Sieci neuronowe

Wstęp

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

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

Wykorzystywane biblioteki

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

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

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

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

Konfiguracja własnego komputera

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

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

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

In [1]:
```
# for conda users
!conda install -y matplotlib pandas pytorch torchvision -c pytorch -c conda-forge
```

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

## Package Plan ##

  environment location: G:\anaconda3

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


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    libjpeg-turbo-2.0.0        |       h196d8e1_0         618 KB
    libuv-1.44.2               |       h8ffe710_0         362 KB  conda-forge
    mpmath-1.3.0               |     pyhd8ed1ab_0         428 KB  conda-forge
    networkx-3.2.1             |     pyhd8ed1ab_0         1.1 MB  conda-forge
    pytorch-2.1.0              |     py3.11_cpu_0       166.3 MB  pytorch
    pytorch-mutex-1.0          |             cpu           3 KB  pytorch
    sympy-1.12                 |     pyh04b8f61_3         4.0 MB  conda-forge
    torchvision-0.16.0         |        py311_cpu         6.9 MB  pytorch
    ------------------------------------------------------------
                                           Total:       179.7 MB

The following NEW packages will be INSTALLED:

  libjpeg-turbo      pkgs/main/win-64::libjpeg-turbo-2.0.0-h196d8e1_0
  libuv              conda-forge/win-64::libuv-1.44.2-h8ffe710_0
  matplotlib         pkgs/main/win-64::matplotlib-3.8.0-py311haa95532_0
  mpmath             conda-forge/noarch::mpmath-1.3.0-pyhd8ed1ab_0
  networkx           conda-forge/noarch::networkx-3.2.1-pyhd8ed1ab_0
  pytorch            pytorch/win-64::pytorch-2.1.0-py3.11_cpu_0
  pytorch-mutex      pytorch/noarch::pytorch-mutex-1.0-cpu
  sympy              conda-forge/noarch::sympy-1.12-pyh04b8f61_3
  torchvision        pytorch/win-64::torchvision-0.16.0-py311_cpu

The following packages will be SUPERSEDED by a higher-priority channel:
```

```
  ca-certificates    pkgs/main::ca-certificates-2023.08.22~ --> conda-forge::ca-certific
ates-2023.7.22-h56e8100_0
  certifi            pkgs/main/win-64::certifi-2023.7.22-p~ --> conda-forge/noarch::cert
ifi-2023.7.22-pyhd8ed1ab_0



Downloading and Extracting Packages: ...working... done
Preparing transaction: ...working... done
Verifying transaction: ...working... done
Executing transaction: ...working... done

==> WARNING: A newer version of conda exists. <==
  current version: 23.9.0
  latest version: 23.10.0

Please update conda by running

    $ conda update -n base -c defaults conda

Or to minimize the number of packages updated during conda update use

     conda install conda=23.10.0
```

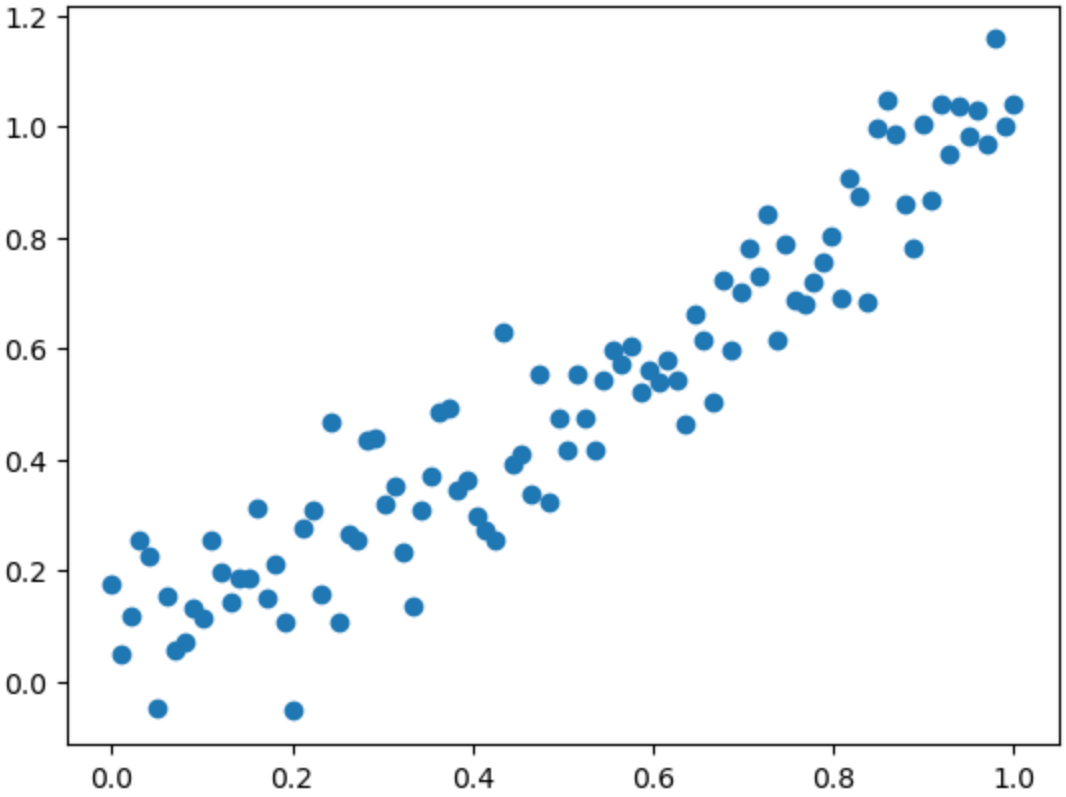
Wprowadzenie

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

In [2]:
```
from typing import Tuple, Dict

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

In [3]:
```
np.random.seed(0)

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

plt.scatter(x, y)
```

Out[3]:
```
<matplotlib.collections.PathCollection at 0x26ec8ff4c50>
```



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

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

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

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

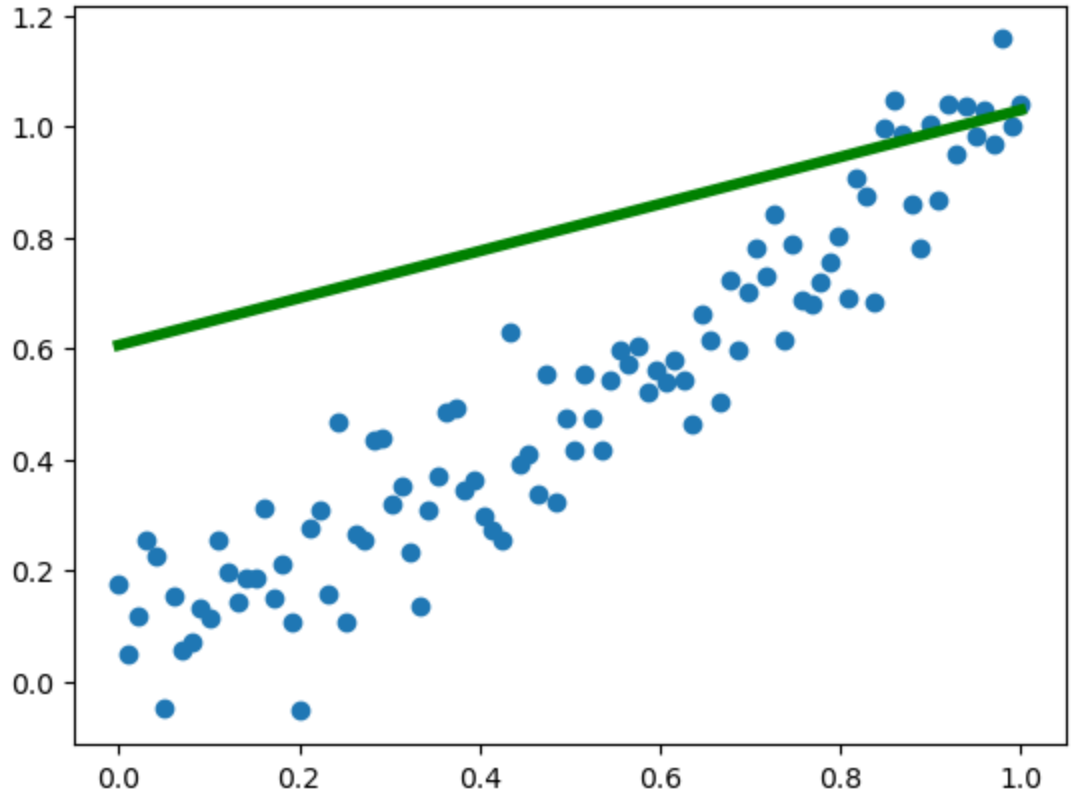
Zadanie 1 (0.5 punkt)

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

In [4]:
```
def mse(y: np.ndarray, y_hat: np.ndarray) -> float:
    return (np.square(y - y_hat)).sum() / len(y)
```

