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 pytorc
```

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

# All requested packages already installed.
```

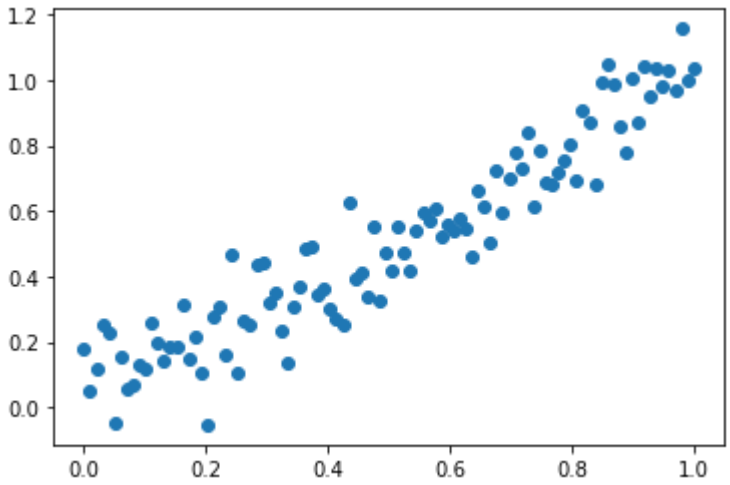
Wprowadzenie

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 0x28641ccd0>
```



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)**:

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

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

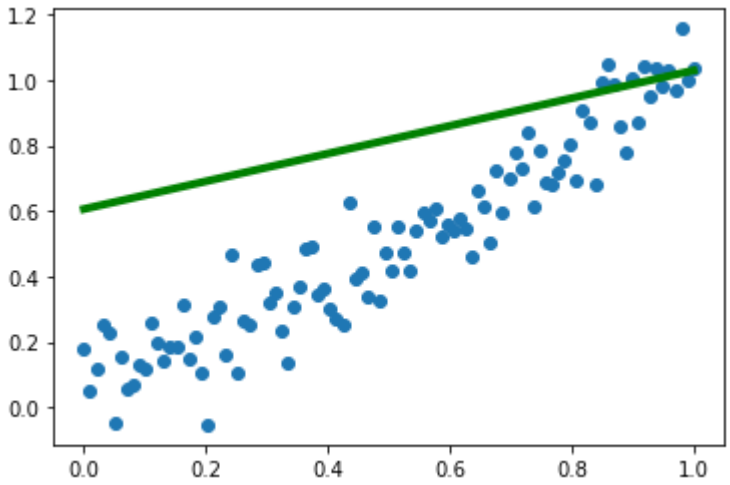
Zadanie 1 (0.5 punkt)

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(np.square(y - y_hat)) / len(y)
```

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)
```

Out[ ]:
```
MSE: 0.133
[<matplotlib.lines.Line2D at 0x286465f10>]
```



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)**, dająca 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:

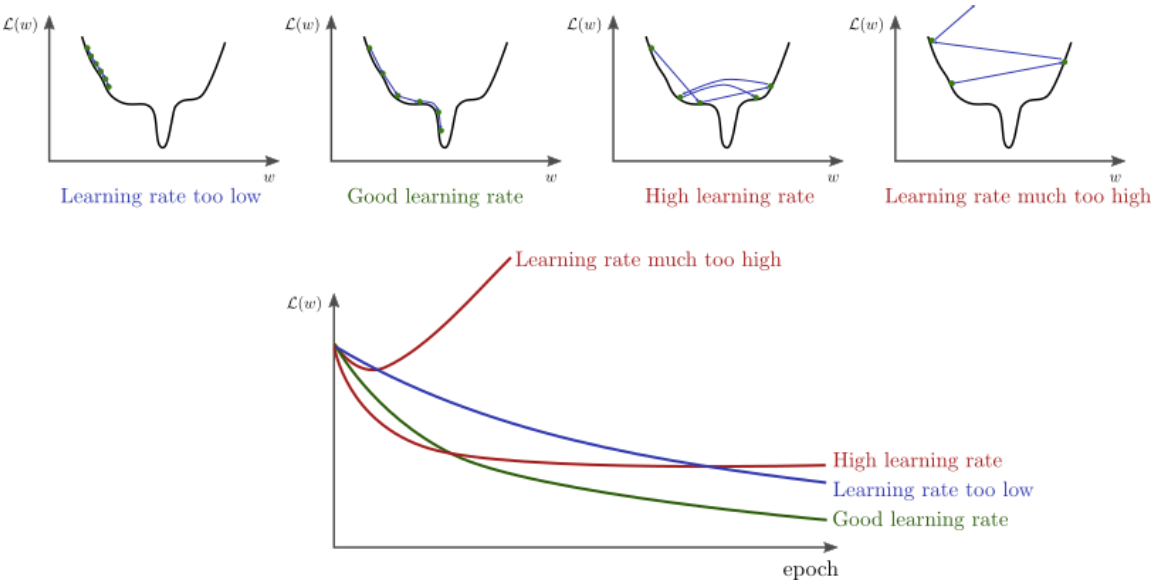
$$x_{t+1} = x_t - \alpha \cdot \frac{f(x)}{dx}$$

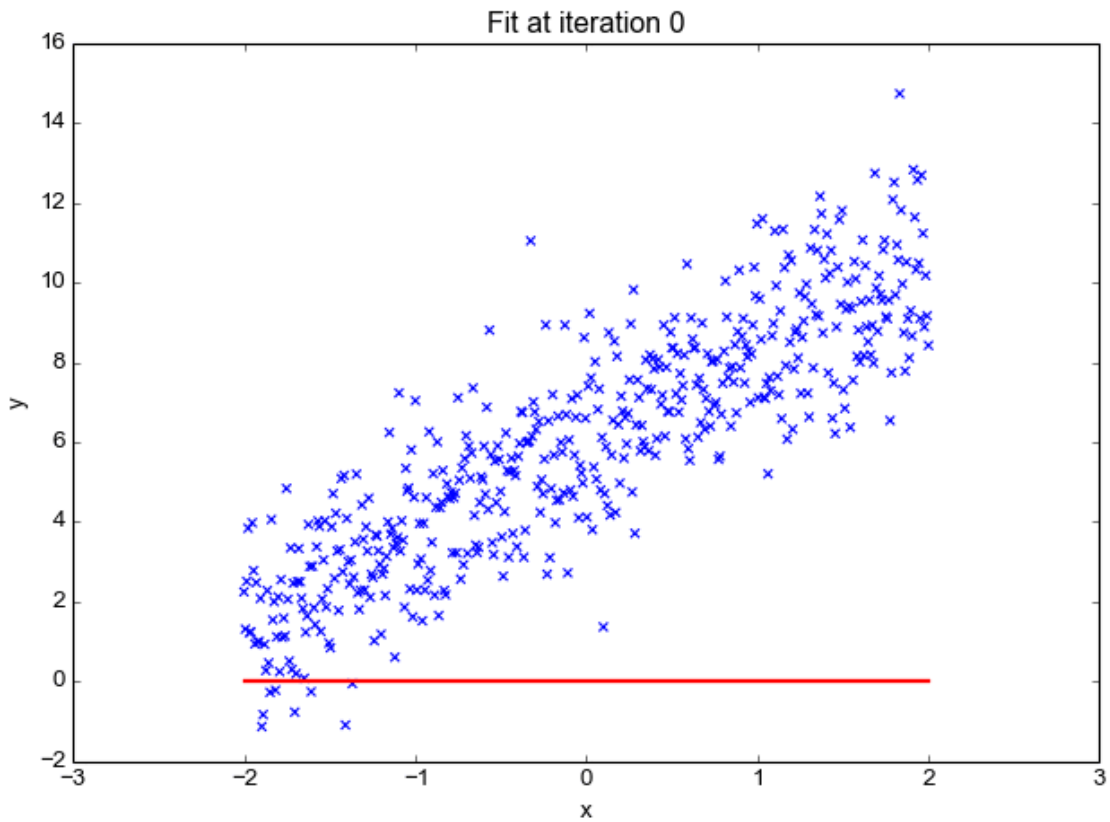
α to niewielka wartość (rzędu zwykle 10^{-5} - 10^{-2}), wprowadzona, aby trzymać się założenia o małej zmianie parametrów (ϵ). Nazywa się ją **stałą uczącą (learning rate)** i

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:

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

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

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: float =
):
    y_hat = a * x + b
    errors = y - y_hat
    n = len(y)

    new_a = a + learning_rate * -2 / n * np.sum(errors * -x)
    new_b = b + learning_rate * -2 / n * np.sum(errors * -1)

    return new_a, new_b
```

