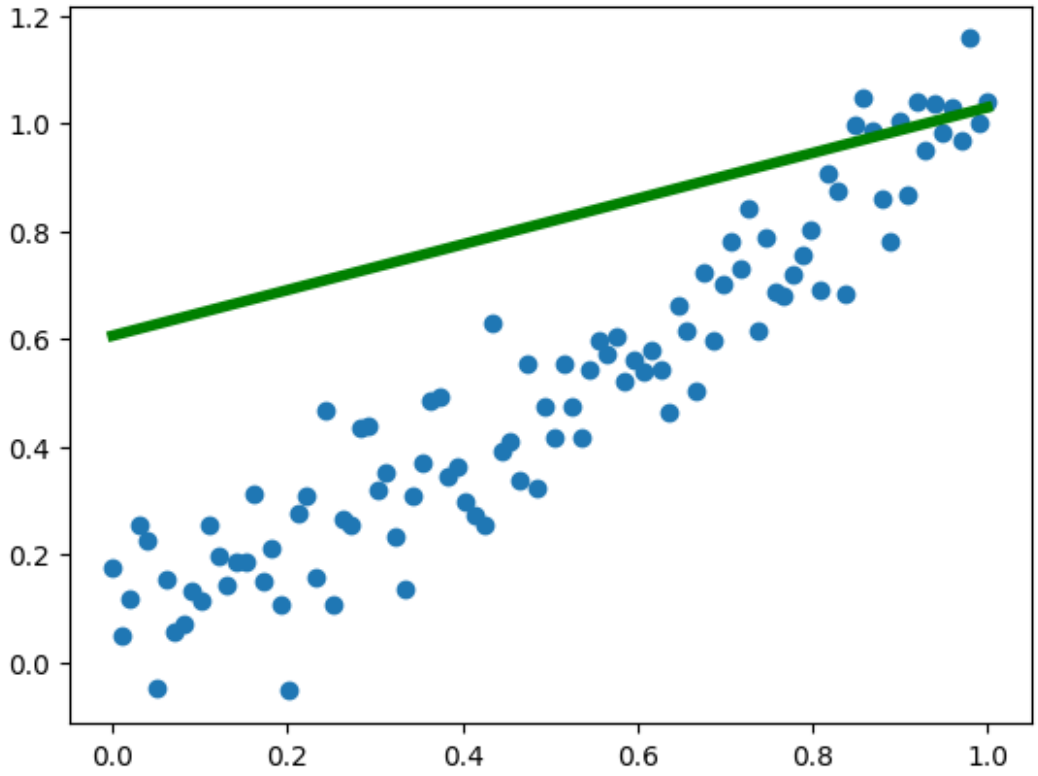
Uzupełnij kod funkcji `mse`, obliczającej błąd średniokwadratowy. Wykorzystaj Numpy'a w celu wektoryzacji obliczeń dla wydajności.

```
def mse(y: np.ndarray, y_hat: np.ndarray) -> float:
  return np.sum(np.square(y-y_hat))/len(y)
    # result = 0.0
    # for i in range(len(y)):
    #     result += (y[i] - y_hat[i])**2
    # return result/len(y)
```

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

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

```
MSE: 0.133
```

```
[<matplotlib.lines.Line2D at 0x7b942d60c040>]
```



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

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

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

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

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

Przyglądając się temu równaniu widzimy, że:

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

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

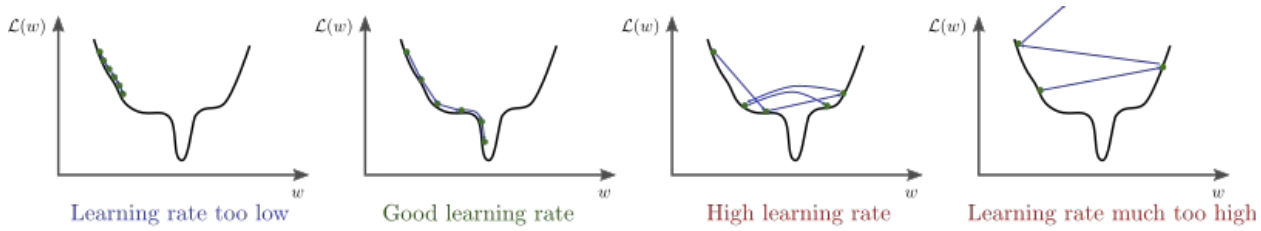
Stosując powyższe do optymalizacji, mamy:

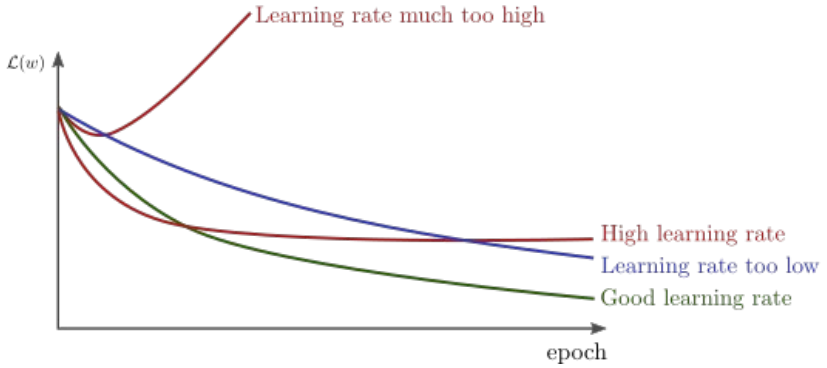
$$\large x_{t+1} = x_{t} - \alpha * \frac{f(x)}{dx} $$

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

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

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





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

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

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

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

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

Pełna aktualizacja to zatem:

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

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

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

Obliczenie pochodnych cząstkowych ze względu na każdy

Zadanie 2 (1.5 punkty)

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

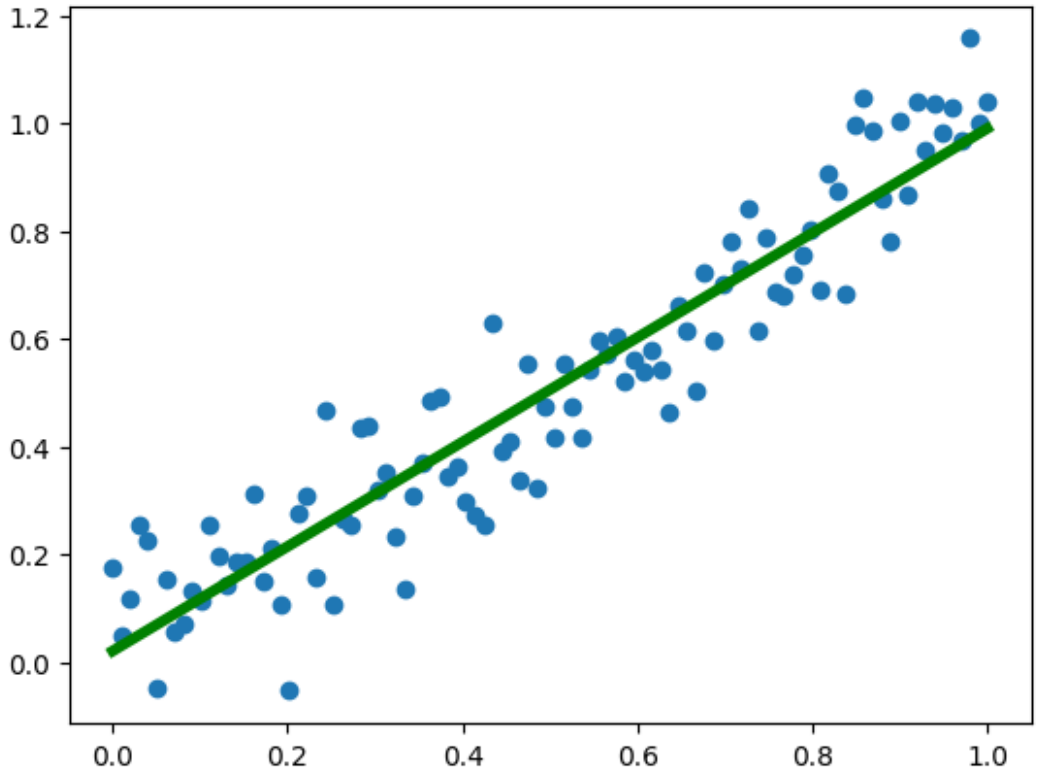
```
def optimize(
    x: np.ndarray, y: np.ndarray, a: float, b: float, learning_rate:
```

```
float = 0.1
):
    y_hat = a * x + b
    errors = y - y_hat
    # implement me!
    n = y.shape[0]

    # @ - mnożenie macierzy
    #  errors @ x iloczyn skalarny = sigma ze wzoru
    new_a = a + learning_rate *2 /n * (errors @ x)
    new_b = b + learning_rate *2 / n* np.sum(errors)


    return new_a, new_b

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

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

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

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