In [5]:
```
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[5]:
```
MSE: 0.133
[<matplotlib.lines.Line2D at 0x26ec900fc90>]
```



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

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

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

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

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

Przyglądając się temu równaniu widzimy, że:

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

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

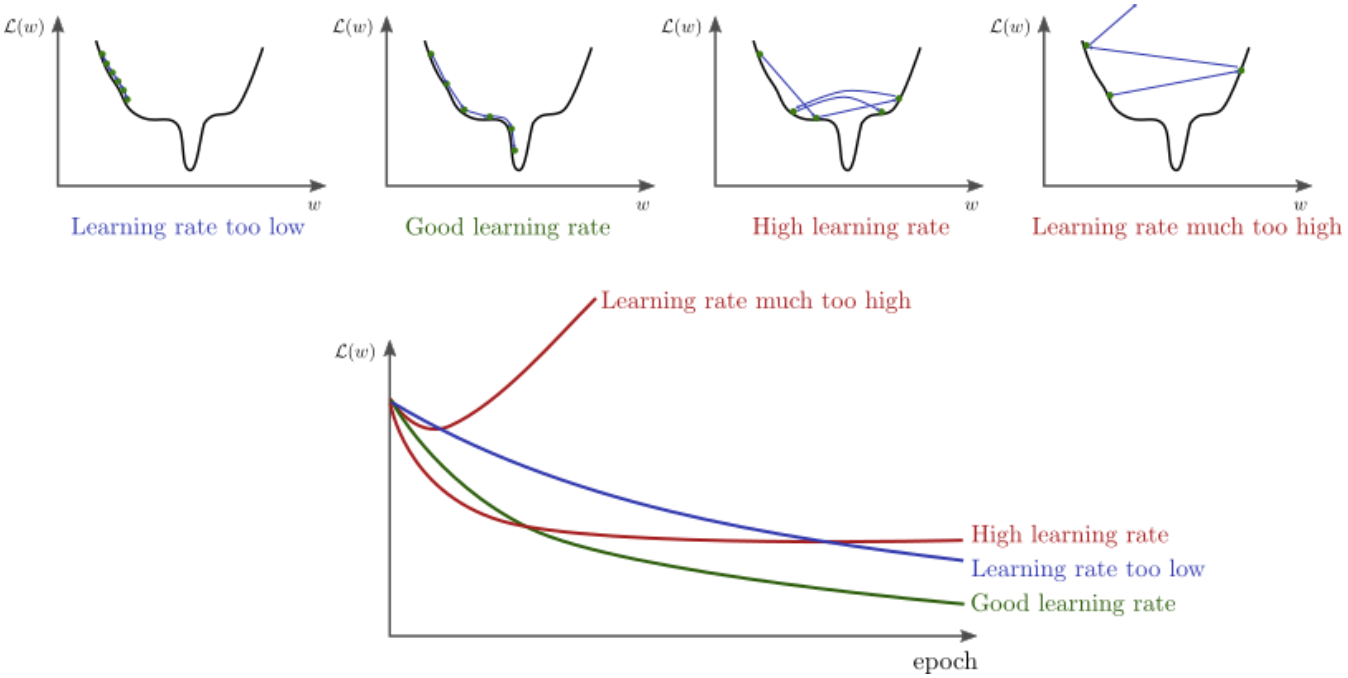
Stosując powyższe do optymalizacji, mamy:

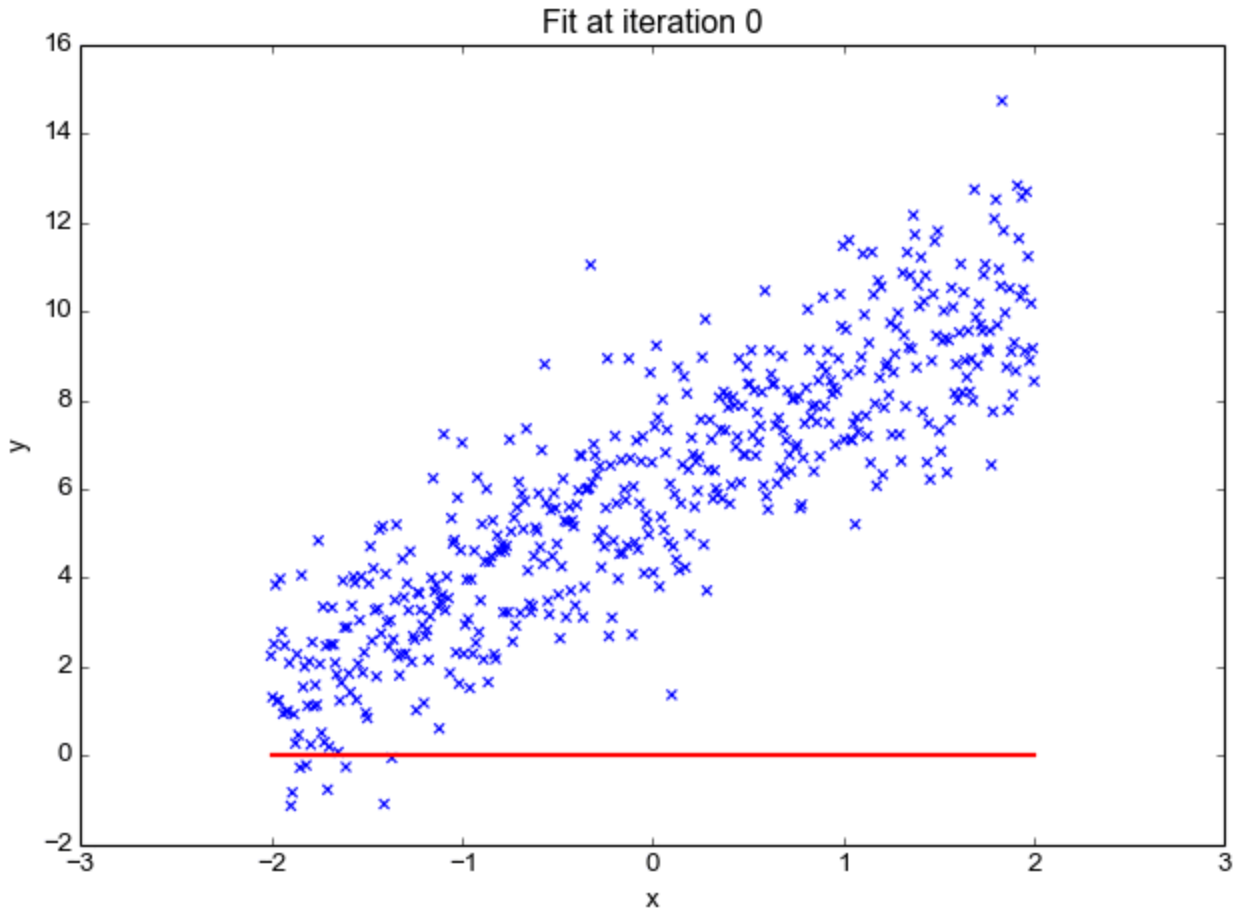
$$x_{t+1} = x_t - \alpha * \frac{f(x)}{dx}$$

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





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

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

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

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

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

Policzmy te pochodne po kolei:

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

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

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

Łącząc powyższe wyniki dostaniemy:

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

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

Aktualizacja parametrów wygląda tak:

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

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

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

Zadanie 2 (1.0 punkt)

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

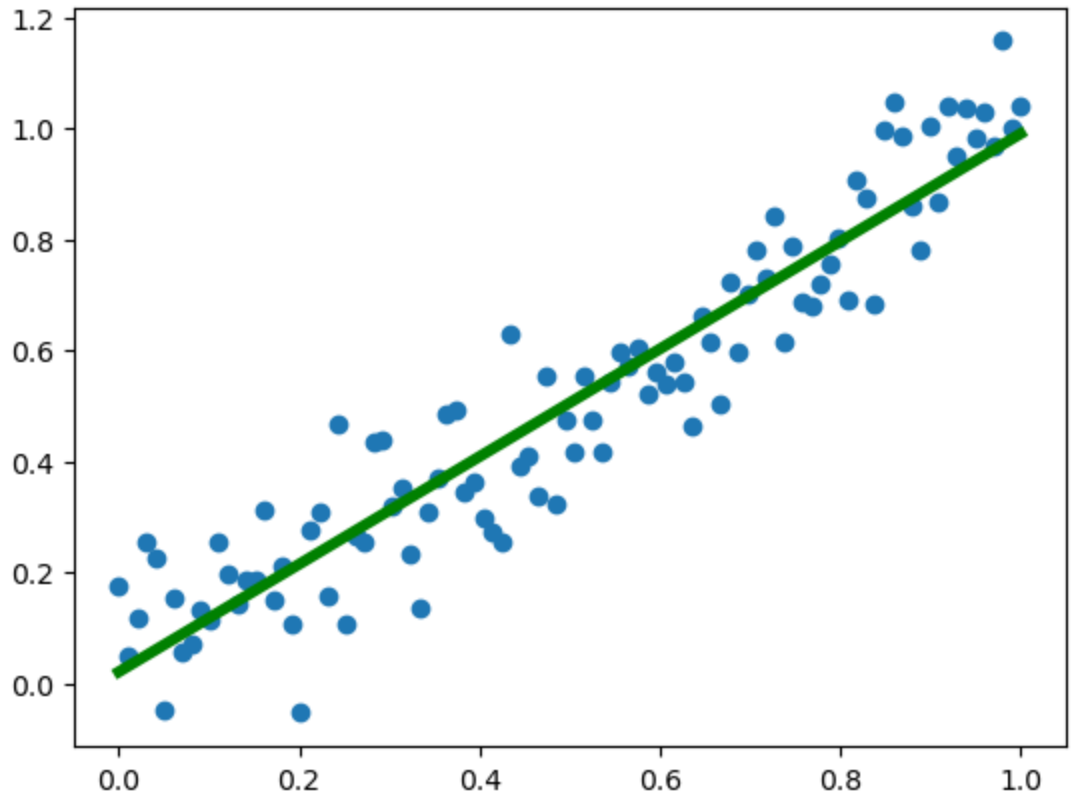
In [6]:
```
def optimize(
    x: np.ndarray, y: np.ndarray, a: float, b: float, learning_rate: float = 0.1
):
    y_hat = a * x + b
    errors = y - y_hat
    n = len(x)
    new_a = a - learning_rate * ((-2/n) * sum(x * errors))
    new_b = b - learning_rate * ((-2/n) * sum(errors))
    return new_a, new_b
```

In [7]:
```
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:  0.012673197778527 68
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:  0.010083108773135263
step 800 loss:  0.010083108550709076
step 900 loss:  0.01008310853576281
final loss: 0.010083108534760455
```

In [8]:
```
plt.scatter(x, y)
plt.plot(x, a * x + b, color="g", linewidth=4)
```

Out[8]:
```
[<matplotlib.lines.Line2D at 0x26ec9026190>]
```



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

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

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

Wprowadzenie do PyTorcha

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

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

Samo API PyTorcha bardzo przypomina Numpy'a, a podstawowym obiektem jest `Tensor`, klasa reprezentująca tensory dowolnego wymiaru. Dodatkowo niektóre tensory będą miały automatycznie

obliczony gradient. Co ważne, tensor jest na pewnym urządzeniu, CPU lub GPU, a przenosić między nimi trzeba explicite.

Najważniejsze moduły:

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

In [9]:
```
import torch
import torch.nn as nn
import torch.optim as optim
```

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

# elementwise sum
print(ones + noise)

# elementwise multiplication
print(ones * noise)

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

```
tensor([1.8119, 1.0318, 1.5833, 1.2463, 1.1166, 1.2324, 1.0452, 1.4340, 1.2517,
        1.0776])
tensor([0.8119, 0.0318, 0.5833, 0.2463, 0.1166, 0.2324, 0.0452, 0.4340, 0.2517,
        0.0776])
tensor(2.8307)
```

In [11]:
```
# 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 [12]:
```
a = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)
a, b
```

Out[12]:
```
(tensor([0.8657], requires_grad=True), tensor([0.1288], requires_grad=True))
```

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

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

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

In [13]:
```
mse = nn.MSELoss()
mse(y, a * x + b)
```

Out[13]:
```
tensor(0.0141, 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 [14]:
```
loss = mse(y, a * x + b)
loss.backward()
```

In [15]:
```
print(a.grad)
```

```
tensor([0.0379])
```

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 [16]:
```
loss = mse(y, a * x + b)
loss.backward()
a.grad
```

Out[16]:
```
tensor([0.0758])
```

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

In [17]:
```
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(0.0141, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 100 loss:  tensor(0.0102, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 200 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 300 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 400 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 500 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 600 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 700 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
step 800 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
```

```
step 900 loss:  tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
final loss: tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
```

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

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

In [18]:
```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: 0.3662
step 100 loss: 0.0126
step 200 loss: 0.0103
step 300 loss: 0.0101
step 400 loss: 0.0101
step 500 loss: 0.0101
step 600 loss: 0.0101
step 700 loss: 0.0101
step 800 loss: 0.0101
step 900 loss: 0.0101
final loss: tensor(0.0101, dtype=torch.float64, grad_fn=<MseLossBackward0>)
```

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

Funkcją kosztu zwykle jest **entropia krzyżowa (cross-entropy)**, stosowana też w klasycznej regresji logistycznej. Co ważne, sieci neuronowe, nawet tak proste, uczą się szybciej i stabilniej, gdy dane na wejściu

(a przynajmniej zmienne numeryczne) są **ustandaryzowane (standardized)**. Operacja ta polega na odjęciu średniej i podzieleniu przez odchylenie standardowe (tzw. *Z-score transformation*).

Uwaga - PyTorch wymaga tensora klas będącego liczbami zmiennoprzecinkowymi!

Zbiór danych

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

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

In [19]:
```
!wget https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data
```

In [19]:
```
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, W
fnlwgt: continuous.
education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9
education-num: continuous.
marital-status: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married
occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-spec
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
"""