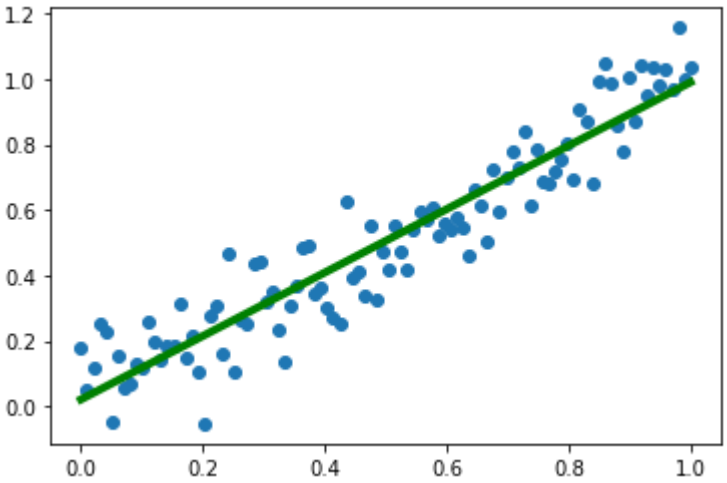
In [ ]:
```
for i in range(1000):
    loss = mse(y, a * x + b)
    a, b = optimize(x, y, a, b)
```

```
    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 0x2864d07c0>]
```



Udało Ci się wytrenować Twoją 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 stwierdzeniem, 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.rand(10)

# elementwise sum
print(ones + noise)

# elementwise multiplication
print(ones * noise)

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

```
tensor([1.8974, 1.0824, 1.8647, 1.7251, 1.1006, 1.1998, 1.5010, 1.6244, 1.43
53,
        1.9289])
tensor([0.8974, 0.0824, 0.8647, 0.7251, 0.1006, 0.1998, 0.5010, 0.6244, 0.43
53,
        0.9289])
tensor(5.3598)
```

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([0.3096], requires_grad=True), tensor([0.7051], requires_grad=True))
```

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

- 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(0.1724, 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([0.2415])
```

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([0.4830])
```

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(0.1724, dtype=torch.float64, grad_fn=<MseLossBackward0
>)
step 100 loss:  tensor(0.0136, dtype=torch.float64, grad_fn=<MseLossBackward
0>)
step 200 loss:  tensor(0.0103, dtype=torch.float64, grad_fn=<MseLossBackward
0>)
step 300 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward
0>)
step 400 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward
0>)
step 500 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward
0>)
step 600 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward
0>)
step 700 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward
0>)
step 800 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward
0>)
step 900 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward
0>)
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. Back propagation - obliczenie pochodnych oraz zerowanie gradientów
4. Optimalization - aktualizacja wag
5. Other - ewaluacja na zbiorze walidacyjnym, logging etc.

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

# training loop in each epoch
for 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: 0.0932
step 100 loss: 0.0114
step 200 loss: 0.0102
step 300 loss: 0.0101
step 400 loss: 0.0101
step 500 loss: 0.0101
step 600 loss: 0.0101
step 700 loss: 0.0101
step 800 loss: 0.0101
step 900 loss: 0.0101
final loss: tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
```

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

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

Uwaga - PyTorch wymaga tensora klas będącego liczbami zmiennoprzecinkowymi!

Zbiór danych

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

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

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

```
--2022-12-25 17:55:23--  https://archive.ics.uci.edu/ml/machine-learning-dat
abases/adult/adult.data
Resolving archive.ics.uci.edu (archive.ics.uci.edu)... 128.195.10.252
Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|128.195.10.252|:44
3... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3974305 (3.8M) [application/x-httpd-php]
Saving to: 'adult.data.4'

adult.data.4        100%[===================>]   3.79M   403KB/s    in 11s

2022-12-25 17:55:35 (352 KB/s) - 'adult.data.4' saved [3974305/3974305]
```

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

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

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

In [ ]:
```python
# attribution: https://www.kaggle.com/code/royshih23/topic7-classification-i
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
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, StandardScale

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

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

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

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

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

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

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

```
categorical_X_test = categorical_encoder.transform(categorical_X_test)

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

X_train.shape, y_train.shape
```

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

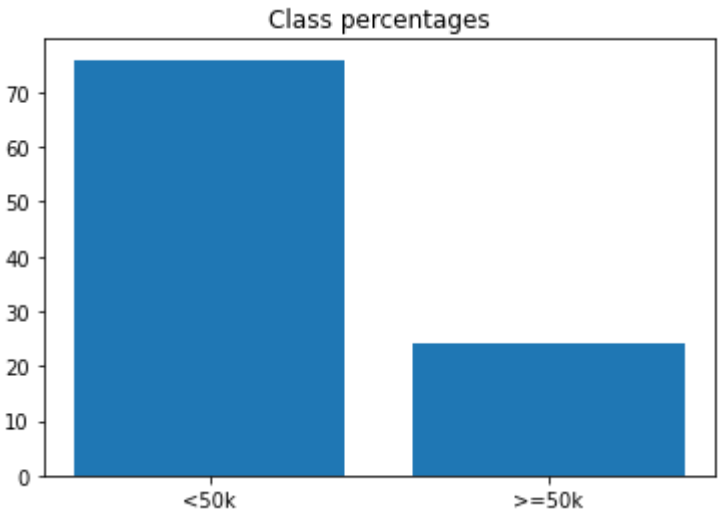
Uwaga co do typów - w PyTorchu wszystko w sieci neuronowej musi być typu `float32`. W szczególności trzeba uważać na konwersje z Numpy'a, który używa domyślnie typu `float64`. Może ci się przydać metoda `.float()`.

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

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

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

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

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

In [ ]:
```
import matplotlib.pyplot as plt