```
[<matplotlib.lines.Line2D at 0x7b942d68d7e0>]
```



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

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

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

Wprowadzenie do PyTorcha

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

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

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

Najważniejsze moduły:

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

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

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

# elementwise sum
print(ones + noise)

# elementwise multiplication
print(ones * noise)

# dot product
print(ones @ noise)

# dot product = 1*1 *10
print(ones @ ones)
```

```
tensor([1.5514, 1.6103, 1.8203, 1.5919, 1.2481, 1.5935, 1.5476,
1.9022, 1.8845,
        1.4277])
tensor([0.5514, 0.6103, 0.8203, 0.5919, 0.2481, 0.5935, 0.5476,
0.9022, 0.8845,
        0.4277])
tensor(6.1775)
tensor(10.)
```

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

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

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

```
(tensor([0.0646], requires_grad=True), tensor([0.3717],
requires_grad=True))
```

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

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

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

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

```
tensor(0.0902, dtype=torch.float64, grad_fn=<MseLossBackward0>)
```

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

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

print(a.grad)
```

```
tensor([-0.2559])
```

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

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

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

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

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

```
    # compute gradients
    loss.backward()

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

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

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

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

```
step 0 loss:  tensor(0.0902, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 100 loss:  tensor(0.0140, dtype=torch.float64,
grad_fn=<MseLossBackward0>)
step 200 loss:  tensor(0.0103, 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.

```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: 0.3491
step 100 loss: 0.0110
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>)
```

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

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

Uwaga - PyTorch wymaga tensora klas będącego liczbami zmiennoprzecinkowymi!

Zbiór danych

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

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

```
!wget https://archive.ics.uci.edu/ml/machine-learning-
databases/adult/adult.data --no-check-certificate

--2023-11-13 04:04:16--  https://archive.ics.uci.edu/ml/machine-
learning-databases/adult/adult.data
Resolving archive.ics.uci.edu (archive.ics.uci.edu)... 128.195.10.252
Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|
128.195.10.252|:443... connected.
WARNING: cannot verify archive.ics.uci.edu's certificate, issued by
‘CN=InCommon RSA Server CA,OU=InCommon,O=Internet2,L=Ann
Arbor,ST=MI,C=US’:
  Issued certificate has expired.
HTTP request sent, awaiting response... 200 OK
Length: unspecified
Saving to: ‘adult.data.1’

adult.data.1            [ <=>                ]   3.79M  --.-KB/s    in
0.1s

2023-11-13 04:04:16 (25.3 MB/s) - ‘adult.data.1’ saved [3974305]


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()
df.shape
```

```
(32561, 15)
```

```
# attribution: https://www.kaggle.com/code/royshih23/topic7-
classification-in-python
```

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

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

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

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

train_valid_size = 0.2

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

```
    random_state=0,
    shuffle=True,
    stratify=y_train
)

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

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

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

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

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

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

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

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

X_train.shape, y_train.shape
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/
_encoders.py:868: FutureWarning: `sparse` was renamed to
`sparse_output` in version 1.2 and will be removed in 1.4.
`sparse_output` is ignored unless you leave `sparse` to its default
```

```
value.
  warnings.warn(

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

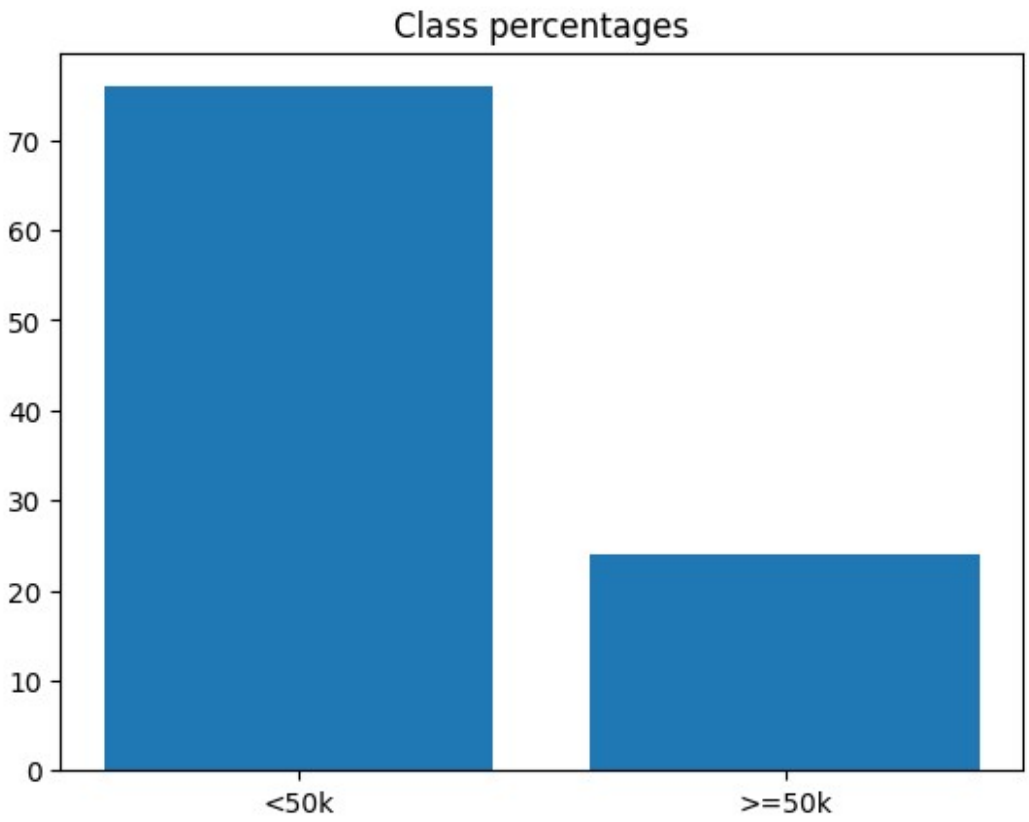
Uwaga co do typów - PyTorchu wszystko w sieci neuronowej musi być typu `float32`. W szczególności trzeba uważać na konwersje z Numpy'a, który używa domyślnie typu `float64`. Może ci się przydać metoda `.float()`.