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

Out[19]:
```
array([' <=50K', ' >50K'], dtype=object)
```

In [20]:
```
# attribution: https://www.kaggle.com/code/royshih23/topic7-classification-in-python
```

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

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

In [26]:

```
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', 'ho
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[26]:
```
((20838, 108), (20838,))
```

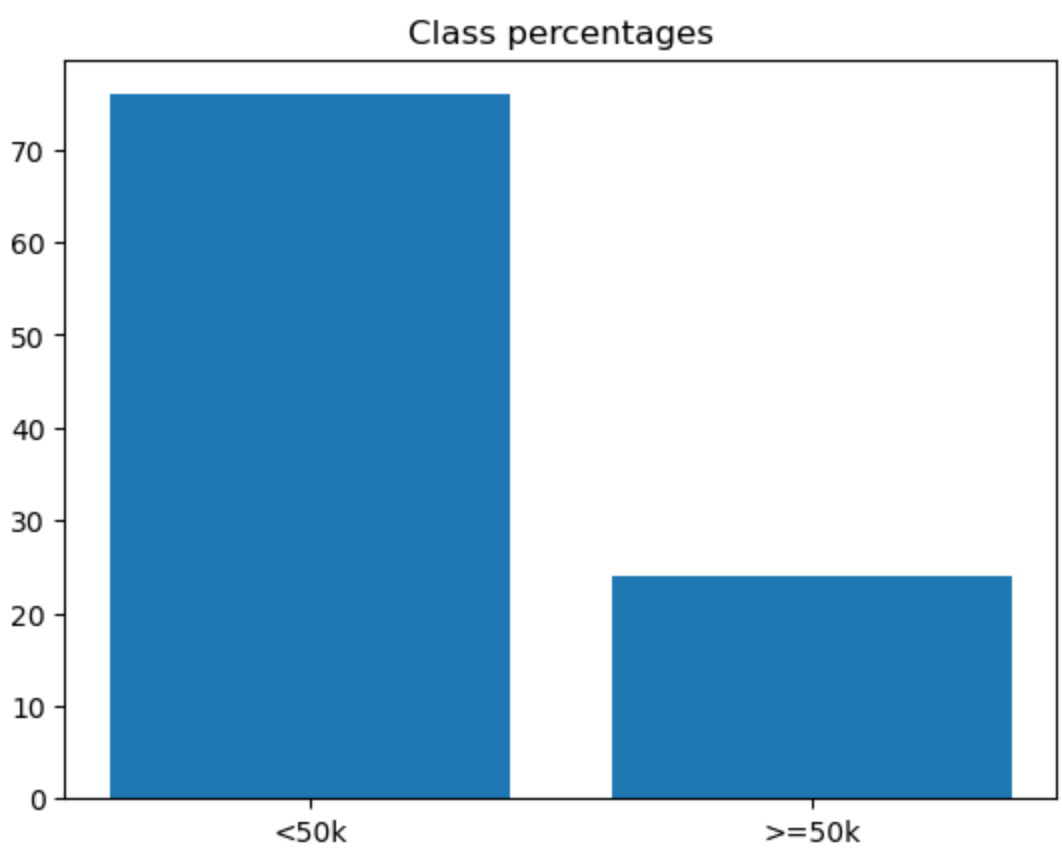
Uwaga co do typów - PyTorchu wszystko w sieci neuronowej musi być typu `float32`. W szczególności trzeba uważać na konwersje z Numpy'a, który używa domyślnie typu `float64`. Może ci się przydać metoda `.float()`.

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

In [27]:
```
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 [28]:
```
import matplotlib.pyplot as plt

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

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



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

Zadanie 3 (1.0 punkt)

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

Sama sieć składa się z 2 elementów:

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

Użyj binarnej entropii krzyżowej `nn.BCELoss` jako funkcji kosztu. Użyj optymalizatora SGD ze stałą uczącą `1e-3`. Trenuj przez 3000 epok. Pamiętaj, aby przekazać do optymalizatora `torch.optim.SGD` parametry sieci (metoda `.parameters()`). Dopisz logowanie kosztu raz na 100 epok.

In [29]:
```
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()

for t in range(3000):
    y_pred = activation(model(X_train))

    loss = loss_fn(y_pred, y_train)
    if t % 100 == 99: print(t, loss.item())
    loss.backward()

    optimizer.step()
    optimizer.zero_grad()

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

```
99 0.6438173055648804
199 0.617991030216217
299 0.5963792204856873
399 0.5781421661376953
499 0.5626155138015747
599 0.5492758750915527
699 0.5377103090286255
799 0.5275936722755432
899 0.5186682939529419
999 0.5107294321060181
1099 0.5036137104034424
1199 0.4971902370452881
1299 0.49135297536849976
1399 0.486016184091568
1499 0.4811096787452698
1599 0.47657570242881775
1699 0.47236689925193787
1799 0.468443363904953
1899 0.4647720456123352
1999 0.4613250195980072
2099 0.4580785632133484
2199 0.45501261949539185
2299 0.45210978388786316
2399 0.4493551254272461
2499 0.44673585891723633
2599 0.44424062967300415
2699 0.44185957312583923
2799 0.43958383798599243
2899 0.4374057948589325
2999 0.4353184998035431
final loss: 0.4353
```

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 [30]:
```python
from sklearn.metrics import precision_recall_curve, precision_recall_fscore_support, roc


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: 84.86%
```

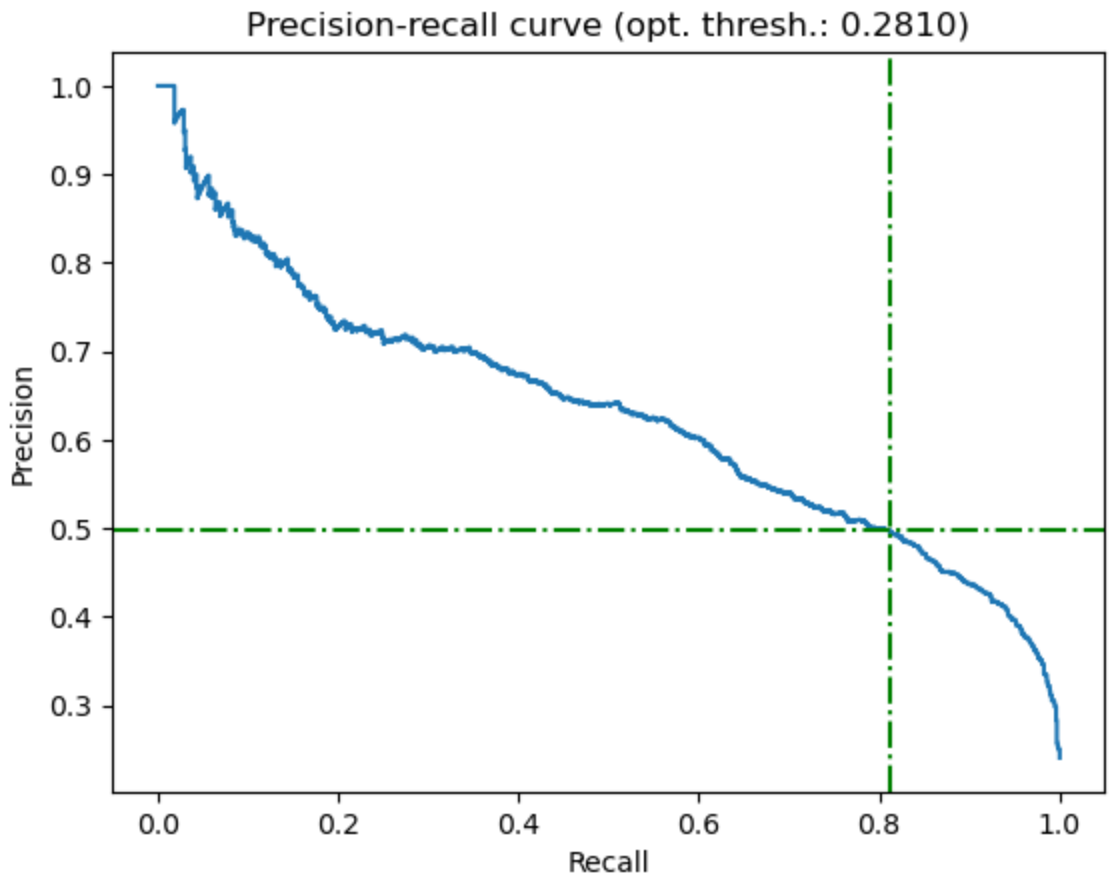
Jest to całkiem dobry wynik, a może być jeszcze lepszy. Sprawdźmy dla pewności jeszcze inne metryki: precyzję, recall oraz F1-score. Dodatkowo narysujemy krzywą precision-recall, czyli jak zmieniają się te metryki w zależności od przyjętego progu (threshold) prawdopodobieństwa, powyżej którego przyjmujemy 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 [34]:
```python
from sklearn.metrics import PrecisionRecallDisplay


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

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

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

    return optimal_idx, optimal_threshold


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

    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 [35]:
```python
model.eval()
with torch.no_grad():
    y_pred_valid_score = activation(model(X_valid))