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

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



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

Zadanie 3 (1 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 [ ]:
```
a = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)

learning_rate = 1e-3

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

for i in range(3000):
    y_pred = activation(model(X_train))
    loss = loss_fn(y_pred, y_train)

    if i % 100 == 0:
        print(i, loss.item())

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

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

```
0 0.7027778625488281
100 0.669400155544281
200 0.6415488123893738
300 0.6181450486183167
400 0.5983203649520874
500 0.5813822746276855
600 0.5667816400527954
700 0.5540837049484253
800 0.542944073677063
900 0.5330893993377686
1000 0.5243021249771118
1100 0.5164081454277039
1200 0.5092675685882568
1300 0.5027670860290527
1400 0.49681466817855835
1500 0.49133530259132385
1600 0.48626700043678284
1700 0.48155835270881653
1800 0.47716671228408813
1900 0.47305628657341003
2000 0.4691968560218811
2100 0.4655625522136688
2200 0.4621315598487854
2300 0.45888492465019226
2400 0.45580625534057617
2500 0.4528813660144806
2600 0.45009779930114746
2700 0.44744449853897095
2800 0.44491180777549744
2900 0.4424910247325897
final loss: tensor(0.4402, grad_fn=<BinaryCrossEntropyBackward0>)
```

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

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

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

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: 84.88%
```

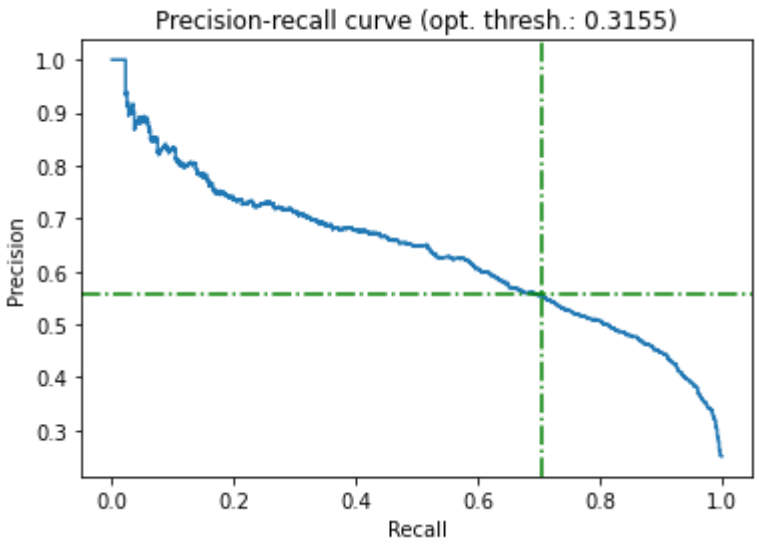
Jest to całkiem dobry wynik, a może być jeszcze lepszy. Sprawdźmy dla pewności jeszcze inne metryki: precyzję, recall oraz F1-score. Dodatkowo narysujemy krzywą precision-recall, czyli jak zmieniają się te metryki w zależności od przyjętego progu (threshold) prawdopodobieństwa, powyżej którego przyjmujemy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

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

In [ ]:
```
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.

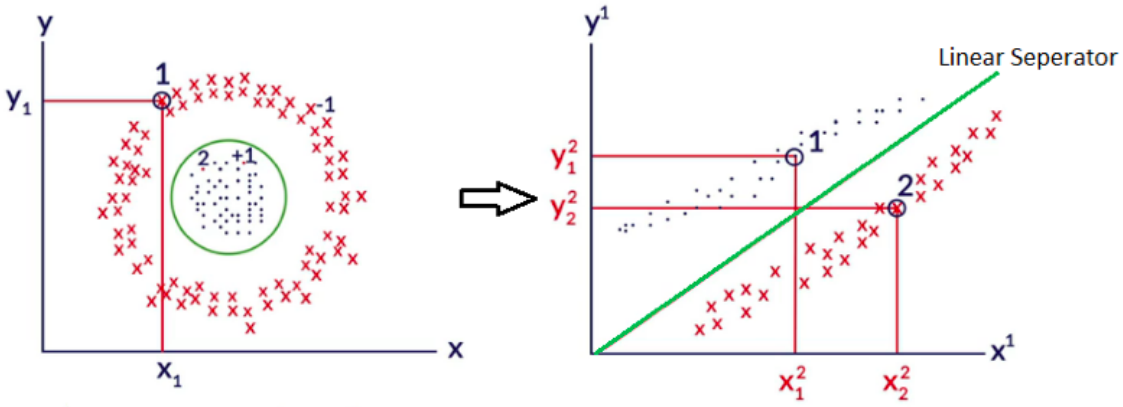
Dodatkowo, pomimo prostoty MLP są bardzo potężne - udowodniono, że perceptrony (ich powszechna nazwa) są uniwersalnym aproksymatorem, będącym w stanie przybliżyć dowolną funkcję z odpowiednio małym błędem, zakładając wystarczającą wielkość warstw sieci. Szczególne ich wersje potrafią nawet reprezentować 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 na przykład (wide?) learning. (aka. czemu staramy się budować głębokie sieci, a nie płytkie za to szerokie)

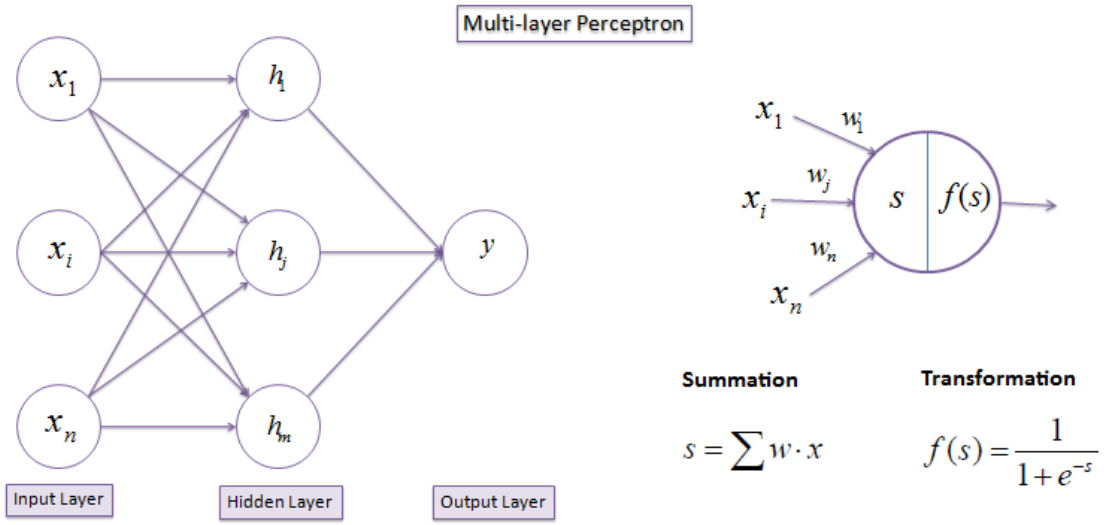
Sieci MLP

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.

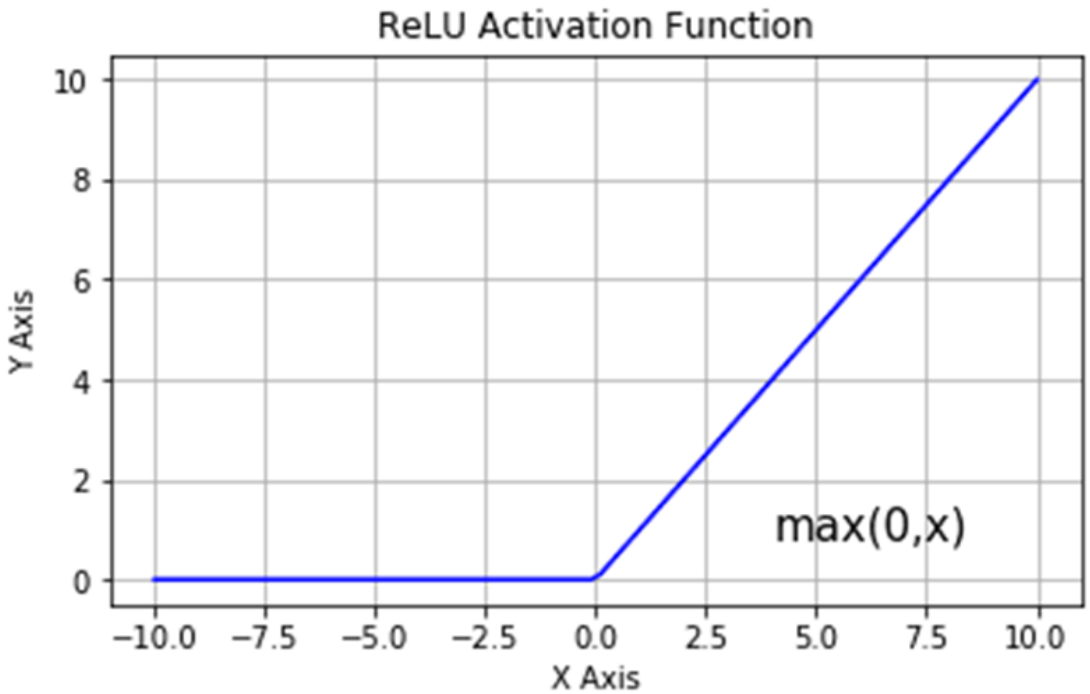


Zapisane matematycznie MLP to:
$h_1 = f_1(x)$
$h_2 = f_2(h_1)$
$h_3 = f_3(h_2)$
$\dots h_n = f_n(h_{n-1})$
gdzie x to wejście f_i to funkcja aktywacji i-tej warstwy, a h_i to wyjście i-tej warstwy, nazywane **ukrytą reprezentacją (hidden representation)**, lub *latent representation*. Nazwa bierze się 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:
$h_1 = f_1(x) = xh_2 = f_2(f_1) = f_2(x) = x \ldots h_n = f_n(f_{n-1}) = f_n(x) = x$ 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 wywoływanych 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 [ ]:
```
from torch import sigmoid