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

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

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

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

X_valid.shape
# type(X_valid)
# y_train

torch.Size([5210, 108])
```

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

```
import matplotlib.pyplot as plt

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

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



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

Zadanie 3 (1 punkt)

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

Sama sieć składa się z 2 elementów:

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

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

```
from torch.optim import SGD

learning_rate = 1e-3

# in_features - liczba cech, out fatures - liczba wyników (1: 0 lub 1)
model = nn.Linear(in_features = X_train.shape[1], out_features =1)
activation = nn.Sigmoid()
```

```
optimizer = SGD(params=model.parameters(), lr = learning_rate)
loss_fn = nn.BCELoss()
#implement me!

for i in range(3000):
  y_pred = activation(model(X_train))
  loss = loss_fn(y_pred, y_train)
  loss.backward()
  optimizer.step()
  optimizer.zero_grad()
  if i % 100 == 0:
    print(f"step {i} loss: {loss.item():.4f}")


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

```
step 0 loss: 0.6834
step 100 loss: 0.6525
step 200 loss: 0.6268
step 300 loss: 0.6050
step 400 loss: 0.5866
step 500 loss: 0.5708
step 600 loss: 0.5572
step 700 loss: 0.5453
step 800 loss: 0.5348
step 900 loss: 0.5255
step 1000 loss: 0.5172
step 1100 loss: 0.5098
step 1200 loss: 0.5030
step 1300 loss: 0.4968
step 1400 loss: 0.4912
step 1500 loss: 0.4860
step 1600 loss: 0.4812
step 1700 loss: 0.4767
step 1800 loss: 0.4725
step 1900 loss: 0.4686
step 2000 loss: 0.4649
step 2100 loss: 0.4614
step 2200 loss: 0.4582
step 2300 loss: 0.4551
step 2400 loss: 0.4521
step 2500 loss: 0.4493
step 2600 loss: 0.4467
step 2700 loss: 0.4441
step 2800 loss: 0.4417
step 2900 loss: 0.4394
final loss: 0.4372
```

Teraz trzeba sprawdzić, jak poszło naszej sieci. W PyTorchu sieć pracuje zawsze w jednym z dwóch trybów: treningowym lub ewaluacyjnym (predykcyjnym). Ten drugi wyłącza niektóre

mechanizmy, które są używane tylko podczas treningu, w szczególności regularyzację dropout. Do przełączania służą metody modelu `.train()` i `.eval()`.

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

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

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

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

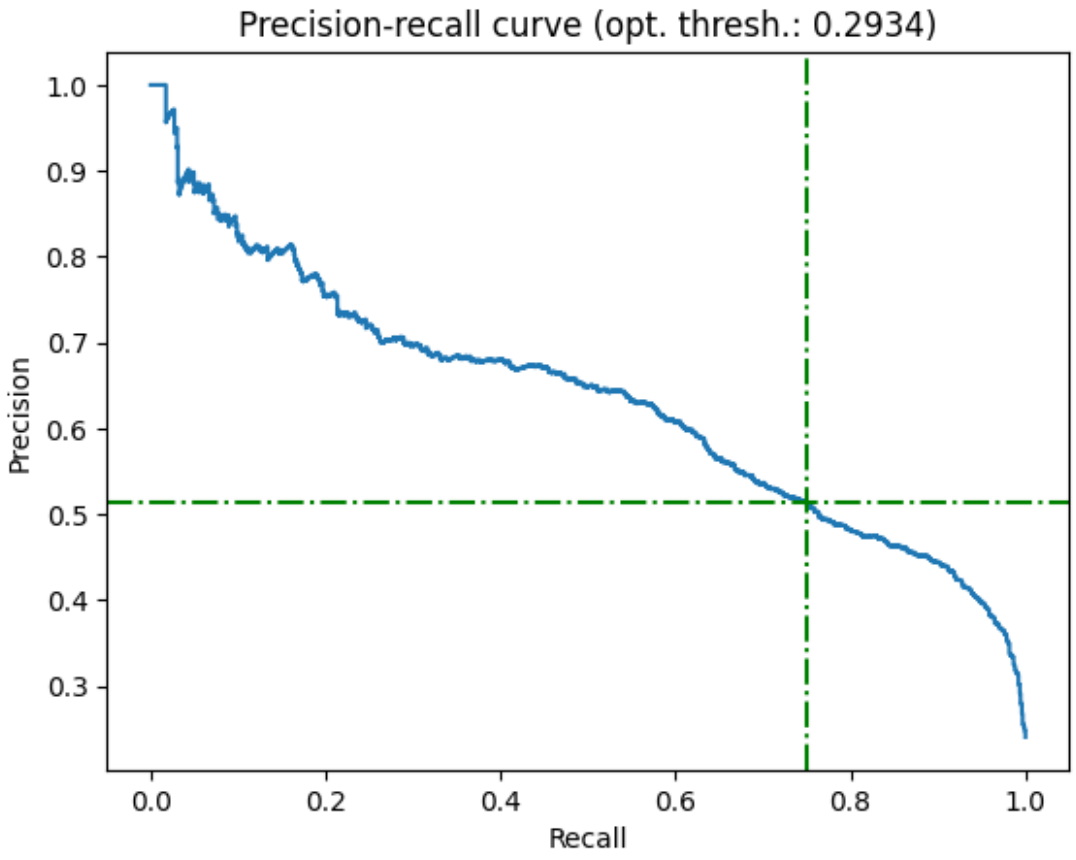
```
AUROC: 84.87%
```

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

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

```
from sklearn.metrics import PrecisionRecallDisplay


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

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

    return optimal_idx, optimal_threshold


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

```
recalls, thresholds)

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

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

plot_precision_recall_curve(y_valid, y_pred_valid_score)
```