plot_precision_recall_curve(y_valid, y_pred_valid_score)
```



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

Sieci neuronowe

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

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

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

Na tym laboratorium skupimy się na najprostszej architekturze, czyli MLP. Jest ona powszechnie łączona z wszelkimi innymi architekturami, bo pozwala dokonywać klasyfikacji i regresji. Przykładowo, klasyfikacja

obrazów to zwykle CNN + MLP, klasyfikacja tekstów to transformer + MLP, a regresja na grafach to GNN + MLP.

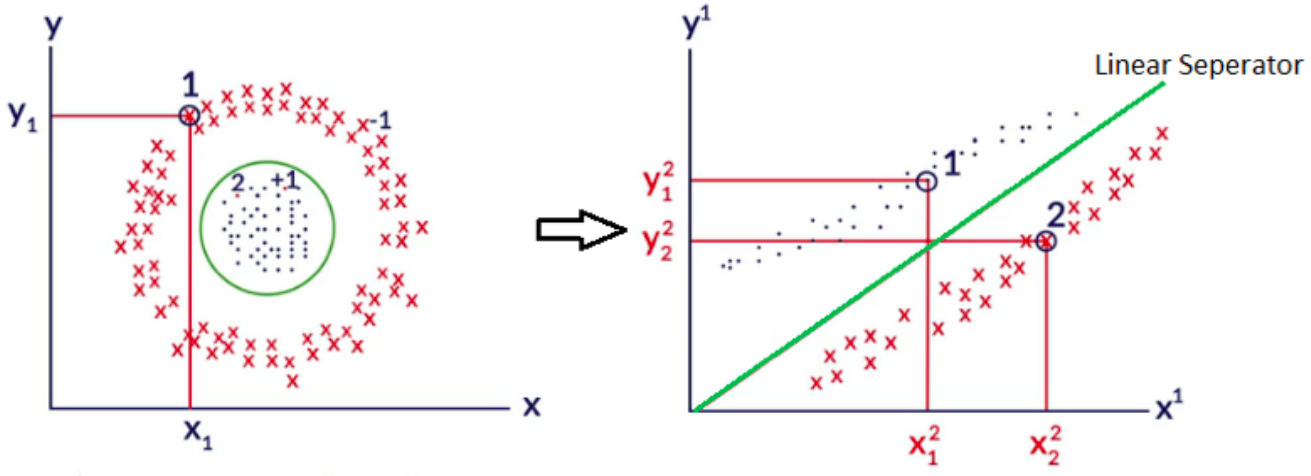
Dodatkowo, pomimo prostoty MLP są bardzo potężne - udowodniono, że perceptrony (ich powszechna nazwa) są uniwersalnym aproksymatorem, będącym w stanie przybliżyć dowolną funkcję z odpowiednio małym błędem, zakładając wystarczającą wielkość warstw sieci. Szczególne ich wersje potrafią nawet reprezentować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

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

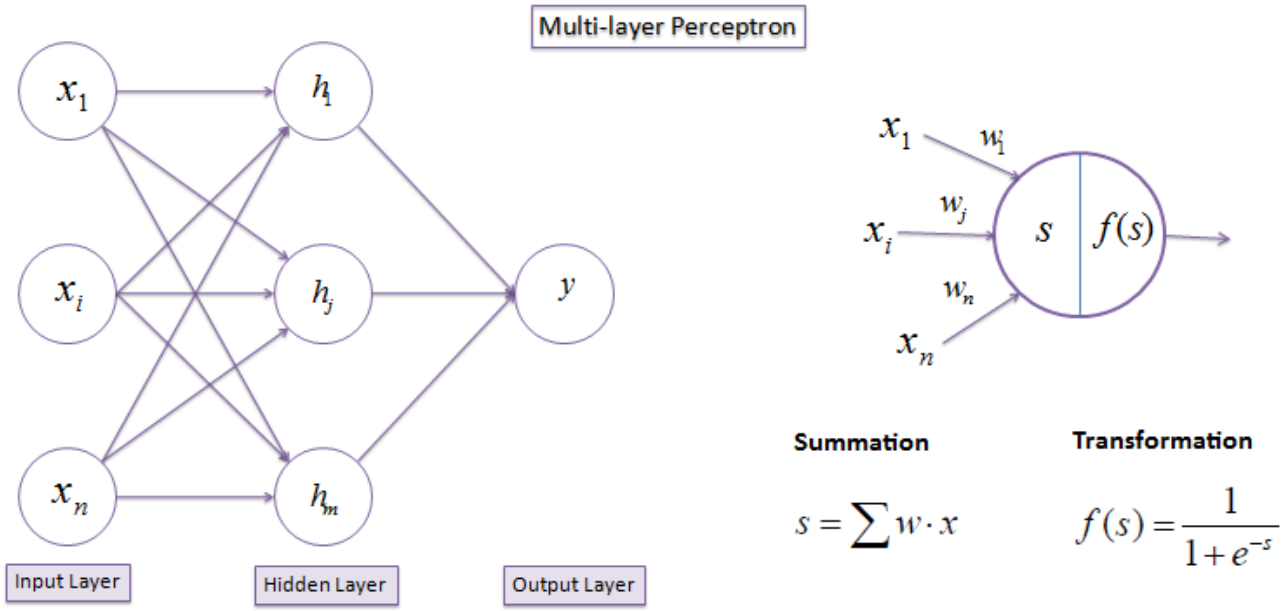
Sieci MLP

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

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



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



Zapisane matematycznie MLP to:

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

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

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

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

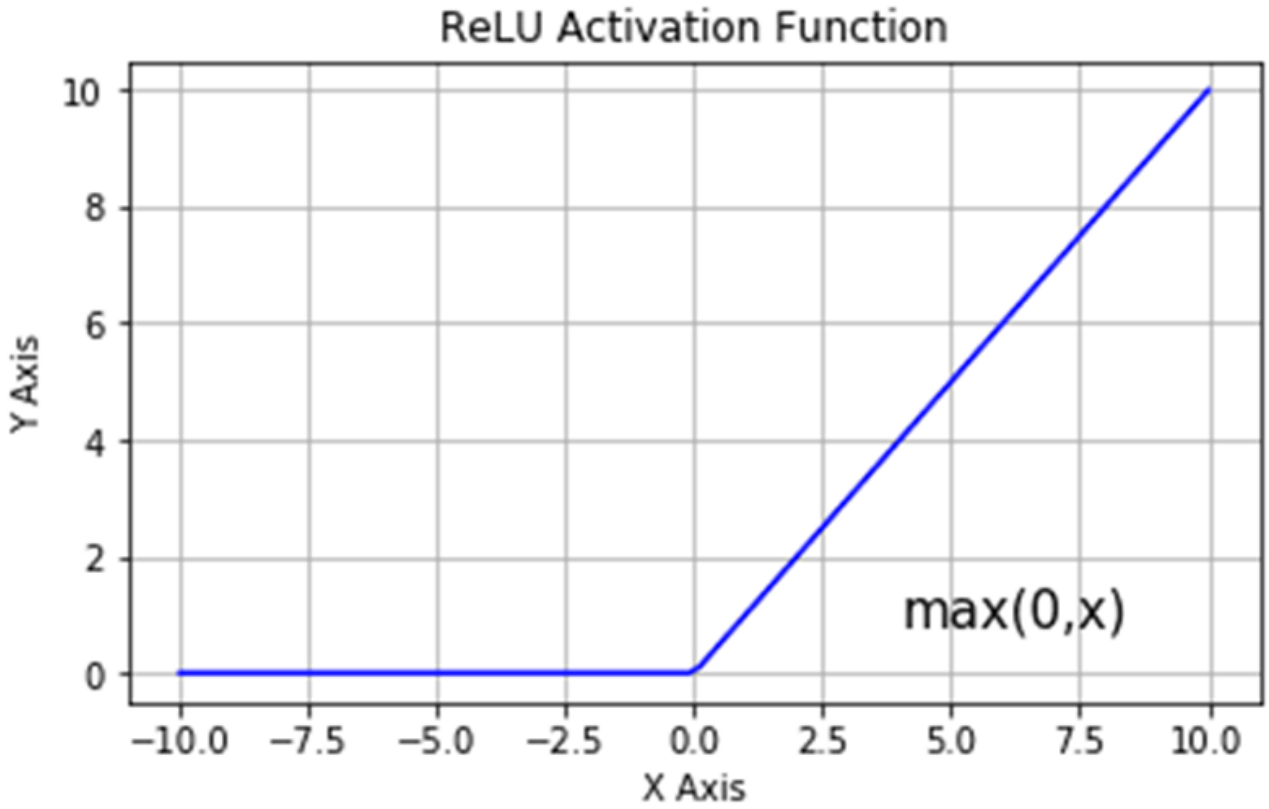
gdzie w_i to jest parametr i-tej warstwy sieci, x to są dane (w naszym przypadku jedna liczba) wejściowa, a h_i to wyjście i-tej warstwy.

Jak widać, taka sieć o n warstwach jest równoważna sieci o jednej warstwie z parametrem $w = w_1w_2 \dots w_n$. Wynika to z tego, że złożenie funkcji liniowych jest także funkcją liniową - patrz notatki z algebry :)

Jeżeli natomiast użyjemy nieliniowej funkcji aktywacji, często oznaczanej jako 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. Każdy atrybut dziedziczący po `nn.Module` lub `nn.Parameter` jest uważany za taki, który zawiera parametry sieci, a więc przy wywołaniu metody `parameters()` - parametry z tych atrybutów pojawią się w liście wszystkich parametrów. Musimy też zdefiniować metodę `forward()`, która przyjmuje tensor `x` i zwraca wynik. Typowo ta metoda po prostu używa obiektów zdefiniowanych w konstruktorze.

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

Zadanie 4 (0.5 punktu)

Uzupełnij implementację 3-warstwowej sieci MLP. Użyj rozmiarów:

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

Użyj funkcji aktywacji ReLU.

Przydatne klasy:

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

In [36]:
```
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, threshold: float = 0.5):
        y_pred_score = self.predict_proba(x)
        return (y_pred_score > threshold).to(torch.int32)
```

In [37]:
```
learning_rate = 1e-3
model = MLP(input_size=X_train.shape[1])
```

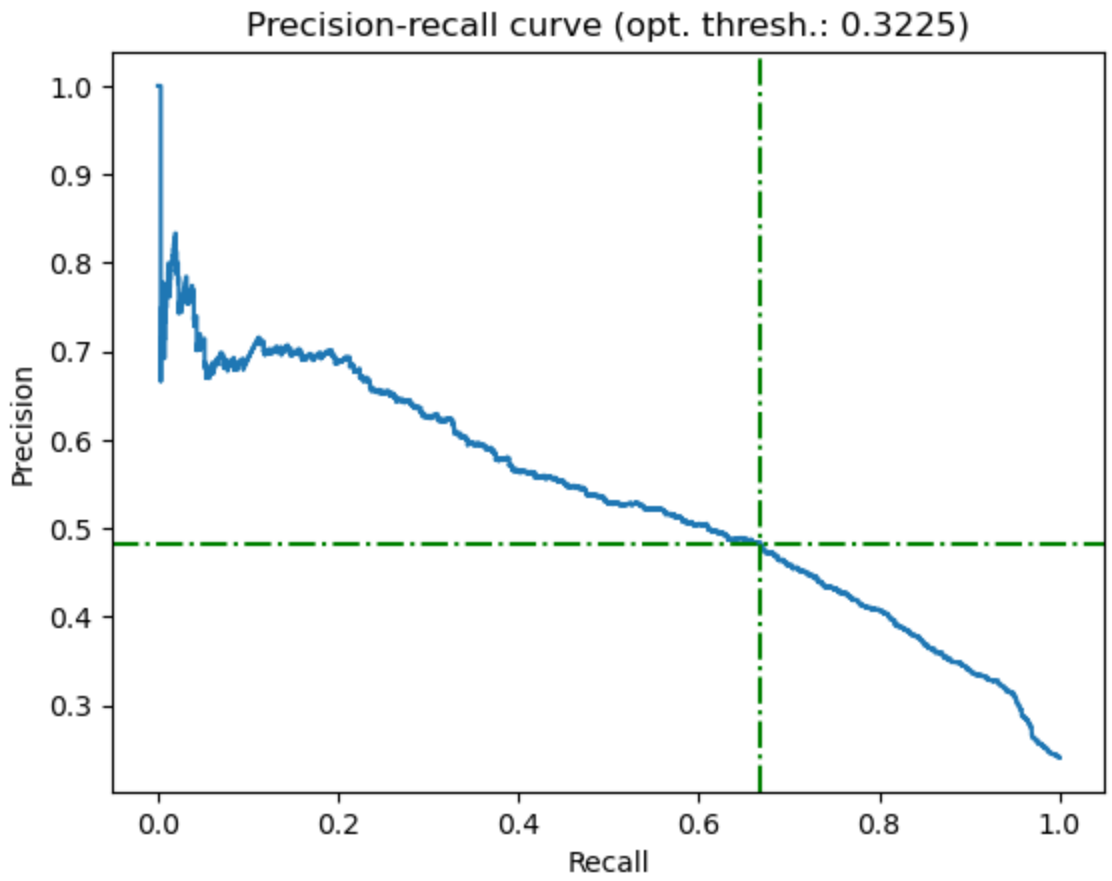
```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# 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: 0.6712
Epoch 200 train loss: 0.6509
Epoch 400 train loss: 0.6332
Epoch 600 train loss: 0.6175
Epoch 800 train loss: 0.6037
Epoch 1000 train loss: 0.5914
Epoch 1200 train loss: 0.5805
Epoch 1400 train loss: 0.5709
Epoch 1600 train loss: 0.5623
Epoch 1800 train loss: 0.5547
final loss: 0.5479
```

In [38]:

```
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: 77.68%
```



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

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

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

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

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

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

Dodatkowo ryzyko przeuczenia można zmniejszyć, używając mniejszej stałej uczącej.

Zadanie 5 (1.5 punktu)

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

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

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

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

In [39]:
```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)

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

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

    y_pred = (y_pred_score>threshold).float()

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

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

Zadanie 6 (0.5 punktu)

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

In [40]:
```
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, threshold: float = 0.5):
        y_pred_score = self.predict_proba(x)
        return (y_pred_score > threshold).to(torch.int32)
```

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

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

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

Na razie użyjemy jednak minibatch SGD.