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

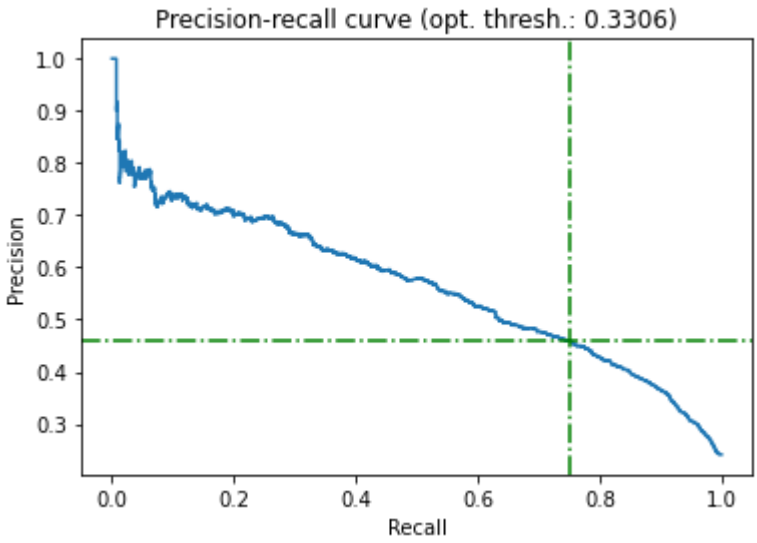
```
        super().__init__()

        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 [ ]:
```
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: 0.7087
Epoch 200 train loss: 0.6824
Epoch 400 train loss: 0.6597
Epoch 600 train loss: 0.6397
Epoch 800 train loss: 0.6220
Epoch 1000 train loss: 0.6062
Epoch 1200 train loss: 0.5921
Epoch 1400 train loss: 0.5797
Epoch 1600 train loss: 0.5686
Epoch 1800 train loss: 0.5589
final loss: 0.5503
```

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: 78.56%
```



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 ona 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 [ ]:
```python
from typing import Optional

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

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

    loss = loss_fn(y_pred_score, y)
    auroc = roc_auc_score(y, y_pred_score)

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

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

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

```
    }

    return results
```

Zadanie 6 (1 punkt)

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

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

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

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

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

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

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

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

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

Na razie użyjemy jednak minibatch SGD.

Poniżej znajduje się implementacja prostej klasy dziedziczącej po `Dataset` - tak w PyTorchu implementuje się własne zbiory danych. Użycie takich klas umożliwia użycie

klas ładujących dane (`DataLoader`), które z kolei pozwalają łatwo ładować batche danych. Trzeba w takiej klasie zaimplementować metody:

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

In [ ]:
```
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):
    model.train()

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

        optimizer.step()
        optimizer.zero_grad()

    # model evaluation, early stopping
    model.eval()
    valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
    if valid_metrics['loss'] < best_val_loss:
        best_val_loss = valid_metrics['loss']
        best_threshold = valid_metrics['optimal_threshold']
        best_model = deepcopy(model)
        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 {vali
```