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

Sieci neuronowe

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

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

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

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

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

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

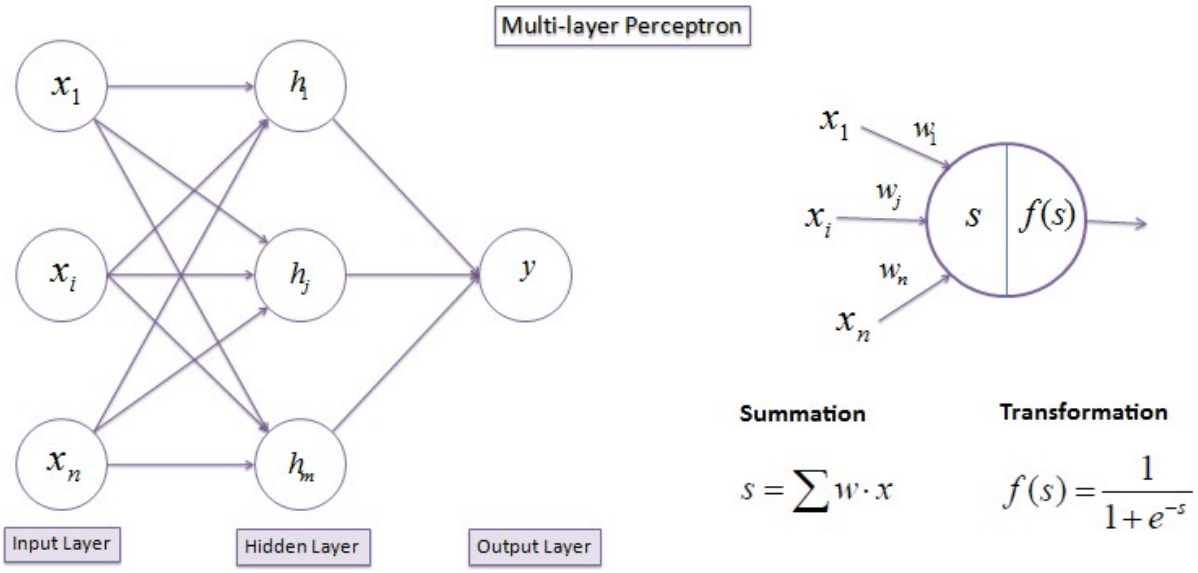
Sieci MLP

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

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

1_x-3NGQv0pRIab8xDT-f_Hg.png

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

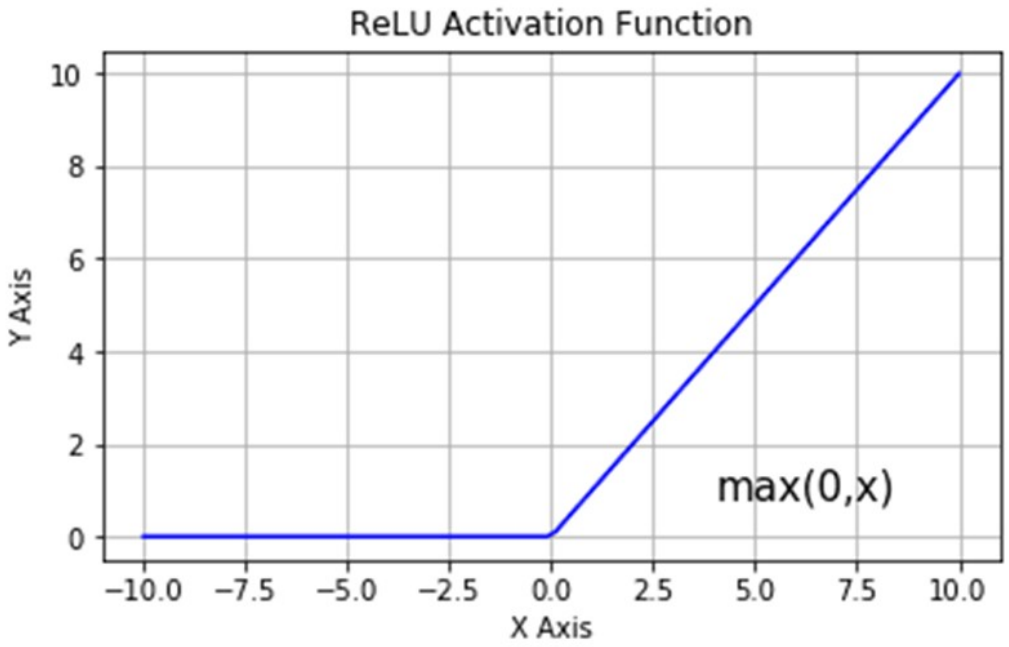


Zapisane matematycznie MLP to: $ h_1 = f_1(x) \ h_2 = f_2(h_1) \ h_3 = f_3(h_2) \ ... h_n = f_n(h_{n-1}) $ gdzie x to wejście f_i to funkcja aktywacji i-tej warstwy, a h_i to wyjście i-tej warstwy, nazywane **ukrytą reprezentacją (hidden representation)**, lub *latent representation*. Nazwa bierze się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

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

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

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



MLP w PyTorchu

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

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

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

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

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

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

Zadanie 4 (1 punkt)

Uzupełnij implementację 3-warstwowej sieci MLP. Użyj rozmiarów:

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

Użyj funkcji aktywacji ReLU.

Przydatne klasy:

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

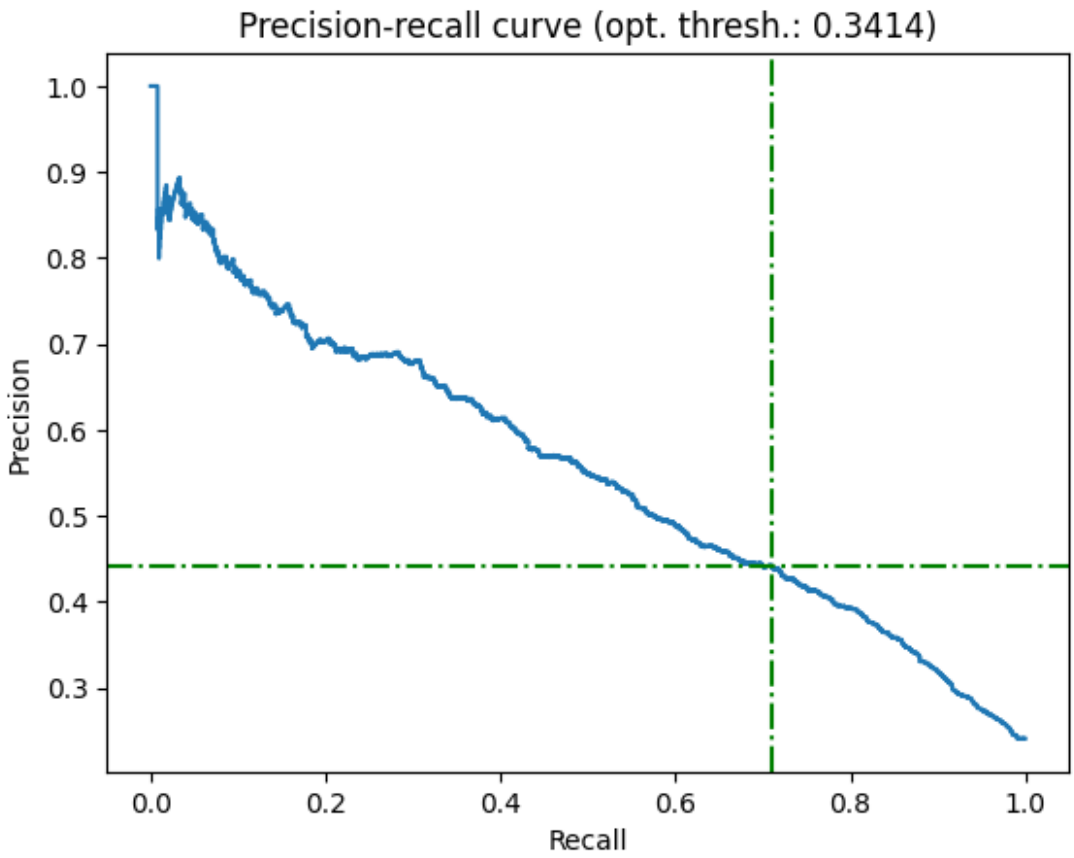
```
from torch import sigmoid

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

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

    def forward(self, x):
        # implement me!
        x = self.relu(self.mlp(x))
        x = self.relu(self.hidden(x))
        x=self.out(x)

        return x

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

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

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

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

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

    optimizer.step()
    optimizer.zero_grad()

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

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

```
Epoch 0 train loss: 0.6898
Epoch 200 train loss: 0.6692
Epoch 400 train loss: 0.6513
Epoch 600 train loss: 0.6357
Epoch 800 train loss: 0.6218
Epoch 1000 train loss: 0.6094
Epoch 1200 train loss: 0.5984
Epoch 1400 train loss: 0.5884
Epoch 1600 train loss: 0.5794
Epoch 1800 train loss: 0.5712
final loss: 0.5638
```

```
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: 75.89%
```



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

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

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

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

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

Dodatkowo ryzyko przeuczenia można zmniejszyć, używając mniejszej stałej uczącej.

Zadanie 5 (1 punkt)

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

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

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

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

```
from typing import Optional

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

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

      # y_pred_proba = torch.sigmoid(model(X))
      y_pred_proba = model(X)
      loss = loss_fn(y_pred_proba, y)

    auroc = roc_auc_score(y.cpu().numpy(), y_pred_proba.cpu().numpy())
```

```
    if threshold is None:
        precisions, recalls, thresholds =
precision_recall_curve(y.cpu().numpy(), y_pred_proba.cpu().numpy())
        f1 = 2*(precisions*recalls)/ (precisions + recalls)
        # optimal treshold index = np.argmax(f1)
        threshold = thresholds[np.argmax(f1)]
# y_pred = how many y_proba >= treshold

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

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

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

Zadanie 6 (1 punkt)

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

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

        self.mlp = nn.Linear(input_size, 256)
        self.hidden = nn.Linear(256, 128)
        self.out = nn.Linear(128, 1)
        self.relu = nn.ReLU()

        self.dropout = nn.Dropout(p=dropout_p)

    def forward(self, x):
        # implement me!
        x = self.relu(self.mlp(x))
        x = self.dropout(x)
        x = self.relu(self.hidden(x))
        x = self.dropout(x)
        x=self.out(x)

        return x
```

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

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

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

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

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

Na razie użyjemy jednak minibatch SGD.

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

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

```
from torch.utils.data import Dataset

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

        self.data = data
        self.y = y

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

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

Zadanie 7 (2 punkty)

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

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

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

```
from copy import deepcopy

from torch.utils.data import DataLoader


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

early_stopping_patience = 4

model = RegularizedMLP(
    input_size=X_train.shape[1],
    dropout_p=dropout_p
)
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=learning_rate,
    weight_decay=l2_reg #here is l2 added
)
loss_fn = torch.nn.BCEWithLogitsLoss()

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

steps_without_improvement = 0

best_val_loss = np.inf
best_model = None
best_threshold = None

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

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

```
        # model training
        # implement me!