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

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

In [41]:
```
from torch.utils.data import Dataset


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

```
        super().__init__()

        self.data = data
        self.y = y

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

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

Zadanie 7 (1.5 punktu)

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

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

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

In [42]:
```
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 [43]:
```
model = RegularizedMLP(
    input_size=X_train.shape[1],
    dropout_p=dropout_p
)
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=learning_rate,
    weight_decay=l2_reg
)
loss_fn = torch.nn.BCEWithLogitsLoss()

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

steps_without_improvement = 0

best_val_loss = np.inf
best_model = None
best_threshold = None

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

    # note that we are using DataLoader to get batches
```

```
    for X_batch, y_batch in train_dataloader:
        y_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"]
        steps_without_improvement = 0
        best_model = deepcopy(model)
        best_threshold = valid_metrics["optimal_threshold"]
    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['l
```

```
tensor([[0.4752],
        [0.4724],
        [0.4760],
        ...,
        [0.4705],
        [0.4707],
        [0.4707]])
Epoch 0 train loss: 0.6514, eval loss 0.8422772884368896
tensor([[0.4566],
        [0.4564],
        [0.4591],
        ...,
        [0.4515],
        [0.4523],
        [0.4517]])
Epoch 1 train loss: 0.6422, eval loss 0.8354431986808777
tensor([[0.4389],
        [0.4412],
        [0.4431],
        ...,
        [0.4333],
        [0.4346],
        [0.4334]])
Epoch 2 train loss: 0.6247, eval loss 0.8289394974708557
tensor([[0.4221],
        [0.4271],
        [0.4280],
        ...,
        [0.4159],
        [0.4177],
        [0.4157]])
Epoch 3 train loss: 0.6067, eval loss 0.8227604627609253
tensor([[0.4061],
        [0.4140],
        [0.4139],
        ...,
        [0.3993],
        [0.4012],
        [0.3986]])
Epoch 4 train loss: 0.5955, eval loss 0.8168655037879944
tensor([[0.3909],
        [0.4019],
        [0.4008],
        ...,
```

```
        [0.3836],
        [0.3852],
        [0.3823]])
Epoch 5 train loss: 0.5906, eval loss 0.8112855553627014
tensor([[0.3764],
        [0.3907],
        [0.3885],
        ...,
        [0.3685],
        [0.3695],
        [0.3668]])
Epoch 6 train loss: 0.5781, eval loss 0.8059895634651184
tensor([[0.3624],
        [0.3806],
        [0.3770],
        ...,
        [0.3543],
        [0.3544],
        [0.3520]])
Epoch 7 train loss: 0.5656, eval loss 0.8009659647941589
tensor([[0.3493],
        [0.3716],
        [0.3664],
        ...,
        [0.3408],
        [0.3399],
        [0.3382]])
Epoch 8 train loss: 0.5676, eval loss 0.7962377071380615
tensor([[0.3373],
        [0.3637],
        [0.3570],
        ...,
        [0.3281],
        [0.3261],
        [0.3254]])
Epoch 9 train loss: 0.5547, eval loss 0.7918524742126465
tensor([[0.3260],
        [0.3567],
        [0.3484],
        ...,
        [0.3162],
        [0.3129],
        [0.3132]])
Epoch 10 train loss: 0.5441, eval loss 0.7877190113067627
tensor([[0.3156],
        [0.3507],
        [0.3406],
        ...,
        [0.3053],
        [0.3004],
        [0.3019]])
Epoch 11 train loss: 0.5403, eval loss 0.7838712930679321
tensor([[0.3062],
        [0.3458],
        [0.3339],
        ...,
        [0.2952],
        [0.2888],
        [0.2915]])
Epoch 12 train loss: 0.5339, eval loss 0.7803198099136353
tensor([[0.2977],
        [0.3418],
        [0.3280],
        ...,
        [0.2860],
        [0.2779],
```

```
        [0.2820]])
Epoch 13 train loss: 0.5321, eval loss 0.7770267128944397
tensor([[0.2898],
        [0.3386],
        [0.3227],
        ...,
        [0.2774],
        [0.2676],
        [0.2731]])
Epoch 14 train loss: 0.5265, eval loss 0.7739357352256775
tensor([[0.2825],
        [0.3361],
        [0.3181],
        ...,
        [0.2695],
        [0.2579],
        [0.2649]])
Epoch 15 train loss: 0.5120, eval loss 0.7710629105567932
tensor([[0.2758],
        [0.3344],
        [0.3142],
        ...,
        [0.2620],
        [0.2487],
        [0.2573]])
Epoch 16 train loss: 0.5252, eval loss 0.7683786749839783
tensor([[0.2699],
        [0.3335],
        [0.3112],
        ...,
        [0.2552],
        [0.2400],
        [0.2505]])
Epoch 17 train loss: 0.5198, eval loss 0.765913724899292
tensor([[0.2645],
        [0.3332],
        [0.3088],
        ...,
        [0.2489],
        [0.2319],
        [0.2443]])
Epoch 18 train loss: 0.5152, eval loss 0.763627290725708
tensor([[0.2594],
        [0.3334],
        [0.3068],
        ...,
        [0.2430],
        [0.2241],
        [0.2384]])
Epoch 19 train loss: 0.5164, eval loss 0.7614505290985107
tensor([[0.2548],
        [0.3341],
        [0.3053],
        ...,
        [0.2376],
        [0.2167],
        [0.2330]])
Epoch 20 train loss: 0.5058, eval loss 0.7594103813171387
tensor([[0.2505],
        [0.3353],
        [0.3042],
        ...,
        [0.2324],
        [0.2096],
        [0.2278]])
Epoch 21 train loss: 0.5049, eval loss 0.7574554085731506
```

```
tensor([[0.2465],
        [0.3368],
        [0.3036],
        ...,
        [0.2275],
        [0.2028],
        [0.2230]])
Epoch 22 train loss: 0.5002, eval loss 0.7556126713752747
tensor([[0.2431],
        [0.3390],
        [0.3036],
        ...,
        [0.2229],
        [0.1964],
        [0.2186]])
Epoch 23 train loss: 0.5017, eval loss 0.7539048790931702
tensor([[0.2399],
        [0.3415],
        [0.3039],
        ...,
        [0.2185],
        [0.1902],
        [0.2145]])
Epoch 24 train loss: 0.4827, eval loss 0.7522733211517334
tensor([[0.2370],
        [0.3445],
        [0.3046],
        ...,
        [0.2144],
        [0.1844],
        [0.2107]])
Epoch 25 train loss: 0.4882, eval loss 0.7507303357124329
tensor([[0.2343],
        [0.3478],
        [0.3056],
        ...,
        [0.2106],
        [0.1786],
        [0.2071]])
Epoch 26 train loss: 0.4705, eval loss 0.749243438243866
tensor([[0.2320],
        [0.3515],
        [0.3070],
        ...,
        [0.2069],
        [0.1731],
        [0.2038]])
Epoch 27 train loss: 0.4778, eval loss 0.747844934463501
tensor([[0.2296],
        [0.3553],
        [0.3084],
        ...,
        [0.2033],
        [0.1676],
        [0.2004]])
Epoch 28 train loss: 0.4571, eval loss 0.746440589427948
tensor([[0.2275],
        [0.3596],
        [0.3103],
        ...,
        [0.1999],
        [0.1624],
        [0.1974]])
Epoch 29 train loss: 0.4834, eval loss 0.7451349496841431
tensor([[0.2254],
        [0.3640],
```

```
        [0.3122],
        ...,
        [0.1965],
        [0.1571],
        [0.1943]])
Epoch 30 train loss: 0.4696, eval loss 0.7438175678253174
tensor([[0.2234],
        [0.3686],
        [0.3144],
        ...,
        [0.1933],
        [0.1521],
        [0.1914]])
Epoch 31 train loss: 0.4811, eval loss 0.7425572872161865
tensor([[0.2217],
        [0.3734],
        [0.3170],
        ...,
        [0.1903],
        [0.1471],
        [0.1886]])
Epoch 32 train loss: 0.4778, eval loss 0.7413490414619446
tensor([[0.2202],
        [0.3783],
        [0.3198],
        ...,
        [0.1874],
        [0.1424],
        [0.1861]])
Epoch 33 train loss: 0.4653, eval loss 0.740216076374054
tensor([[0.2187],
        [0.3831],
        [0.3229],
        ...,
        [0.1845],
        [0.1377],
        [0.1835]])
Epoch 34 train loss: 0.4707, eval loss 0.7390872240066528
tensor([[0.2172],
        [0.3881],
        [0.3261],
        ...,
        [0.1817],
        [0.1331],
        [0.1810]])
Epoch 35 train loss: 0.4635, eval loss 0.7379762530326843
tensor([[0.2159],
        [0.3931],
        [0.3295],
        ...,
        [0.1791],
        [0.1287],
        [0.1787]])
Epoch 36 train loss: 0.4594, eval loss 0.7369191646575928
tensor([[0.2145],
        [0.3982],
        [0.3329],
        ...,
        [0.1763],
        [0.1243],
        [0.1763]])
Epoch 37 train loss: 0.4635, eval loss 0.7358587980270386
tensor([[0.2134],
        [0.4035],
        [0.3368],
        ...,
```

```
        [0.1737],
        [0.1200],
        [0.1741]])
Epoch 38 train loss: 0.4638, eval loss 0.7348578572273254
tensor([[0.2122],
        [0.4090],
        [0.3407],
        ...,
        [0.1711],
        [0.1158],
        [0.1719]])
Epoch 39 train loss: 0.4622, eval loss 0.733850359916687
tensor([[0.2110],
        [0.4144],
        [0.3447],
        ...,
        [0.1685],
        [0.1117],
        [0.1697]])
Epoch 40 train loss: 0.4578, eval loss 0.7328647971153259
tensor([[0.2099],
        [0.4201],
        [0.3489],
        ...,
        [0.1660],
        [0.1078],
        [0.1676]])
Epoch 41 train loss: 0.4473, eval loss 0.7319169640541077
tensor([[0.2088],
        [0.4257],
        [0.3531],
        ...,
        [0.1635],
        [0.1039],
        [0.1655]])
Epoch 42 train loss: 0.4522, eval loss 0.7309888005256653
tensor([[0.2078],
        [0.4309],
        [0.3576],
        ...,
        [0.1610],
        [0.1001],
        [0.1635]])
Epoch 43 train loss: 0.4379, eval loss 0.7300700545310974
tensor([[0.2071],
        [0.4362],
        [0.3624],
        ...,
        [0.1588],
        [0.0965],
        [0.1617]])
Epoch 44 train loss: 0.4321, eval loss 0.7292235493659973
tensor([[0.2063],
        [0.4412],
        [0.3672],
        ...,
        [0.1565],
        [0.0930],
        [0.1599]])
Epoch 45 train loss: 0.4412, eval loss 0.7283875942230225
tensor([[0.2056],
        [0.4461],
        [0.3720],
        ...,
        [0.1542],
        [0.0896],
```