```
Epoch 0 train loss: 0.6820, eval loss 0.8482637405395508
Epoch 1 train loss: 0.6599, eval loss 0.8407472372055054
Epoch 2 train loss: 0.6363, eval loss 0.8336960673332214
Epoch 3 train loss: 0.6264, eval loss 0.8270767331123352
Epoch 4 train loss: 0.6095, eval loss 0.8208481073379517
Epoch 5 train loss: 0.5907, eval loss 0.8149405717849731
Epoch 6 train loss: 0.5956, eval loss 0.809410035610199
Epoch 7 train loss: 0.5693, eval loss 0.80413818359375
Epoch 8 train loss: 0.5705, eval loss 0.7992390990257263
Epoch 9 train loss: 0.5610, eval loss 0.7946004271507263
Epoch 10 train loss: 0.5592, eval loss 0.7903221249580383
Epoch 11 train loss: 0.5500, eval loss 0.7863208055496216
Epoch 12 train loss: 0.5409, eval loss 0.7825679779052734
Epoch 13 train loss: 0.5251, eval loss 0.779140293598175
Epoch 14 train loss: 0.5212, eval loss 0.775876522064209
Epoch 15 train loss: 0.5301, eval loss 0.7728952169418335
Epoch 16 train loss: 0.5270, eval loss 0.7700825929641724
Epoch 17 train loss: 0.5222, eval loss 0.7674464583396912
Epoch 18 train loss: 0.5195, eval loss 0.7650181651115417
Epoch 19 train loss: 0.5138, eval loss 0.7627365589141846
Epoch 20 train loss: 0.5184, eval loss 0.7606518268585205
Epoch 21 train loss: 0.5016, eval loss 0.7586310505867004
Epoch 22 train loss: 0.5078, eval loss 0.7567402124404907
Epoch 23 train loss: 0.4772, eval loss 0.7549142837524414
Epoch 24 train loss: 0.4993, eval loss 0.7531425356864929
Epoch 25 train loss: 0.4836, eval loss 0.7515053153038025
Epoch 26 train loss: 0.4799, eval loss 0.7499474883079529
Epoch 27 train loss: 0.4789, eval loss 0.7484464645385742
Epoch 28 train loss: 0.4701, eval loss 0.746923565864563
Epoch 29 train loss: 0.4735, eval loss 0.7454953193664551
Epoch 30 train loss: 0.4591, eval loss 0.7441527843475342
Epoch 31 train loss: 0.4828, eval loss 0.7428088188171387
Epoch 32 train loss: 0.4698, eval loss 0.7415484189987183
Epoch 33 train loss: 0.4743, eval loss 0.7402647733688354
Epoch 34 train loss: 0.4686, eval loss 0.7390721440315247
Epoch 35 train loss: 0.4599, eval loss 0.7378392219543457
Epoch 36 train loss: 0.4778, eval loss 0.7366955280303955
Epoch 37 train loss: 0.4594, eval loss 0.7355461716651917
Epoch 38 train loss: 0.4574, eval loss 0.7344045639038086
Epoch 39 train loss: 0.4600, eval loss 0.7333373427391052
Epoch 40 train loss: 0.4554, eval loss 0.7322627902030945
Epoch 41 train loss: 0.4301, eval loss 0.7312365770339966
Epoch 42 train loss: 0.4457, eval loss 0.7302135825157166
Epoch 43 train loss: 0.4401, eval loss 0.7292301654815674
Epoch 44 train loss: 0.4201, eval loss 0.728317379951477
Epoch 45 train loss: 0.4419, eval loss 0.7274327278137207
Epoch 46 train loss: 0.4333, eval loss 0.7265297174453735
Epoch 47 train loss: 0.4329, eval loss 0.7256688475608826
Epoch 48 train loss: 0.4326, eval loss 0.7248437404632568
Epoch 49 train loss: 0.4183, eval loss 0.7240288853645325
Epoch 50 train loss: 0.4293, eval loss 0.7232667207717896
Epoch 51 train loss: 0.4207, eval loss 0.7224746346473694
Epoch 52 train loss: 0.4116, eval loss 0.7216762900352478
Epoch 53 train loss: 0.4068, eval loss 0.7209464907646179
Epoch 54 train loss: 0.4128, eval loss 0.7202683687210083
Epoch 55 train loss: 0.4215, eval loss 0.7195867896080017
Epoch 56 train loss: 0.4291, eval loss 0.7189809679985046
Epoch 57 train loss: 0.4179, eval loss 0.718367874622345
Epoch 58 train loss: 0.4302, eval loss 0.7177196741104126
Epoch 59 train loss: 0.4057, eval loss 0.7171512842178345
Epoch 60 train loss: 0.4333, eval loss 0.7165314555168152
Epoch 61 train loss: 0.4190, eval loss 0.715965211391449
Epoch 62 train loss: 0.4173, eval loss 0.7153418064117432
Epoch 63 train loss: 0.4399, eval loss 0.7148175835609436
```

```
Epoch 64 train loss: 0.4186, eval loss 0.7143299579620361
Epoch 65 train loss: 0.3668, eval loss 0.7138274312019348
Epoch 66 train loss: 0.3995, eval loss 0.7132821679115295
Epoch 67 train loss: 0.3946, eval loss 0.7128444910049438
Epoch 68 train loss: 0.4080, eval loss 0.712427020072937
Epoch 69 train loss: 0.4008, eval loss 0.71196049451828
Epoch 70 train loss: 0.4060, eval loss 0.7115620970726013
Epoch 71 train loss: 0.3865, eval loss 0.7111899256706238
Epoch 72 train loss: 0.3818, eval loss 0.710695207118988
Epoch 73 train loss: 0.3963, eval loss 0.7102987766265869
Epoch 74 train loss: 0.3989, eval loss 0.7099325656890869
Epoch 75 train loss: 0.4006, eval loss 0.7095325589179993
Epoch 76 train loss: 0.3801, eval loss 0.7091837525367737
Epoch 77 train loss: 0.3817, eval loss 0.7088682651519775
Epoch 78 train loss: 0.4002, eval loss 0.7085543274879456
Epoch 79 train loss: 0.3997, eval loss 0.7082706093788147
Epoch 80 train loss: 0.3751, eval loss 0.7079785466194153
Epoch 81 train loss: 0.3995, eval loss 0.7076250314712524
Epoch 82 train loss: 0.4041, eval loss 0.7073274850845337
Epoch 83 train loss: 0.3934, eval loss 0.7070432901382446
Epoch 84 train loss: 0.3541, eval loss 0.7068264484405518
Epoch 85 train loss: 0.3969, eval loss 0.7065616250038147
Epoch 86 train loss: 0.4171, eval loss 0.706336498260498
Epoch 87 train loss: 0.3733, eval loss 0.7061344981193542
Epoch 88 train loss: 0.3866, eval loss 0.7058531045913696
Epoch 89 train loss: 0.4083, eval loss 0.7056257128715515
Epoch 90 train loss: 0.3883, eval loss 0.7053776383399963
Epoch 91 train loss: 0.3796, eval loss 0.705182671546936
Epoch 92 train loss: 0.3873, eval loss 0.7050406336784363
Epoch 93 train loss: 0.3696, eval loss 0.7048986554145813
Epoch 94 train loss: 0.4185, eval loss 0.7046884894371033
Epoch 95 train loss: 0.3915, eval loss 0.7044448256492615
Epoch 96 train loss: 0.3794, eval loss 0.7042502164840698
Epoch 97 train loss: 0.4001, eval loss 0.7040590047836304
Epoch 98 train loss: 0.4062, eval loss 0.7038429379463196
Epoch 99 train loss: 0.3859, eval loss 0.7036913633346558
Epoch 100 train loss: 0.3812, eval loss 0.7035761475563049
Epoch 101 train loss: 0.3835, eval loss 0.7034266591072083
Epoch 102 train loss: 0.3672, eval loss 0.7032167911529541
Epoch 103 train loss: 0.3620, eval loss 0.703087568283081
Epoch 104 train loss: 0.3666, eval loss 0.7029927372932434
Epoch 105 train loss: 0.3867, eval loss 0.7028989195823669
Epoch 106 train loss: 0.3627, eval loss 0.7027407884597778
Epoch 107 train loss: 0.3879, eval loss 0.7026439309120178
Epoch 108 train loss: 0.3908, eval loss 0.7024739384651184
Epoch 109 train loss: 0.3733, eval loss 0.7022480368614197
Epoch 110 train loss: 0.3895, eval loss 0.7021757364273071
Epoch 111 train loss: 0.3944, eval loss 0.7019827961921692
Epoch 112 train loss: 0.3835, eval loss 0.7018439173698425
Epoch 113 train loss: 0.3866, eval loss 0.7017215490341187
Epoch 114 train loss: 0.3702, eval loss 0.7015711069107056
Epoch 115 train loss: 0.3938, eval loss 0.7013972401618958
Epoch 116 train loss: 0.3914, eval loss 0.7012431025505066
Epoch 117 train loss: 0.3768, eval loss 0.7011927366256714
Epoch 118 train loss: 0.3562, eval loss 0.7010598182678223
Epoch 119 train loss: 0.3956, eval loss 0.7009446024894714
Epoch 120 train loss: 0.3873, eval loss 0.700867235660553
Epoch 121 train loss: 0.4076, eval loss 0.7007826566696167
Epoch 122 train loss: 0.3795, eval loss 0.7007232308387756
Epoch 123 train loss: 0.3524, eval loss 0.700610876083374
Epoch 124 train loss: 0.3650, eval loss 0.7004923820495605
Epoch 125 train loss: 0.3776, eval loss 0.7003883123397827
Epoch 126 train loss: 0.3947, eval loss 0.7002488970756531
Epoch 127 train loss: 0.4013, eval loss 0.7001670598983765
```