        # Forward pass
        outputs = model(X_batch)
        # Loss calculation
        loss = loss_fn(outputs, y_batch)
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        # Optimalization
        optimizer.step()

    # model evaluation, early stopping
    # implement me!

    model.eval()
    valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
    # best models and results
    if valid_metrics.get("loss") < best_val_loss:
      best_val_loss = valid_metrics.get("loss")
      best_model = model
      best_threshold = valid_metrics.get("treshold")

      steps_without_improvement =0
    else:
      steps_without_improvement += 1

      if steps_without_improvement == early_stopping_patience:
        break

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

```
Epoch 0 train loss: 0.6945, eval loss 0.6914260387420654
Epoch 1 train loss: 0.6736, eval loss 0.6738576889038086
Epoch 2 train loss: 0.6590, eval loss 0.6582309007644653

<ipython-input-133-91752ac91c01>:25: RuntimeWarning: invalid value
encountered in divide
  f1 = 2*(precisions*recalls)/ (precisions + recalls)

Epoch 3 train loss: 0.6420, eval loss 0.6442409157752991

<ipython-input-133-91752ac91c01>:25: RuntimeWarning: invalid value
encountered in divide
  f1 = 2*(precisions*recalls)/ (precisions + recalls)

Epoch 4 train loss: 0.6286, eval loss 0.6317198276519775
```

```
<ipython-input-133-91752ac91c01>:25: RuntimeWarning: invalid value
encountered in divide
  f1 = 2*(precisions*recalls)/ (precisions + recalls)

Epoch 5 train loss: 0.6175, eval loss 0.6203551888465881

<ipython-input-133-91752ac91c01>:25: RuntimeWarning: invalid value
encountered in divide
  f1 = 2*(precisions*recalls)/ (precisions + recalls)

Epoch 6 train loss: 0.6034, eval loss 0.6100955009460449

<ipython-input-133-91752ac91c01>:25: RuntimeWarning: invalid value
encountered in divide
  f1 = 2*(precisions*recalls)/ (precisions + recalls)

Epoch 7 train loss: 0.5947, eval loss 0.6007881760597229

<ipython-input-133-91752ac91c01>:25: RuntimeWarning: invalid value
encountered in divide
  f1 = 2*(precisions*recalls)/ (precisions + recalls)

Epoch 8 train loss: 0.5845, eval loss 0.5923198461532593

<ipython-input-133-91752ac91c01>:25: RuntimeWarning: invalid value
encountered in divide
  f1 = 2*(precisions*recalls)/ (precisions + recalls)

Epoch 9 train loss: 0.5749, eval loss 0.5845517516136169
Epoch 10 train loss: 0.5689, eval loss 0.5775198936462402
Epoch 11 train loss: 0.5584, eval loss 0.5711037516593933
Epoch 12 train loss: 0.5630, eval loss 0.5652040839195251
Epoch 13 train loss: 0.5498, eval loss 0.5597771406173706
Epoch 14 train loss: 0.5500, eval loss 0.5547919273376465
Epoch 15 train loss: 0.5308, eval loss 0.5501497387886047
Epoch 16 train loss: 0.5338, eval loss 0.5458751916885376
Epoch 17 train loss: 0.5376, eval loss 0.5418037176132202
Epoch 18 train loss: 0.5363, eval loss 0.5379738807678223
Epoch 19 train loss: 0.5194, eval loss 0.5343407392501831
Epoch 20 train loss: 0.5251, eval loss 0.5308569073677063
Epoch 21 train loss: 0.5115, eval loss 0.527452290058136
Epoch 22 train loss: 0.5285, eval loss 0.5241726636886597
Epoch 23 train loss: 0.5161, eval loss 0.5209476947784424
Epoch 24 train loss: 0.5095, eval loss 0.517772912979126
Epoch 25 train loss: 0.5046, eval loss 0.5146156549453735
Epoch 26 train loss: 0.5067, eval loss 0.5114591717720032
Epoch 27 train loss: 0.5171, eval loss 0.5083288550376892
Epoch 28 train loss: 0.5005, eval loss 0.5051959753036499
Epoch 29 train loss: 0.4951, eval loss 0.5019993185997009
Epoch 30 train loss: 0.5037, eval loss 0.49882185459136963
Epoch 31 train loss: 0.4966, eval loss 0.4956596493721008
```

```
Epoch 32 train loss: 0.4982, eval loss 0.4924232065677643
Epoch 33 train loss: 0.4874, eval loss 0.489203542470932
Epoch 34 train loss: 0.4699, eval loss 0.48598018288612366
Epoch 35 train loss: 0.4898, eval loss 0.48270073533058167
Epoch 36 train loss: 0.4803, eval loss 0.47941678762435913
Epoch 37 train loss: 0.4779, eval loss 0.4761411249637604
Epoch 38 train loss: 0.4780, eval loss 0.47286033630371094
Epoch 39 train loss: 0.4838, eval loss 0.46958306431770325
Epoch 40 train loss: 0.4690, eval loss 0.46627944707870483
Epoch 41 train loss: 0.4582, eval loss 0.4630068242549896
Epoch 42 train loss: 0.4538, eval loss 0.45972466468811035
Epoch 43 train loss: 0.4558, eval loss 0.45644432306289673
Epoch 44 train loss: 0.4584, eval loss 0.4531761705875397
Epoch 45 train loss: 0.4661, eval loss 0.4499596059322357
Epoch 46 train loss: 0.4525, eval loss 0.4467327296733856
Epoch 47 train loss: 0.4467, eval loss 0.44352424144744873
Epoch 48 train loss: 0.4518, eval loss 0.44038206338882446
Epoch 49 train loss: 0.4657, eval loss 0.4372619390487671
Epoch 50 train loss: 0.4597, eval loss 0.43416866660118103
Epoch 51 train loss: 0.4380, eval loss 0.431109219789505
Epoch 52 train loss: 0.4387, eval loss 0.4281149208545685
Epoch 53 train loss: 0.4339, eval loss 0.42518138885498047
Epoch 54 train loss: 0.4250, eval loss 0.4222955107688904
Epoch 55 train loss: 0.4418, eval loss 0.4194469451904297
Epoch 56 train loss: 0.4173, eval loss 0.4166627526283264
Epoch 57 train loss: 0.4337, eval loss 0.4139425456523895
Epoch 58 train loss: 0.4098, eval loss 0.41126275062561035
Epoch 59 train loss: 0.4238, eval loss 0.4086705446243286
Epoch 60 train loss: 0.4341, eval loss 0.4061044156551361
Epoch 61 train loss: 0.4256, eval loss 0.40363070368766785
Epoch 62 train loss: 0.4330, eval loss 0.4011969566345215
Epoch 63 train loss: 0.4160, eval loss 0.39886435866355896
Epoch 64 train loss: 0.4285, eval loss 0.3966013789176941
Epoch 65 train loss: 0.4157, eval loss 0.3943808674812317
Epoch 66 train loss: 0.4217, eval loss 0.39224252104759216
Epoch 67 train loss: 0.4234, eval loss 0.390191912651062
Epoch 68 train loss: 0.4088, eval loss 0.38819199800491333
Epoch 69 train loss: 0.4139, eval loss 0.38624855875968933
Epoch 70 train loss: 0.4181, eval loss 0.3843855559825897
Epoch 71 train loss: 0.4176, eval loss 0.38259559869766235
Epoch 72 train loss: 0.4058, eval loss 0.3808835446834564
Epoch 73 train loss: 0.3954, eval loss 0.3792148530483246
Epoch 74 train loss: 0.3991, eval loss 0.37762805819511414
Epoch 75 train loss: 0.3998, eval loss 0.3761296570301056
Epoch 76 train loss: 0.4270, eval loss 0.3746716380119324
Epoch 77 train loss: 0.4140, eval loss 0.37326961755752563
Epoch 78 train loss: 0.4048, eval loss 0.37195953726768494
Epoch 79 train loss: 0.4134, eval loss 0.37069380283355713
Epoch 80 train loss: 0.3957, eval loss 0.36946961283683777
```

```