```
        [0.1581]])
Epoch 46 train loss: 0.4385, eval loss 0.7275686860084534
tensor([[0.2050],
        [0.4510],
        [0.3772],
        ...,
        [0.1521],
        [0.0863],
        [0.1565]])
Epoch 47 train loss: 0.4603, eval loss 0.7267912030220032
tensor([[0.2043],
        [0.4556],
        [0.3823],
        ...,
        [0.1499],
        [0.0831],
        [0.1548]])
Epoch 48 train loss: 0.4381, eval loss 0.7260323762893677
tensor([[0.2034],
        [0.4601],
        [0.3872],
        ...,
        [0.1474],
        [0.0798],
        [0.1528]])
Epoch 49 train loss: 0.4272, eval loss 0.725221574306488
tensor([[0.2027],
        [0.4644],
        [0.3925],
        ...,
        [0.1453],
        [0.0768],
        [0.1512]])
Epoch 50 train loss: 0.4346, eval loss 0.7244939804077148
tensor([[0.2023],
        [0.4683],
        [0.3981],
        ...,
        [0.1432],
        [0.0740],
        [0.1496]])
Epoch 51 train loss: 0.4272, eval loss 0.723808765411377
tensor([[0.2015],
        [0.4720],
        [0.4032],
        ...,
        [0.1409],
        [0.0711],
        [0.1479]])
Epoch 52 train loss: 0.4463, eval loss 0.723092257976532
tensor([[0.2009],
        [0.4759],
        [0.4085],
        ...,
        [0.1388],
        [0.0684],
        [0.1463]])
Epoch 53 train loss: 0.4387, eval loss 0.7224153876304626
tensor([[0.2004],
        [0.4798],
        [0.4138],
        ...,
        [0.1369],
        [0.0658],
        [0.1449]])
Epoch 54 train loss: 0.4296, eval loss 0.72178053855896
```

```
tensor([[0.1997],
        [0.4838],
        [0.4192],
        ...,
        [0.1348],
        [0.0632],
        [0.1433]])
Epoch 55 train loss: 0.4115, eval loss 0.7211340069770813
tensor([[0.1993],
        [0.4876],
        [0.4246],
        ...,
        [0.1329],
        [0.0609],
        [0.1419]])
Epoch 56 train loss: 0.4017, eval loss 0.7205418348312378
tensor([[0.1992],
        [0.4916],
        [0.4304],
        ...,
        [0.1312],
        [0.0587],
        [0.1408]])
Epoch 57 train loss: 0.4290, eval loss 0.7200368046760559
tensor([[0.1988],
        [0.4956],
        [0.4360],
        ...,
        [0.1293],
        [0.0565],
        [0.1395]])
Epoch 58 train loss: 0.4122, eval loss 0.7194926738739014
tensor([[0.1981],
        [0.4996],
        [0.4412],
        ...,
        [0.1272],
        [0.0543],
        [0.1378]])
Epoch 59 train loss: 0.4067, eval loss 0.7188825011253357
tensor([[0.1978],
        [0.5037],
        [0.4468],
        ...,
        [0.1256],
        [0.0523],
        [0.1367]])
Epoch 60 train loss: 0.4077, eval loss 0.7184053659439087
tensor([[0.1972],
        [0.5078],
        [0.4521],
        ...,
        [0.1236],
        [0.0503],
        [0.1353]])
Epoch 61 train loss: 0.4211, eval loss 0.7178740501403809
tensor([[0.1969],
        [0.5118],
        [0.4576],
        ...,
        [0.1219],
        [0.0485],
        [0.1342]])
Epoch 62 train loss: 0.4133, eval loss 0.7174138426780701
tensor([[0.1964],
        [0.5158],
```

```
        [0.4629],
        ...,
        [0.1200],
        [0.0467],
        [0.1328]])
Epoch 63 train loss: 0.4005, eval loss 0.7169052362442017
tensor([[0.1957],
        [0.5198],
        [0.4680],
        ...,
        [0.1179],
        [0.0449],
        [0.1314]])
Epoch 64 train loss: 0.4241, eval loss 0.7163910865783691
tensor([[0.1953],
        [0.5237],
        [0.4732],
        ...,
        [0.1163],
        [0.0433],
        [0.1302]])
Epoch 65 train loss: 0.4154, eval loss 0.7159478664398193
tensor([[0.1948],
        [0.5276],
        [0.4784],
        ...,
        [0.1143],
        [0.0417],
        [0.1288]])
Epoch 66 train loss: 0.4021, eval loss 0.7154698967933655
tensor([[0.1944],
        [0.5315],
        [0.4835],
        ...,
        [0.1127],
        [0.0402],
        [0.1277]])
Epoch 67 train loss: 0.3928, eval loss 0.7150357961654663
tensor([[0.1939],
        [0.5354],
        [0.4886],
        ...,
        [0.1108],
        [0.0387],
        [0.1264]])
Epoch 68 train loss: 0.4045, eval loss 0.7145876884460449
tensor([[0.1935],
        [0.5395],
        [0.4941],
        ...,
        [0.1093],
        [0.0373],
        [0.1253]])
Epoch 69 train loss: 0.3987, eval loss 0.7141931653022766
tensor([[0.1931],
        [0.5434],
        [0.4990],
        ...,
        [0.1076],
        [0.0360],
        [0.1242]])
Epoch 70 train loss: 0.4038, eval loss 0.713791012763977
tensor([[0.1930],
        [0.5474],
        [0.5043],
        ...,
```

```
        [0.1061],
        [0.0348],
        [0.1232]])
Epoch 71 train loss: 0.3910, eval loss 0.713442325592041
tensor([[0.1924],
        [0.5514],
        [0.5091],
        ...,
        [0.1044],
        [0.0336],
        [0.1221]])
Epoch 72 train loss: 0.4171, eval loss 0.7130584120750427
tensor([[0.1924],
        [0.5554],
        [0.5142],
        ...,
        [0.1030],
        [0.0325],
        [0.1213]])
Epoch 73 train loss: 0.3950, eval loss 0.7127339243888855
tensor([[0.1917],
        [0.5593],
        [0.5187],
        ...,
        [0.1013],
        [0.0314],
        [0.1200]])
Epoch 74 train loss: 0.3978, eval loss 0.7123456597328186
tensor([[0.1913],
        [0.5632],
        [0.5235],
        ...,
        [0.0999],
        [0.0303],
        [0.1190]])
Epoch 75 train loss: 0.3877, eval loss 0.7120031118392944
tensor([[0.1910],
        [0.5670],
        [0.5282],
        ...,
        [0.0984],
        [0.0293],
        [0.1180]])
Epoch 76 train loss: 0.4061, eval loss 0.7116842865943909
tensor([[0.1911],
        [0.5707],
        [0.5330],
        ...,
        [0.0972],
        [0.0285],
        [0.1174]])
Epoch 77 train loss: 0.3756, eval loss 0.7114243507385254
tensor([[0.1903],
        [0.5743],
        [0.5369],
        ...,
        [0.0956],
        [0.0275],
        [0.1162]])
Epoch 78 train loss: 0.4072, eval loss 0.711035430431366
tensor([[0.1900],
        [0.5777],
        [0.5412],
        ...,
        [0.0943],
        [0.0267],
```

```
        [0.1153]])
Epoch 79 train loss: 0.3888, eval loss 0.7107555270195007
tensor([[0.1899],
        [0.5815],
        [0.5455],
        ...,
        [0.0930],
        [0.0259],
        [0.1145]])
Epoch 80 train loss: 0.4060, eval loss 0.7104886770248413
tensor([[0.1892],
        [0.5850],
        [0.5493],
        ...,
        [0.0916],
        [0.0251],
        [0.1134]])
Epoch 81 train loss: 0.3794, eval loss 0.7101578712463379
tensor([[0.1888],
        [0.5886],
        [0.5532],
        ...,
        [0.0904],
        [0.0243],
        [0.1126]])
Epoch 82 train loss: 0.3553, eval loss 0.7098970413208008
tensor([[0.1887],
        [0.5923],
        [0.5574],
        ...,
        [0.0892],
        [0.0237],
        [0.1119]])
Epoch 83 train loss: 0.4031, eval loss 0.7096783518791199
tensor([[0.1879],
        [0.5955],
        [0.5608],
        ...,
        [0.0878],
        [0.0229],
        [0.1107]])
Epoch 84 train loss: 0.3860, eval loss 0.7093378901481628
tensor([[0.1875],
        [0.5989],
        [0.5644],
        ...,
        [0.0866],
        [0.0223],
        [0.1099]])
Epoch 85 train loss: 0.4119, eval loss 0.7090747952461243
tensor([[0.1872],
        [0.6021],
        [0.5679],
        ...,
        [0.0856],
        [0.0218],
        [0.1093]])
Epoch 86 train loss: 0.4035, eval loss 0.7088756561279297
tensor([[0.1865],
        [0.6053],
        [0.5712],
        ...,
        [0.0843],
        [0.0212],
        [0.1084]])
Epoch 87 train loss: 0.3893, eval loss 0.7086224555969238
```

```
tensor([[0.1862],
        [0.6086],
        [0.5745],
        ...,
        [0.0833],
        [0.0206],
        [0.1077]])
Epoch 88 train loss: 0.3730, eval loss 0.7084040641784668
tensor([[0.1852],
        [0.6116],
        [0.5775],
        ...,
        [0.0819],
        [0.0200],
        [0.1065]])
Epoch 89 train loss: 0.3817, eval loss 0.7080830931663513
tensor([[0.1841],
        [0.6147],
        [0.5806],
        ...,
        [0.0803],
        [0.0194],
        [0.1052]])
Epoch 90 train loss: 0.3906, eval loss 0.7077388763427734
tensor([[0.1835],
        [0.6178],
        [0.5837],
        ...,
        [0.0792],
        [0.0189],
        [0.1045]])
Epoch 91 train loss: 0.4222, eval loss 0.7075234055519104
tensor([[0.1832],
        [0.6209],
        [0.5871],
        ...,
        [0.0781],
        [0.0185],
        [0.1038]])
Epoch 92 train loss: 0.4139, eval loss 0.7073392868041992
tensor([[0.1828],
        [0.6239],
        [0.5902],
        ...,
        [0.0772],
        [0.0180],
        [0.1031]])
Epoch 93 train loss: 0.3735, eval loss 0.7071530222892761
tensor([[0.1817],
        [0.6266],
        [0.5929],
        ...,
        [0.0759],
        [0.0176],
        [0.1020]])
Epoch 94 train loss: 0.3870, eval loss 0.7068676352500916
tensor([[0.1811],
        [0.6292],
        [0.5956],
        ...,
        [0.0749],
        [0.0171],
        [0.1012]])
Epoch 95 train loss: 0.3920, eval loss 0.7066364884376526
tensor([[0.1801],
        [0.6317],
```