```
Epoch 128 train loss: 0.3988, eval loss 0.7001720070838928
Epoch 129 train loss: 0.3682, eval loss 0.7001944780349731
Epoch 130 train loss: 0.4080, eval loss 0.7001946568489075
Epoch 131 train loss: 0.3830, eval loss 0.700046718120575
Epoch 132 train loss: 0.3858, eval loss 0.6998353600502014
Epoch 133 train loss: 0.3483, eval loss 0.6997349858283997
Epoch 134 train loss: 0.3731, eval loss 0.6996384263038635
Epoch 135 train loss: 0.3803, eval loss 0.6995757222175598
Epoch 136 train loss: 0.3552, eval loss 0.69949871301651
Epoch 137 train loss: 0.3778, eval loss 0.6994035840034485
Epoch 138 train loss: 0.3911, eval loss 0.6994004249572754
Epoch 139 train loss: 0.3425, eval loss 0.6992517113685608
Epoch 140 train loss: 0.4003, eval loss 0.699190616607666
Epoch 141 train loss: 0.3465, eval loss 0.6991244554519653
Epoch 142 train loss: 0.3782, eval loss 0.6990728378295898
Epoch 143 train loss: 0.3668, eval loss 0.6990339756011963
Epoch 144 train loss: 0.3590, eval loss 0.6988798379898071
Epoch 145 train loss: 0.3969, eval loss 0.6988191604614258
Epoch 146 train loss: 0.3653, eval loss 0.6988219618797302
Epoch 147 train loss: 0.3745, eval loss 0.6987206339836121
Epoch 148 train loss: 0.3836, eval loss 0.6985975503921509
Epoch 149 train loss: 0.3815, eval loss 0.6985000967979431
Epoch 150 train loss: 0.3832, eval loss 0.6984884142875671
Epoch 151 train loss: 0.3730, eval loss 0.6984612345695496
Epoch 152 train loss: 0.3575, eval loss 0.6984227299690247
Epoch 153 train loss: 0.3789, eval loss 0.6983429193496704
Epoch 154 train loss: 0.3465, eval loss 0.6983123421669006
Epoch 155 train loss: 0.3724, eval loss 0.6982325911521912
Epoch 156 train loss: 0.3930, eval loss 0.6981523036956787
Epoch 157 train loss: 0.3731, eval loss 0.6980410814285278
Epoch 158 train loss: 0.3800, eval loss 0.6980115175247192
Epoch 159 train loss: 0.3657, eval loss 0.6979005932807922
Epoch 160 train loss: 0.3721, eval loss 0.6979076862335205
Epoch 161 train loss: 0.3763, eval loss 0.697760820388794
Epoch 162 train loss: 0.3888, eval loss 0.6976848840713501
Epoch 163 train loss: 0.3594, eval loss 0.697629988193512
Epoch 164 train loss: 0.3814, eval loss 0.6975898146629333
Epoch 165 train loss: 0.3529, eval loss 0.6975265741348267
Epoch 166 train loss: 0.3650, eval loss 0.6974433064460754
Epoch 167 train loss: 0.3934, eval loss 0.6973336935043335
Epoch 168 train loss: 0.3633, eval loss 0.6973417401313782
Epoch 169 train loss: 0.3634, eval loss 0.6973891854286194
Epoch 170 train loss: 0.3547, eval loss 0.697336733341217
Epoch 171 train loss: 0.3529, eval loss 0.6973036527633667
Epoch 172 train loss: 0.3747, eval loss 0.6972717046737671
Epoch 173 train loss: 0.3815, eval loss 0.6972364783287048
Epoch 174 train loss: 0.3602, eval loss 0.697152316570282
Epoch 175 train loss: 0.3759, eval loss 0.6971192955970764
Epoch 176 train loss: 0.3777, eval loss 0.6971026659011841
Epoch 177 train loss: 0.4009, eval loss 0.6969349980354309
Epoch 178 train loss: 0.3635, eval loss 0.6968976259231567
Epoch 179 train loss: 0.3499, eval loss 0.6968873143196106
Epoch 180 train loss: 0.3707, eval loss 0.6968182921409607
Epoch 181 train loss: 0.3797, eval loss 0.6967674493789673
Epoch 182 train loss: 0.3486, eval loss 0.6967872977256775
Epoch 183 train loss: 0.3722, eval loss 0.6967336535453796
Epoch 184 train loss: 0.3779, eval loss 0.6966606378555298
Epoch 185 train loss: 0.3582, eval loss 0.6965858936309814
Epoch 186 train loss: 0.3470, eval loss 0.6965110301971436
Epoch 187 train loss: 0.4054, eval loss 0.69647216796875
Epoch 188 train loss: 0.3881, eval loss 0.6963747143745422
Epoch 189 train loss: 0.3722, eval loss 0.6963286399841309
Epoch 190 train loss: 0.3838, eval loss 0.6963318586349487
Epoch 191 train loss: 0.3758, eval loss 0.6963492035865784
```