Epoch 81 train loss: 0.3967, eval loss 0.36831578612327576
Epoch 82 train loss: 0.4345, eval loss 0.36717551946640015
Epoch 83 train loss: 0.3889, eval loss 0.36609938740730286
Epoch 84 train loss: 0.4120, eval loss 0.3650866746902466
Epoch 85 train loss: 0.3968, eval loss 0.36413508653640747
Epoch 86 train loss: 0.4038, eval loss 0.3632333278656006
Epoch 87 train loss: 0.4151, eval loss 0.3623620569705963
Epoch 88 train loss: 0.4031, eval loss 0.36152908205986023
Epoch 89 train loss: 0.3720, eval loss 0.36073294281959534
Epoch 90 train loss: 0.4009, eval loss 0.3599797785282135
Epoch 91 train loss: 0.3815, eval loss 0.35926634073257446
Epoch 92 train loss: 0.3974, eval loss 0.3586030602455139
Epoch 93 train loss: 0.3918, eval loss 0.35794249176979065
Epoch 94 train loss: 0.4022, eval loss 0.3572889268398285
Epoch 95 train loss: 0.3823, eval loss 0.35670167207717896
Epoch 96 train loss: 0.4133, eval loss 0.35614049434661865
Epoch 97 train loss: 0.4041, eval loss 0.3556019365787506
Epoch 98 train loss: 0.3883, eval loss 0.3550717830657959
Epoch 99 train loss: 0.3660, eval loss 0.3545735776424408
Epoch 100 train loss: 0.3868, eval loss 0.35409119725227356
Epoch 101 train loss: 0.3834, eval loss 0.35362592339515686
Epoch 102 train loss: 0.4042, eval loss 0.3531791567802429
Epoch 103 train loss: 0.3899, eval loss 0.3527553677558899
Epoch 104 train loss: 0.4014, eval loss 0.3523525297641754
Epoch 105 train loss: 0.3872, eval loss 0.35194602608680725
Epoch 106 train loss: 0.3609, eval loss 0.351563423871994
Epoch 107 train loss: 0.4077, eval loss 0.35118556022644043
Epoch 108 train loss: 0.4025, eval loss 0.3508109152317047
Epoch 109 train loss: 0.3920, eval loss 0.3504753112792969
Epoch 110 train loss: 0.3954, eval loss 0.3501528203487396
Epoch 111 train loss: 0.4016, eval loss 0.34982913732528687
Epoch 112 train loss: 0.4097, eval loss 0.34952178597450256
Epoch 113 train loss: 0.4150, eval loss 0.3492019474506378
Epoch 114 train loss: 0.3734, eval loss 0.3489009737968445
Epoch 115 train loss: 0.3883, eval loss 0.34860727190971375
Epoch 116 train loss: 0.3946, eval loss 0.34830886125564575
Epoch 117 train loss: 0.3829, eval loss 0.3480370342731476
Epoch 118 train loss: 0.3975, eval loss 0.347773015499115
Epoch 119 train loss: 0.3648, eval loss 0.3475090265274048
Epoch 120 train loss: 0.4024, eval loss 0.3472457230091095
Epoch 121 train loss: 0.4196, eval loss 0.3469940423965454
Epoch 122 train loss: 0.3992, eval loss 0.34674355387687683
Epoch 123 train loss: 0.3771, eval loss 0.34649956226348877
Epoch 124 train loss: 0.3718, eval loss 0.3462553322315216
Epoch 125 train loss: 0.3941, eval loss 0.3460225760936737
Epoch 126 train loss: 0.3787, eval loss 0.3457861542701721
Epoch 127 train loss: 0.3877, eval loss 0.34557902812957764
Epoch 128 train loss: 0.3804, eval loss 0.3453572988510132
Epoch 129 train loss: 0.3836, eval loss 0.3451259732246399
```

```
Epoch 130 train loss: 0.4066, eval loss 0.34491029381752014
Epoch 131 train loss: 0.4096, eval loss 0.34470969438552856
Epoch 132 train loss: 0.3913, eval loss 0.3444884717464447
Epoch 133 train loss: 0.3979, eval loss 0.34428972005844116
Epoch 134 train loss: 0.3912, eval loss 0.3440838158130646
Epoch 135 train loss: 0.3686, eval loss 0.3438931405544281
Epoch 136 train loss: 0.3830, eval loss 0.34371036291122437
Epoch 137 train loss: 0.4063, eval loss 0.34351205825805664
Epoch 138 train loss: 0.3761, eval loss 0.34332069754600525
Epoch 139 train loss: 0.3595, eval loss 0.3431326150894165
Epoch 140 train loss: 0.3677, eval loss 0.34294214844703674
Epoch 141 train loss: 0.4126, eval loss 0.3427544832229614
Epoch 142 train loss: 0.4120, eval loss 0.3425801694393158
Epoch 143 train loss: 0.3983, eval loss 0.34239301085472107
Epoch 144 train loss: 0.3505, eval loss 0.3421890139579773
Epoch 145 train loss: 0.3611, eval loss 0.3419947326183319
Epoch 146 train loss: 0.3468, eval loss 0.3418259918689728
Epoch 147 train loss: 0.3889, eval loss 0.3416353464126587
Epoch 148 train loss: 0.3675, eval loss 0.34145480394363403
Epoch 149 train loss: 0.3660, eval loss 0.3412906527519226
Epoch 150 train loss: 0.3708, eval loss 0.3411174416542053
Epoch 151 train loss: 0.4003, eval loss 0.3409281075000763
Epoch 152 train loss: 0.3605, eval loss 0.34075573086738586
Epoch 153 train loss: 0.3937, eval loss 0.340590238571167
Epoch 154 train loss: 0.3991, eval loss 0.3404347896575928
Epoch 155 train loss: 0.3952, eval loss 0.34027302265167236
Epoch 156 train loss: 0.3764, eval loss 0.34011954069137573
Epoch 157 train loss: 0.3912, eval loss 0.3399837911128998
Epoch 158 train loss: 0.3754, eval loss 0.33981892466545105
Epoch 159 train loss: 0.4064, eval loss 0.33966466784477234
Epoch 160 train loss: 0.3669, eval loss 0.3395030200481415
Epoch 161 train loss: 0.3609, eval loss 0.33936285972595215
Epoch 162 train loss: 0.3845, eval loss 0.33918237686157227
Epoch 163 train loss: 0.3897, eval loss 0.33902981877326965
Epoch 164 train loss: 0.4114, eval loss 0.3388780355453491
Epoch 165 train loss: 0.3655, eval loss 0.33872923254966736
Epoch 166 train loss: 0.3689, eval loss 0.3385778069496155
Epoch 167 train loss: 0.3717, eval loss 0.3384391665458679
Epoch 168 train loss: 0.3873, eval loss 0.3382900655269623
Epoch 169 train loss: 0.3917, eval loss 0.33812907338142395
Epoch 170 train loss: 0.4001, eval loss 0.3379793167114258
Epoch 171 train loss: 0.4017, eval loss 0.33784738183021545
Epoch 172 train loss: 0.4100, eval loss 0.3377035856246948
Epoch 173 train loss: 0.3845, eval loss 0.3375672698020935
Epoch 174 train loss: 0.3822, eval loss 0.33741626143455505
Epoch 175 train loss: 0.3834, eval loss 0.3372737467288971
Epoch 176 train loss: 0.3938, eval loss 0.33714643120765686
Epoch 177 train loss: 0.4021, eval loss 0.33701860904693604
Epoch 178 train loss: 0.4122, eval loss 0.3368891477584839
```

```
Epoch 179 train loss: 0.3843, eval loss 0.33674874901771545
Epoch 180 train loss: 0.3894, eval loss 0.33660638332366943
Epoch 181 train loss: 0.3955, eval loss 0.33647364377975464
Epoch 182 train loss: 0.3715, eval loss 0.33633434772491455
Epoch 183 train loss: 0.3572, eval loss 0.3361971080303192
Epoch 184 train loss: 0.3926, eval loss 0.33606570959091187
Epoch 185 train loss: 0.3705, eval loss 0.3359355628490448
Epoch 186 train loss: 0.3949, eval loss 0.3358091711997986
Epoch 187 train loss: 0.3795, eval loss 0.33568814396858215
Epoch 188 train loss: 0.3720, eval loss 0.33556926250457764
Epoch 189 train loss: 0.4005, eval loss 0.3354395031929016
Epoch 190 train loss: 0.3695, eval loss 0.3353138566017151
Epoch 191 train loss: 0.3782, eval loss 0.3351828157901764