```
        [0.5980],
        ...,
        [0.0738],
        [0.0167],
        [0.1002]])
Epoch 96 train loss: 0.4151, eval loss 0.7063779234886169
tensor([[0.1797],
        [0.6344],
        [0.6009],
        ...,
        [0.0730],
        [0.0164],
        [0.0996]])
Epoch 97 train loss: 0.4005, eval loss 0.7061991691589355
tensor([[0.1791],
        [0.6369],
        [0.6034],
        ...,
        [0.0722],
        [0.0160],
        [0.0989]])
Epoch 98 train loss: 0.3824, eval loss 0.7060139775276184
tensor([[0.1789],
        [0.6395],
        [0.6061],
        ...,
        [0.0715],
        [0.0158],
        [0.0985]])
Epoch 99 train loss: 0.3765, eval loss 0.7058973908424377
tensor([[0.1784],
        [0.6421],
        [0.6087],
        ...,
        [0.0707],
        [0.0155],
        [0.0979]])
Epoch 100 train loss: 0.4046, eval loss 0.7057429552078247
tensor([[0.1776],
        [0.6445],
        [0.6111],
        ...,
        [0.0699],
        [0.0151],
        [0.0972]])
Epoch 101 train loss: 0.3750, eval loss 0.7055399417877197
tensor([[0.1770],
        [0.6469],
        [0.6134],
        ...,
        [0.0691],
        [0.0149],
        [0.0966]])
Epoch 102 train loss: 0.3792, eval loss 0.7053830623626709
tensor([[0.1759],
        [0.6490],
        [0.6154],
        ...,
        [0.0682],
        [0.0145],
        [0.0956]])
Epoch 103 train loss: 0.3800, eval loss 0.7051323652267456
tensor([[0.1750],
        [0.6511],
        [0.6174],
        ...,
```

```
        [0.0673],
        [0.0143],
        [0.0948]])
Epoch 104 train loss: 0.3856, eval loss 0.7049172520637512
tensor([[0.1744],
        [0.6533],
        [0.6197],
        ...,
        [0.0666],
        [0.0140],
        [0.0943]])
Epoch 105 train loss: 0.3808, eval loss 0.7047616243362427
tensor([[0.1738],
        [0.6555],
        [0.6219],
        ...,
        [0.0657],
        [0.0137],
        [0.0936]])
Epoch 106 train loss: 0.3843, eval loss 0.7045823931694031
tensor([[0.1733],
        [0.6574],
        [0.6240],
        ...,
        [0.0650],
        [0.0135],
        [0.0931]])
Epoch 107 train loss: 0.3882, eval loss 0.704451858997345
tensor([[0.1730],
        [0.6597],
        [0.6264],
        ...,
        [0.0645],
        [0.0133],
        [0.0928]])
Epoch 108 train loss: 0.3748, eval loss 0.7043580412864685
tensor([[0.1720],
        [0.6614],
        [0.6280],
        ...,
        [0.0637],
        [0.0131],
        [0.0921]])
Epoch 109 train loss: 0.3928, eval loss 0.7041369080543518
tensor([[0.1715],
        [0.6635],
        [0.6301],
        ...,
        [0.0630],
        [0.0129],
        [0.0916]])
Epoch 110 train loss: 0.3997, eval loss 0.7040091753005981
tensor([[0.1710],
        [0.6656],
        [0.6322],
        ...,
        [0.0624],
        [0.0127],
        [0.0912]])
Epoch 111 train loss: 0.3746, eval loss 0.7038609385490417
tensor([[0.1705],
        [0.6674],
        [0.6341],
        ...,
        [0.0616],
        [0.0125],
```

```
        [0.0906]])
Epoch 112 train loss: 0.3584, eval loss 0.703709602355957
tensor([[0.1700],
        [0.6691],
        [0.6359],
        ...,
        [0.0611],
        [0.0123],
        [0.0902]])
Epoch 113 train loss: 0.3574, eval loss 0.7035880088806152
tensor([[0.1696],
        [0.6708],
        [0.6377],
        ...,
        [0.0605],
        [0.0122],
        [0.0898]])
Epoch 114 train loss: 0.4076, eval loss 0.7034966945648193
tensor([[0.1687],
        [0.6723],
        [0.6392],
        ...,
        [0.0597],
        [0.0120],
        [0.0891]])
Epoch 115 train loss: 0.3948, eval loss 0.7033085823059082
tensor([[0.1680],
        [0.6742],
        [0.6410],
        ...,
        [0.0591],
        [0.0118],
        [0.0887]])
Epoch 116 train loss: 0.3864, eval loss 0.7031906247138977
tensor([[0.1677],
        [0.6757],
        [0.6427],
        ...,
        [0.0587],
        [0.0117],
        [0.0884]])
Epoch 117 train loss: 0.3775, eval loss 0.7031001448631287
tensor([[0.1670],
        [0.6774],
        [0.6445],
        ...,
        [0.0581],
        [0.0115],
        [0.0879]])
Epoch 118 train loss: 0.3760, eval loss 0.7029528021812439
tensor([[0.1665],
        [0.6790],
        [0.6462],
        ...,
        [0.0576],
        [0.0113],
        [0.0875]])
Epoch 119 train loss: 0.3837, eval loss 0.7028353214263916
tensor([[0.1657],
        [0.6804],
        [0.6475],
        ...,
        [0.0571],
        [0.0112],
        [0.0871]])
Epoch 120 train loss: 0.3561, eval loss 0.7027158141136169
```

```
tensor([[0.1653],
        [0.6819],
        [0.6490],
        ...,
        [0.0566],
        [0.0111],
        [0.0868]])
Epoch 121 train loss: 0.3879, eval loss 0.7026017904281616
tensor([[0.1649],
        [0.6834],
        [0.6506],
        ...,
        [0.0562],
        [0.0110],
        [0.0865]])
Epoch 122 train loss: 0.4033, eval loss 0.7024974226951599
tensor([[0.1640],
        [0.6848],
        [0.6519],
        ...,
        [0.0555],
        [0.0108],
        [0.0859]])
Epoch 123 train loss: 0.3899, eval loss 0.7023136615753174
tensor([[0.1639],
        [0.6864],
        [0.6536],
        ...,
        [0.0552],
        [0.0107],
        [0.0859]])
Epoch 124 train loss: 0.3594, eval loss 0.7022779583930969
tensor([[0.1629],
        [0.6877],
        [0.6549],
        ...,
        [0.0546],
        [0.0106],
        [0.0852]])
Epoch 125 train loss: 0.3518, eval loss 0.7020955085754395
tensor([[0.1618],
        [0.6889],
        [0.6560],
        ...,
        [0.0541],
        [0.0104],
        [0.0847]])
Epoch 126 train loss: 0.3869, eval loss 0.7019370794296265
tensor([[0.1615],
        [0.6904],
        [0.6576],
        ...,
        [0.0538],
        [0.0104],
        [0.0845]])
Epoch 127 train loss: 0.3839, eval loss 0.7018577456474304
tensor([[0.1606],
        [0.6914],
        [0.6585],
        ...,
        [0.0532],
        [0.0102],
        [0.0840]])
Epoch 128 train loss: 0.4112, eval loss 0.7016969323158264
tensor([[0.1604],
        [0.6926],
```

```
        [0.6599],
        ...,
        [0.0531],
        [0.0102],
        [0.0840]])
Epoch 129 train loss: 0.3834, eval loss 0.7016733884811401
tensor([[0.1603],
        [0.6939],
        [0.6612],
        ...,
        [0.0529],
        [0.0101],
        [0.0840]])
Epoch 130 train loss: 0.4136, eval loss 0.7016483545303345
tensor([[0.1598],
        [0.6951],
        [0.6625],
        ...,
        [0.0526],
        [0.0100],
        [0.0838]])
Epoch 131 train loss: 0.3912, eval loss 0.7015579342842102
tensor([[0.1586],
        [0.6960],
        [0.6631],
        ...,
        [0.0522],
        [0.0099],
        [0.0833]])
Epoch 132 train loss: 0.3817, eval loss 0.7014084458351135
tensor([[0.1578],
        [0.6974],
        [0.6644],
        ...,
        [0.0518],
        [0.0098],
        [0.0829]])
Epoch 133 train loss: 0.3756, eval loss 0.701295793056488
tensor([[0.1571],
        [0.6985],
        [0.6655],
        ...,
        [0.0515],
        [0.0097],
        [0.0826]])
Epoch 134 train loss: 0.3754, eval loss 0.7011910676956177
tensor([[0.1564],
        [0.6993],
        [0.6662],
        ...,
        [0.0512],
        [0.0097],
        [0.0823]])
Epoch 135 train loss: 0.4205, eval loss 0.701094388961792
tensor([[0.1554],
        [0.7004],
        [0.6672],
        ...,
        [0.0508],
        [0.0096],
        [0.0818]])
Epoch 136 train loss: 0.3884, eval loss 0.7009437680244446
tensor([[0.1553],
        [0.7015],
        [0.6684],
        ...,
```

```
        [0.0505],
        [0.0095],
        [0.0817]])
Epoch 137 train loss: 0.3473, eval loss 0.7008875012397766
tensor([[0.1546],
        [0.7024],
        [0.6693],
        ...,
        [0.0501],
        [0.0094],
        [0.0815]])
Epoch 138 train loss: 0.3795, eval loss 0.7007794976234436
tensor([[0.1539],
        [0.7032],
        [0.6701],
        ...,
        [0.0498],
        [0.0094],
        [0.0812]])
Epoch 139 train loss: 0.3709, eval loss 0.7006811499595642
tensor([[0.1531],
        [0.7043],
        [0.6711],
        ...,
        [0.0495],
        [0.0093],
        [0.0809]])
Epoch 140 train loss: 0.3799, eval loss 0.7005756497383118
tensor([[0.1525],
        [0.7052],
        [0.6721],
        ...,
        [0.0492],
        [0.0092],
        [0.0807]])
Epoch 141 train loss: 0.3937, eval loss 0.7004987597465515
tensor([[0.1516],
        [0.7061],
        [0.6729],
        ...,
        [0.0489],
        [0.0091],
        [0.0803]])
Epoch 142 train loss: 0.3860, eval loss 0.7003626227378845
tensor([[0.1510],
        [0.7072],
        [0.6740],
        ...,
        [0.0486],
        [0.0091],
        [0.0801]])
Epoch 143 train loss: 0.4037, eval loss 0.7002866268157959
tensor([[0.1508],
        [0.7082],
        [0.6752],
        ...,
        [0.0484],
        [0.0090],
        [0.0800]])
Epoch 144 train loss: 0.3691, eval loss 0.7002372741699219
tensor([[0.1499],
        [0.7089],
        [0.6757],
        ...,
        [0.0481],
        [0.0089],
```

```
        [0.0797]])
Epoch 145 train loss: 0.3788, eval loss 0.700129508972168
tensor([[0.1496],
        [0.7097],
        [0.6765],
        ...,
        [0.0479],
        [0.0089],
        [0.0797]])
Epoch 146 train loss: 0.3676, eval loss 0.7000858783721924
tensor([[0.1489],
        [0.7104],
        [0.6773],
        ...,
        [0.0476],
        [0.0088],
        [0.0794]])
Epoch 147 train loss: 0.3982, eval loss 0.6999854445457458
tensor([[0.1480],
        [0.7113],
        [0.6781],
        ...,
        [0.0472],
        [0.0087],
        [0.0790]])
Epoch 148 train loss: 0.3713, eval loss 0.6998491883277893
tensor([[0.1475],
        [0.7122],
        [0.6790],
        ...,
        [0.0471],
        [0.0087],
        [0.0790]])
Epoch 149 train loss: 0.3648, eval loss 0.6998029947280884
tensor([[0.1469],
        [0.7131],
        [0.6798],
        ...,
        [0.0468],
        [0.0086],
        [0.0787]])
Epoch 150 train loss: 0.3967, eval loss 0.6996985673904419
tensor([[0.1465],
        [0.7138],
        [0.6806],
        ...,
        [0.0467],
        [0.0086],
        [0.0788]])
Epoch 151 train loss: 0.4020, eval loss 0.699641227722168
tensor([[0.1462],
        [0.7146],
        [0.6814],
        ...,
        [0.0465],
        [0.0086],
        [0.0788]])
Epoch 152 train loss: 0.3684, eval loss 0.6995991468429565
tensor([[0.1453],
        [0.7154],
        [0.6822],
        ...,
        [0.0461],
        [0.0085],
        [0.0784]])
Epoch 153 train loss: 0.3827, eval loss 0.6994778513908386
```