```
Epoch 192 train loss: 0.3838, eval loss 0.6961905360221863
Epoch 193 train loss: 0.4060, eval loss 0.6961612105369568
Epoch 194 train loss: 0.3974, eval loss 0.6961546540260315
Epoch 195 train loss: 0.3344, eval loss 0.6961306929588318
Epoch 196 train loss: 0.3740, eval loss 0.696093738079071
Epoch 197 train loss: 0.3811, eval loss 0.6960402131080627
Epoch 198 train loss: 0.3758, eval loss 0.69603431224823
Epoch 199 train loss: 0.3656, eval loss 0.6959965825080872
Epoch 200 train loss: 0.3456, eval loss 0.6959161758422852
Epoch 201 train loss: 0.3729, eval loss 0.6958968639373779
Epoch 202 train loss: 0.3748, eval loss 0.6958342790603638
Epoch 203 train loss: 0.3561, eval loss 0.6957604885101318
Epoch 204 train loss: 0.3578, eval loss 0.695676326751709
Epoch 205 train loss: 0.3732, eval loss 0.6956724524497986
Epoch 206 train loss: 0.3676, eval loss 0.6955326199531555
Epoch 207 train loss: 0.3880, eval loss 0.6954901814460754
Epoch 208 train loss: 0.3682, eval loss 0.6954352259635925
Epoch 209 train loss: 0.3400, eval loss 0.695487380027771
Epoch 210 train loss: 0.3686, eval loss 0.6954367756843567
Epoch 211 train loss: 0.3755, eval loss 0.6953930854797363
Epoch 212 train loss: 0.3781, eval loss 0.6952966451644897
Epoch 213 train loss: 0.3648, eval loss 0.6952895522117615
Epoch 214 train loss: 0.3667, eval loss 0.6953355669975281
Epoch 215 train loss: 0.3633, eval loss 0.6953233480453491
Epoch 216 train loss: 0.3454, eval loss 0.6952219009399414
Epoch 217 train loss: 0.3696, eval loss 0.6951655745506287
Epoch 218 train loss: 0.3677, eval loss 0.695144534111023
Epoch 219 train loss: 0.3739, eval loss 0.6951139569282532
Epoch 220 train loss: 0.3563, eval loss 0.6950494647026062
Epoch 221 train loss: 0.3429, eval loss 0.6950436234474182
Epoch 222 train loss: 0.3554, eval loss 0.6950441598892212
Epoch 223 train loss: 0.3822, eval loss 0.6950433254241943
Epoch 224 train loss: 0.3787, eval loss 0.6950032114982605
Epoch 225 train loss: 0.3714, eval loss 0.6949648857116699
Epoch 226 train loss: 0.3544, eval loss 0.6948496103286743
Epoch 227 train loss: 0.3506, eval loss 0.694853663444519
Epoch 228 train loss: 0.3899, eval loss 0.6948004961013794
Epoch 229 train loss: 0.3920, eval loss 0.6947653293609619
Epoch 230 train loss: 0.3788, eval loss 0.6947618722915649
Epoch 231 train loss: 0.3846, eval loss 0.6947186589241028
Epoch 232 train loss: 0.3370, eval loss 0.6947087645530701
Epoch 233 train loss: 0.3816, eval loss 0.6946312189102173
Epoch 234 train loss: 0.3475, eval loss 0.6946007013320923
Epoch 235 train loss: 0.3593, eval loss 0.6945194602012634
Epoch 236 train loss: 0.3559, eval loss 0.6945101022720337
Epoch 237 train loss: 0.3650, eval loss 0.6944644451141357
Epoch 238 train loss: 0.3730, eval loss 0.6944410800933838
Epoch 239 train loss: 0.3613, eval loss 0.6944094300270081
Epoch 240 train loss: 0.3413, eval loss 0.6943982839584351
Epoch 241 train loss: 0.3567, eval loss 0.6944191455841064
Epoch 242 train loss: 0.4288, eval loss 0.6944565176963806
Epoch 243 train loss: 0.3360, eval loss 0.694389283657074
Epoch 244 train loss: 0.4105, eval loss 0.6943188905715942
Epoch 245 train loss: 0.3701, eval loss 0.6942143440246582
Epoch 246 train loss: 0.3605, eval loss 0.6941839456558228
Epoch 247 train loss: 0.3427, eval loss 0.6941363215446472
Epoch 248 train loss: 0.3717, eval loss 0.6940972805023193
Epoch 249 train loss: 0.3809, eval loss 0.6940757632255554
Epoch 250 train loss: 0.3585, eval loss 0.6940659880638123
Epoch 251 train loss: 0.3696, eval loss 0.6940023303031921
Epoch 252 train loss: 0.3786, eval loss 0.6939437985420227
Epoch 253 train loss: 0.3627, eval loss 0.6939724683761597
Epoch 254 train loss: 0.3771, eval loss 0.6939838528633118
Epoch 255 train loss: 0.3571, eval loss 0.6939509510993958
```

```
Epoch 256 train loss: 0.3887, eval loss 0.6939136981964111
Epoch 257 train loss: 0.3741, eval loss 0.6938608884811401
Epoch 258 train loss: 0.3284, eval loss 0.6938488483428955
Epoch 259 train loss: 0.3679, eval loss 0.6937999725341797
Epoch 260 train loss: 0.3314, eval loss 0.6937827467918396
Epoch 261 train loss: 0.3512, eval loss 0.6937282681465149
Epoch 262 train loss: 0.3547, eval loss 0.6936829686164856
Epoch 263 train loss: 0.3562, eval loss 0.6936594843864441
Epoch 264 train loss: 0.3552, eval loss 0.6936770081520081
Epoch 265 train loss: 0.3551, eval loss 0.6936805844306946
Epoch 266 train loss: 0.3590, eval loss 0.6936435699462891
Epoch 267 train loss: 0.3738, eval loss 0.6936110258102417
Epoch 268 train loss: 0.3841, eval loss 0.693518340587616
Epoch 269 train loss: 0.3422, eval loss 0.6934815049171448
Epoch 270 train loss: 0.4048, eval loss 0.6934099793434143
Epoch 271 train loss: 0.3493, eval loss 0.6934046149253845
Epoch 272 train loss: 0.3465, eval loss 0.6934199333190918
Epoch 273 train loss: 0.3710, eval loss 0.6934213638305664
Epoch 274 train loss: 0.3445, eval loss 0.6934735178947449
```

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

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: 90.15%
F1: 68.41%
Precision: 63.43%
Recall: 74.23%
```

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

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

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

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

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

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

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

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

early_stopping_patience = 4
```

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

```
)
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=learning_rate,
    weight_decay=l2_reg
)
loss_fn = torch.nn.BCEWithLogitsLoss(pos_weight=torch.from_numpy(weights)[1]

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

steps_without_improvement = 0

best_val_loss = np.inf
best_model = None
best_threshold = None

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

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

        optimizer.step()
        optimizer.zero_grad()

    # model evaluation, early stopping
    model.eval()
    valid_metrics = evaluate_model(model, X_valid, y_valid, loss_fn)
    if valid_metrics['loss'] < best_val_loss:
        best_val_loss = valid_metrics['loss']
        best_threshold = valid_metrics['optimal_threshold']
        best_model = deepcopy(model)
        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 {vali
```