Epoch 192 train loss: 0.3622, eval loss 0.3350561857223511
Epoch 193 train loss: 0.3873, eval loss 0.33493584394454956
Epoch 194 train loss: 0.3827, eval loss 0.3348226249217987
Epoch 195 train loss: 0.3780, eval loss 0.3347027003765106
Epoch 196 train loss: 0.3686, eval loss 0.3345889151096344
Epoch 197 train loss: 0.3851, eval loss 0.3344807028770447
Epoch 198 train loss: 0.3697, eval loss 0.33435896039009094
Epoch 199 train loss: 0.3648, eval loss 0.3342325985431671
Epoch 200 train loss: 0.3830, eval loss 0.33411774039268494
Epoch 201 train loss: 0.3662, eval loss 0.3339977264404297
Epoch 202 train loss: 0.4033, eval loss 0.333873987197876
Epoch 203 train loss: 0.3920, eval loss 0.33377546072006226
Epoch 204 train loss: 0.3749, eval loss 0.3336738348007202
Epoch 205 train loss: 0.4007, eval loss 0.3335514962673187
Epoch 206 train loss: 0.3941, eval loss 0.33344608545303345
Epoch 207 train loss: 0.3733, eval loss 0.3333373963832855
Epoch 208 train loss: 0.3804, eval loss 0.33323463797569275
Epoch 209 train loss: 0.3936, eval loss 0.3331311345100403
Epoch 210 train loss: 0.3751, eval loss 0.3330186605453491
Epoch 211 train loss: 0.3673, eval loss 0.3329099118709564
Epoch 212 train loss: 0.3988, eval loss 0.33280712366104126
Epoch 213 train loss: 0.3670, eval loss 0.3326951563358307
Epoch 214 train loss: 0.3838, eval loss 0.33259329199790955
Epoch 215 train loss: 0.3572, eval loss 0.3324728012084961
Epoch 216 train loss: 0.3982, eval loss 0.33237773180007935
Epoch 217 train loss: 0.3769, eval loss 0.33226123452186584
Epoch 218 train loss: 0.3719, eval loss 0.33215585350990295
Epoch 219 train loss: 0.3988, eval loss 0.3320513367652893
Epoch 220 train loss: 0.3834, eval loss 0.3319454491138458
Epoch 221 train loss: 0.3670, eval loss 0.3318536579608917
Epoch 222 train loss: 0.3939, eval loss 0.33175480365753174
Epoch 223 train loss: 0.3825, eval loss 0.33165934681892395
Epoch 224 train loss: 0.3399, eval loss 0.33155688643455505
Epoch 225 train loss: 0.3625, eval loss 0.33145400881767273
Epoch 226 train loss: 0.3510, eval loss 0.3313724100589752
Epoch 227 train loss: 0.3753, eval loss 0.33127376437187195
```

```
Epoch 228 train loss: 0.3748, eval loss 0.33117321133613586
Epoch 229 train loss: 0.3676, eval loss 0.33107709884643555
Epoch 230 train loss: 0.3973, eval loss 0.33098524808883667
Epoch 231 train loss: 0.3693, eval loss 0.33089500665664673
Epoch 232 train loss: 0.3885, eval loss 0.33080750703811646
Epoch 233 train loss: 0.3668, eval loss 0.330711305141449
Epoch 234 train loss: 0.3361, eval loss 0.3306247293949127
Epoch 235 train loss: 0.3673, eval loss 0.3305336833000183
Epoch 236 train loss: 0.3582, eval loss 0.33044007420539856
Epoch 237 train loss: 0.3828, eval loss 0.33034905791282654
Epoch 238 train loss: 0.3750, eval loss 0.33026012778282166
Epoch 239 train loss: 0.3815, eval loss 0.33015915751457214
Epoch 240 train loss: 0.3720, eval loss 0.33005884289741516
Epoch 241 train loss: 0.3649, eval loss 0.32998350262641907
Epoch 242 train loss: 0.3862, eval loss 0.32988718152046204
Epoch 243 train loss: 0.3742, eval loss 0.32979264855384827
Epoch 244 train loss: 0.3832, eval loss 0.32971954345703125
Epoch 245 train loss: 0.3851, eval loss 0.32963159680366516
Epoch 246 train loss: 0.3799, eval loss 0.3295457363128662
Epoch 247 train loss: 0.4294, eval loss 0.3294641077518463
Epoch 248 train loss: 0.3727, eval loss 0.3293939530849457
Epoch 249 train loss: 0.3563, eval loss 0.32931599020957947
Epoch 250 train loss: 0.3994, eval loss 0.3292248845100403
Epoch 251 train loss: 0.3494, eval loss 0.32913482189178467
Epoch 252 train loss: 0.3590, eval loss 0.3290702700614929
Epoch 253 train loss: 0.3629, eval loss 0.3289816379547119
Epoch 254 train loss: 0.3985, eval loss 0.3288980722427368
Epoch 255 train loss: 0.3673, eval loss 0.32881614565849304
Epoch 256 train loss: 0.3666, eval loss 0.32874396443367004
Epoch 257 train loss: 0.3557, eval loss 0.32866308093070984
Epoch 258 train loss: 0.3653, eval loss 0.32858890295028687
Epoch 259 train loss: 0.3815, eval loss 0.3285055160522461
Epoch 260 train loss: 0.3998, eval loss 0.3284161686897278
Epoch 261 train loss: 0.3688, eval loss 0.3283412456512451
Epoch 262 train loss: 0.3692, eval loss 0.3282610774040222
Epoch 263 train loss: 0.3718, eval loss 0.3281959593296051
Epoch 264 train loss: 0.3892, eval loss 0.3281315267086029
Epoch 265 train loss: 0.3654, eval loss 0.328055739402771
Epoch 266 train loss: 0.3599, eval loss 0.3279789388179779
Epoch 267 train loss: 0.3641, eval loss 0.3279001712799072
Epoch 268 train loss: 0.3648, eval loss 0.3278297483921051
Epoch 269 train loss: 0.3941, eval loss 0.32775574922561646
Epoch 270 train loss: 0.3557, eval loss 0.3276854157447815
Epoch 271 train loss: 0.3996, eval loss 0.3276311159133911
Epoch 272 train loss: 0.3521, eval loss 0.32755783200263977
Epoch 273 train loss: 0.3921, eval loss 0.32748404145240784
Epoch 274 train loss: 0.3626, eval loss 0.32741793990135193
Epoch 275 train loss: 0.3834, eval loss 0.32735076546669006
Epoch 276 train loss: 0.3524, eval loss 0.3272766172885895
```

```
Epoch 277 train loss: 0.3397, eval loss 0.3272009789943695
Epoch 278 train loss: 0.3507, eval loss 0.3271314203739166
Epoch 279 train loss: 0.3296, eval loss 0.3270718455314636
Epoch 280 train loss: 0.3520, eval loss 0.3269929885864258
Epoch 281 train loss: 0.3806, eval loss 0.3269307017326355
Epoch 282 train loss: 0.3685, eval loss 0.3268565237522125
Epoch 283 train loss: 0.3597, eval loss 0.32678738236427307
Epoch 284 train loss: 0.3767, eval loss 0.32673901319503784
Epoch 285 train loss: 0.3389, eval loss 0.3266809284687042
Epoch 286 train loss: 0.3943, eval loss 0.32660698890686035
Epoch 287 train loss: 0.3304, eval loss 0.32655566930770874
Epoch 288 train loss: 0.3785, eval loss 0.3264819085597992
Epoch 289 train loss: 0.4076, eval loss 0.3264172673225403
Epoch 290 train loss: 0.3755, eval loss 0.3263411223888397
Epoch 291 train loss: 0.3860, eval loss 0.32627540826797485
Epoch 292 train loss: 0.3720, eval loss 0.3262070119380951
Epoch 293 train loss: 0.3641, eval loss 0.3261454403400421
Epoch 294 train loss: 0.3835, eval loss 0.3260810375213623
Epoch 295 train loss: 0.3580, eval loss 0.3260314464569092
Epoch 296 train loss: 0.3504, eval loss 0.32598501443862915
Epoch 297 train loss: 0.3572, eval loss 0.32592469453811646
Epoch 298 train loss: 0.3560, eval loss 0.3258619010448456
Epoch 299 train loss: 0.3574, eval loss 0.32581713795661926
```

```
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: 90.23%
F1: 68.86%
Precision: 63.33%
Recall: 75.45%
```