```
Epoch 0 train loss: 0.6277, eval loss 0.8210190534591675
Epoch 1 train loss: 0.5514, eval loss 0.8167227506637573
Epoch 2 train loss: 0.5999, eval loss 0.8156638741493225
Epoch 3 train loss: 0.5656, eval loss 0.814005434513092
Epoch 4 train loss: 0.5424, eval loss 0.8138073086738586
Epoch 5 train loss: 0.5043, eval loss 0.8133698105812073
Epoch 6 train loss: 0.6245, eval loss 0.8123372793197632
Epoch 7 train loss: 0.5839, eval loss 0.8117238879203796
Epoch 8 train loss: 0.5309, eval loss 0.8113364577293396
Epoch 9 train loss: 0.5642, eval loss 0.8105191588401794
Epoch 10 train loss: 0.5257, eval loss 0.809950053691864
Epoch 11 train loss: 0.5303, eval loss 0.8102796673774719
Epoch 12 train loss: 0.4729, eval loss 0.8091976642608643
Epoch 13 train loss: 0.4526, eval loss 0.8095315098762512
Epoch 14 train loss: 0.4891, eval loss 0.809403657913208
Epoch 15 train loss: 0.5157, eval loss 0.8095149993896484
```

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

```
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: 90.77%
F1: 69.73%
Precision: 61.89%
Recall: 79.85%
```

Pytania kontrolne (1 punkt)

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

Odpowiedzi

1. 4 najważniejsze hiperparametry:

- liczba warstw ukrytych (głębokość sieci neuronowej),
- liczba neuronów każdej z warstw (szerokość sieci neuronowej),
- rodzaj wykorzystanej funkcji aktywacji,
- rodzaj wykorzystanego optymalizatora,

1. Problem w wykorzystaniu regularyzacji L1 (LASSO):

- w przypadku, gdy mamy więcej cech niż obserwacji (problem wielowymiarowy), LASSO zachowa maksymalnie N różnych cech, gdzie N jest liczbą obserwacji,
- regularyzacja L1 zazwyczaj wybiera jedną spośród grupy silnie skorelowanych ze sobą cech, podczas gdy w większości problemów, silnie skorelowane cechy powinny zostać wybrane lub odrzucone jako grupa,
- z powodów opisanych w dwóch powyższych podpunktach, LASSO może doprowadzić do utraty cennych informacji zawartych w wykorzystywanych do treningu modelu danych, co doprowadzi do nieprawidłowych predykcji,

1. Nie da takiego samego modelu końcowego, ponieważ różne metryki zakończą trening modelu po różnej liczbie epok, przez co otrzymamy inny model. Kryterium stopu w metrykach może się różnić, przez co trening modelu może zakończyć się podczas innej epoki.

Akceleracja sprzętowa (dla zainteresowanych)

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

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

In [ ]:
```
import time

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

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

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

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: {t
            time_from_eval = time.time()

        step_counter += 1

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

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

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/Users/mateu/Education/Studies/Term5/basics-of-artificial-intelligence/lab3/lab_3.ipynb Cell 86 in <cell line: 3>()
      <a href='vscode-notebook-cell:/Users/mateu/Education/Studies/Term5/basics-of-artificial-intelligence/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=0'>1</a> import time
----> <a href='vscode-notebook-cell:/Users/mateu/Education/Studies/Term5/basics-of-artificial-intelligence/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=2'>3</a> model = NormalizingMLP(
      <a href='vscode-notebook-cell:/Users/mateu/Education/Studies/Term5/basics-of-artificial-intelligence/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=3'>4</a>     input_size=X_train.shape[1],
      <a href='vscode-notebook-cell:/Users/mateu/Education/Studies/Term5/basics-of-artificial-intelligence/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=4'>5</a>     dropout_p=dropout_p
      <a href='vscode-notebook-cell:/Users/mateu/Education/Studies/Term5/basics-of-artificial-intelligence/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=5'>6</a> ).to('cuda')
      <a href='vscode-notebook-cell:/Users/mateu/Education/Studies/Term5/basics-of-artificial-intelligence/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/mateu/Education/Studies/Term5/basics-of-artificial-intelligence/lab3/lab_3.ipynb#Y151sZmlsZQ%3D%3D?line=9'>10</a> # note that we are using loss function with sigmoid built in

File ~/opt/anaconda3/lib/python3.9/site-packages/torch/nn/modules/module.py:989, in Module.to(self, *args, **kwargs)
    985             return t.to(device, dtype if t.is_floating_point() or t.is_complex() else None,
    986                         non_blocking, memory_format=convert_to_format)
    987     return t.to(device, dtype if t.is_floating_point() or t.is_complex() else None, non_blocking)
--> 989 return self._apply(convert)

File ~/opt/anaconda3/lib/python3.9/site-packages/torch/nn/modules/module.py:641, in Module._apply(self, fn)
    639 def _apply(self, fn):
    640     for module in self.children():
--> 641         module._apply(fn)
    643     def compute_should_use_set_data(tensor, tensor_applied):
    644         if torch._has_compatible_shallow_copy_type(tensor, tensor_applied):
    645             # If the new tensor has compatible tensor type as the existing tensor,
    646             # the current behavior is to change the tensor in-place using `.data =`,
   (...)
    651             # global flag to let the user control whether they want the future
    652             # behavior of overwriting the existing tensor or not.

File ~/opt/anaconda3/lib/python3.9/site-packages/torch/nn/modules/module.py:641, in Module._apply(self, fn)
    639 def _apply(self, fn):
    640     for module in self.children():
--> 641         module._apply(fn)
    643     def compute_should_use_set_data(tensor, tensor_applied):
    644         if torch._has_compatible_shallow_copy_type(tensor, tensor_applied):
    645             # If the new tensor has compatible tensor type as the existing tensor,
    646             # the current behavior is to change the tensor in-place
```

```
 using `.data =`,
   (...)
    651             # global flag to let the user control whether they want
 the future
    652             # behavior of overwriting the existing tensor or not.

File ~/opt/anaconda3/lib/python3.9/site-packages/torch/nn/modules/module.py:
664, in Module._apply(self, fn)
    660 # Tensors stored in modules are graph leaves, and we don't want to
    661 # track autograd history of `param_applied`, so we have to use
    662 # `with torch.no_grad():`
    663 with torch.no_grad():
--> 664     param_applied = fn(param)
    665 should_use_set_data = compute_should_use_set_data(param, param_appli
ed)
    666 if should_use_set_data:

File ~/opt/anaconda3/lib/python3.9/site-packages/torch/nn/modules/module.py:
987, in Module.to.<locals>.convert(t)
    984 if convert_to_format is not None and t.dim() in (4, 5):
    985     return t.to(device, dtype if t.is_floating_point() or t.is_compl
ex() else None,
    986                 non_blocking, memory_format=convert_to_format)
--> 987 return t.to(device, dtype if t.is_floating_point() or t.is_complex()
else None, non_blocking)

File ~/opt/anaconda3/lib/python3.9/site-packages/torch/cuda/__init__.py:221,
in _lazy_init()
    217     raise RuntimeError(
    218         "Cannot re-initialize CUDA in forked subprocess. To use CUDA
with "
    219         "multiprocessing, you must use the 'spawn' start method")
    220 if not hasattr(torch._C, '_cuda_getDeviceCount'):
--> 221     raise AssertionError("Torch not compiled with CUDA enabled")
    222 if _cudart is None:
    223     raise AssertionError(
    224         "libcudart functions unavailable. It looks like you have a b
roken build?")

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