Wyniki wyglądają już dużo lepiej.

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

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

Drugim ulepszeniem jest dodanie **wag klas (class weights)**. Mamy do czynienia z problemem klasyfikacji niezbalansowanej, więc klasa mniejszościowa, ważniejsza dla nas, powinna dostać

większą wagę. Implementuje się to trywialnie prosto - po prostu mnożymy wartość funkcji kosztu dla danego przykładu przez wagę dla prawdziwej klasy tego przykładu. Praktycznie każdy klasyfikator operujący na jakiejś ważonej funkcji może działać w ten sposób, nie tylko sieci neuronowe.

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

Zadanie 8 (1 punkt)

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

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

Zamień używany optymalizator na `AdamW`.

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

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

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

        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_p)

        self.batchMLP = nn.BatchNorm1d(256)
        self.batchHidden = nn.BatchNorm1d(128)

    def forward(self, x):
      x = self.relu(self.mlp(x))
      x = self.batchMLP(x)
      x = self.dropout(x)
      x = self.relu(self.hidden(x))
      x = self.batchHidden(x)
      x = self.dropout(x)
      x = self.out(x)

      return 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)

print (type(y))
```

```
<class 'numpy.ndarray'>
```

```
from sklearn.utils.class_weight import compute_class_weight

weights = compute_class_weight(
    'balanced', classes = [0, 1], y=y
)

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

early_stopping_patience = 4

model = NormalizingMLP(
    input_size=X_train.shape[1],
    dropout_p=dropout_p
)
optimizer = optim.AdamW(
    model.parameters(),
    lr = learning_rate,
    weight_decay = l2_reg
)
weight_tensor = torch.tensor(weights[1], dtype=torch.float32)
# print(type(weights_tensor))
# print(weights_tensor)
loss_fn = torch.nn.BCEWithLogitsLoss(weight = weight_tensor)

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!

        # Forward pass
        outputs = model(X_batch)
        # Loss calculation
        loss = loss_fn(outputs, y_batch)
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        # Optimalization
        optimizer.step()

    # model evaluation, early stopping
    # implement me!

    model.eval()
    valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
    # best models and results
    if valid_metrics.get("loss") < best_val_loss:
      best_val_loss = valid_metrics.get("loss")
      best_model = model
      best_threshold = valid_metrics.get("treshold")

      steps_without_improvement =0
    else:
      steps_without_improvement += 1

      if steps_without_improvement == early_stopping_patience:
        break

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

```
Epoch 0 train loss: 0.9707, eval loss 0.7177432775497437
Epoch 1 train loss: 0.7937, eval loss 0.6724789142608643
Epoch 2 train loss: 0.8336, eval loss 0.6620045304298401
Epoch 3 train loss: 0.7144, eval loss 0.6577975153923035
Epoch 4 train loss: 0.7291, eval loss 0.6543317437171936
Epoch 5 train loss: 0.6326, eval loss 0.6549965143203735
Epoch 6 train loss: 0.7472, eval loss 0.6568703651428223
Epoch 7 train loss: 0.7182, eval loss 0.6564995050430298
```

```
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: 90.80%
F1: 70.17%
Precision: 63.61%
Recall: 78.25%
```

Pytania kontrolne (1 punkt)

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

AD1)

1. liczba warstw i neuronów. Tutaj używaliśmy tylko jednej konfiguracji :(3 warstwy 1-sza ma max, 2-ga 156, 3-cia 128 neuronów). Widać poprawę względem zwykłej regresji liniowej która jest na 1 warstwie
2. learning rate
3. l2 regression
4. dropout

AD2) Użycie L1 może sprawić, że niektóre wagi będą miały wartości dokładnie równe zeru. Wtedy gradient też będzie 0 i wagi nie będą aktualizowane (raz 0 zostanie na zawsze). Można użyć dropoutu zamiast L1 bo on działa jako rodzaj regularyzacj. Losowo "wyłącza" pewne neurony podczas treningu co zmniejsza ryzyko przeuczenia. W ewaluacji nie jest brany pod uwagę żaden dropout i wszystkie neurony działają.

AD3) Tak, bo jedna metryka np. funkcja kosztu może się zatrzymać w innym miejscu niż dokładność czy f1. jak koszt zacznię znowu rosnąć i model zostanie wcześniej zatrzymany to neurony będą inaczej wytrenowane niż takiego który działał dalej bo f1 dalej się zwiększało. ten drugi będzie najczęściej dawał lepszy wynik f1 bo jest pod to przetrenowany

Akceleracja sprzętowa (dla zainteresowanych)

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

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

```
import time

model = NormalizingMLP(
    input_size=X_train.shape[1],
```

```
    dropout_p=dropout_p
).to('cuda')

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

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

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

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

        optimizer.step()
        optimizer.zero_grad()

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

        step_counter += 1

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

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

```
---------------------------------------------------------------------------
-----
RuntimeError                              Traceback (most recent call
last)
<ipython-input-144-e87eae93c7f8> in <cell line: 3>()
      4     input_size=X_train.shape[1],
      5     dropout_p=dropout_p
----> 6 ).to('cuda')
      7
      8 optimizer = torch.optim.AdamW(model.parameters(),
lr=learning_rate, weight_decay=1e-4)

/usr/local/lib/python3.10/dist-packages/torch/nn/modules/module.py in
```

```
to(self, *args, **kwargs)
   1158             return t.to(device, dtype if t.is_floating_point()
or t.is_complex() else None, non_blocking)
   1159
-> 1160         return self._apply(convert)
   1161
   1162     def register_full_backward_pre_hook(

/usr/local/lib/python3.10/dist-packages/torch/nn/modules/module.py in
_apply(self, fn, recurse)
    808         if recurse:
    809             for module in self.children():
--> 810                 module._apply(fn)
    811
    812         def compute_should_use_set_data(tensor,
tensor_applied):

/usr/local/lib/python3.10/dist-packages/torch/nn/modules/module.py in
_apply(self, fn, recurse)
    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_applied)
    835             if should_use_set_data:

/usr/local/lib/python3.10/dist-packages/torch/nn/modules/module.py in
convert(t)
   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)
   1159
   1160         return self._apply(convert)

/usr/local/lib/python3.10/dist-packages/torch/cuda/__init__.py in
_lazy_init()
    296         if "CUDA_MODULE_LOADING" not in os.environ:
    297             os.environ["CUDA_MODULE_LOADING"] = "LAZY"
--> 298         torch._C._cuda_init()
    299         # Some of the queued calls may reentrantly call
_lazy_init();
    300         # we need to just return without initializing in that
case.

RuntimeError: Found no NVIDIA driver on your system. Please check that
you have an NVIDIA GPU and installed a driver from
http://www.nvidia.com/Download/index.aspx
```

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