```
tensor([[0.1450],
        [0.7163],
        [0.6831],
        ...,
        [0.0460],
        [0.0085],
        [0.0784]])
Epoch 154 train loss: 0.3803, eval loss 0.6994487643241882
tensor([[0.1446],
        [0.7173],
        [0.6842],
        ...,
        [0.0458],
        [0.0084],
        [0.0782]])
Epoch 155 train loss: 0.3684, eval loss 0.6994149684906006
tensor([[0.1443],
        [0.7180],
        [0.6849],
        ...,
        [0.0456],
        [0.0084],
        [0.0782]])
Epoch 156 train loss: 0.3890, eval loss 0.6993682384490967
tensor([[0.1438],
        [0.7187],
        [0.6856],
        ...,
        [0.0455],
        [0.0083],
        [0.0781]])
Epoch 157 train loss: 0.4000, eval loss 0.6993017196655273
tensor([[0.1435],
        [0.7195],
        [0.6865],
        ...,
        [0.0453],
        [0.0083],
        [0.0781]])
Epoch 158 train loss: 0.3493, eval loss 0.6992548108100891
tensor([[0.1426],
        [0.7199],
        [0.6867],
        ...,
        [0.0452],
        [0.0083],
        [0.0779]])
Epoch 159 train loss: 0.3639, eval loss 0.6991514563560486
tensor([[0.1419],
        [0.7207],
        [0.6874],
        ...,
        [0.0449],
        [0.0082],
        [0.0777]])
Epoch 160 train loss: 0.3698, eval loss 0.6990823149681091
tensor([[0.1414],
        [0.7213],
        [0.6882],
        ...,
        [0.0446],
        [0.0082],
        [0.0775]])
Epoch 161 train loss: 0.4155, eval loss 0.6990069150924683
tensor([[0.1409],
        [0.7221],
```

```
        [0.6889],
        ...,
        [0.0446],
        [0.0082],
        [0.0775]])
Epoch 162 train loss: 0.3743, eval loss 0.6989786624908447
tensor([[0.1401],
        [0.7225],
        [0.6893],
        ...,
        [0.0444],
        [0.0081],
        [0.0773]])
Epoch 163 train loss: 0.3801, eval loss 0.6988903284072876
tensor([[0.1394],
        [0.7230],
        [0.6896],
        ...,
        [0.0442],
        [0.0081],
        [0.0771]])
Epoch 164 train loss: 0.3946, eval loss 0.6988128423690796
tensor([[0.1390],
        [0.7239],
        [0.6906],
        ...,
        [0.0440],
        [0.0080],
        [0.0770]])
Epoch 165 train loss: 0.3925, eval loss 0.6987646818161011
tensor([[0.1382],
        [0.7243],
        [0.6908],
        ...,
        [0.0438],
        [0.0080],
        [0.0768]])
Epoch 166 train loss: 0.3900, eval loss 0.6986594796180725
tensor([[0.1378],
        [0.7251],
        [0.6917],
        ...,
        [0.0436],
        [0.0080],
        [0.0769]])
Epoch 167 train loss: 0.4008, eval loss 0.6986448764801025
tensor([[0.1366],
        [0.7257],
        [0.6921],
        ...,
        [0.0433],
        [0.0079],
        [0.0764]])
Epoch 168 train loss: 0.3569, eval loss 0.6984814405441284
tensor([[0.1363],
        [0.7263],
        [0.6927],
        ...,
        [0.0432],
        [0.0079],
        [0.0765]])
Epoch 169 train loss: 0.3876, eval loss 0.698462724685669
tensor([[0.1357],
        [0.7267],
        [0.6931],
        ...,
```

```
        [0.0430],
        [0.0078],
        [0.0763]])
Epoch 170 train loss: 0.3715, eval loss 0.698387622833252
tensor([[0.1357],
        [0.7274],
        [0.6939],
        ...,
        [0.0430],
        [0.0078],
        [0.0766]])
Epoch 171 train loss: 0.3869, eval loss 0.6984134912490845
tensor([[0.1348],
        [0.7279],
        [0.6943],
        ...,
        [0.0429],
        [0.0078],
        [0.0764]])
Epoch 172 train loss: 0.3885, eval loss 0.6983288526535034
tensor([[0.1342],
        [0.7285],
        [0.6948],
        ...,
        [0.0426],
        [0.0078],
        [0.0762]])
Epoch 173 train loss: 0.3820, eval loss 0.698250412940979
tensor([[0.1337],
        [0.7291],
        [0.6954],
        ...,
        [0.0425],
        [0.0077],
        [0.0762]])
Epoch 174 train loss: 0.3854, eval loss 0.6982200741767883
tensor([[0.1326],
        [0.7295],
        [0.6955],
        ...,
        [0.0422],
        [0.0077],
        [0.0758]])
Epoch 175 train loss: 0.3401, eval loss 0.6980957984924316
tensor([[0.1323],
        [0.7301],
        [0.6963],
        ...,
        [0.0422],
        [0.0077],
        [0.0759]])
Epoch 176 train loss: 0.3607, eval loss 0.698080837726593
tensor([[0.1322],
        [0.7306],
        [0.6967],
        ...,
        [0.0423],
        [0.0076],
        [0.0761]])
Epoch 177 train loss: 0.3745, eval loss 0.6980745196342468
tensor([[0.1312],
        [0.7310],
        [0.6970],
        ...,
        [0.0421],
        [0.0076],
```

```
        [0.0758]])
Epoch 178 train loss: 0.3678, eval loss 0.697992205619812
tensor([[0.1310],
        [0.7319],
        [0.6978],
        ...,
        [0.0421],
        [0.0076],
        [0.0761]])
Epoch 179 train loss: 0.3766, eval loss 0.6979964375495911
tensor([[0.1304],
        [0.7325],
        [0.6984],
        ...,
        [0.0419],
        [0.0075],
        [0.0759]])
Epoch 180 train loss: 0.3843, eval loss 0.6979105472564697
tensor([[0.1301],
        [0.7332],
        [0.6990],
        ...,
        [0.0419],
        [0.0075],
        [0.0760]])
Epoch 181 train loss: 0.3913, eval loss 0.697867751121521
tensor([[0.1299],
        [0.7338],
        [0.6997],
        ...,
        [0.0419],
        [0.0075],
        [0.0763]])
Epoch 182 train loss: 0.3799, eval loss 0.6979013085365295
tensor([[0.1296],
        [0.7345],
        [0.7005],
        ...,
        [0.0419],
        [0.0075],
        [0.0764]])
Epoch 183 train loss: 0.3897, eval loss 0.6978750824928284
tensor([[0.1289],
        [0.7349],
        [0.7009],
        ...,
        [0.0417],
        [0.0075],
        [0.0762]])
Epoch 184 train loss: 0.3745, eval loss 0.6977971792221069
tensor([[0.1279],
        [0.7352],
        [0.7010],
        ...,
        [0.0415],
        [0.0074],
        [0.0760]])
Epoch 185 train loss: 0.3635, eval loss 0.697679877281189
tensor([[0.1278],
        [0.7356],
        [0.7015],
        ...,
        [0.0415],
        [0.0074],
        [0.0761]])
Epoch 186 train loss: 0.3895, eval loss 0.6976674199104309
```

```
tensor([[0.1273],
        [0.7360],
        [0.7019],
        ...,
        [0.0414],
        [0.0074],
        [0.0761]])
Epoch 187 train loss: 0.3816, eval loss 0.697632372379303
tensor([[0.1265],
        [0.7362],
        [0.7020],
        ...,
        [0.0414],
        [0.0074],
        [0.0760]])
Epoch 188 train loss: 0.3440, eval loss 0.6975563168525696
tensor([[0.1259],
        [0.7368],
        [0.7024],
        ...,
        [0.0413],
        [0.0073],
        [0.0759]])
Epoch 189 train loss: 0.3673, eval loss 0.6975010633468628
tensor([[0.1249],
        [0.7371],
        [0.7026],
        ...,
        [0.0411],
        [0.0073],
        [0.0755]])
Epoch 190 train loss: 0.3852, eval loss 0.6973963379859924
tensor([[0.1246],
        [0.7377],
        [0.7033],
        ...,
        [0.0410],
        [0.0073],
        [0.0755]])
Epoch 191 train loss: 0.3598, eval loss 0.6973908543586731
tensor([[0.1240],
        [0.7382],
        [0.7038],
        ...,
        [0.0409],
        [0.0073],
        [0.0754]])
Epoch 192 train loss: 0.3513, eval loss 0.6973265409469604
tensor([[0.1236],
        [0.7390],
        [0.7045],
        ...,
        [0.0408],
        [0.0072],
        [0.0754]])
Epoch 193 train loss: 0.3814, eval loss 0.6972888708114624
tensor([[0.1230],
        [0.7395],
        [0.7049],
        ...,
        [0.0407],
        [0.0072],
        [0.0754]])
Epoch 194 train loss: 0.3703, eval loss 0.6972500681877136
tensor([[0.1224],
        [0.7400],
```

```
        [0.7053],
        ...,
        [0.0406],
        [0.0072],
        [0.0753]])
Epoch 195 train loss: 0.3846, eval loss 0.6971868872642517
tensor([[0.1217],
        [0.7407],
        [0.7060],
        ...,
        [0.0404],
        [0.0071],
        [0.0750]])
Epoch 196 train loss: 0.3261, eval loss 0.6971120834350586
tensor([[0.1209],
        [0.7408],
        [0.7061],
        ...,
        [0.0402],
        [0.0071],
        [0.0748]])
Epoch 197 train loss: 0.3925, eval loss 0.6970086693763733
tensor([[0.1204],
        [0.7412],
        [0.7064],
        ...,
        [0.0402],
        [0.0071],
        [0.0748]])
Epoch 198 train loss: 0.3487, eval loss 0.6969775557518005
tensor([[0.1197],
        [0.7416],
        [0.7068],
        ...,
        [0.0401],
        [0.0071],
        [0.0747]])
Epoch 199 train loss: 0.4031, eval loss 0.6969247460365295
tensor([[0.1194],
        [0.7420],
        [0.7071],
        ...,
        [0.0400],
        [0.0071],
        [0.0748]])
Epoch 200 train loss: 0.4049, eval loss 0.6968986392021179
tensor([[0.1190],
        [0.7425],
        [0.7075],
        ...,
        [0.0400],
        [0.0070],
        [0.0748]])
Epoch 201 train loss: 0.3562, eval loss 0.6968647837638855
tensor([[0.1184],
        [0.7427],
        [0.7077],
        ...,
        [0.0399],
        [0.0070],
        [0.0746]])
Epoch 202 train loss: 0.3697, eval loss 0.6967989206314087
tensor([[0.1176],
        [0.7429],
        [0.7079],
        ...,
```

```
        [0.0397],
        [0.0070],
        [0.0743]])
Epoch 203 train loss: 0.3811, eval loss 0.6967118978500366
tensor([[0.1170],
        [0.7434],
        [0.7084],
        ...,
        [0.0395],
        [0.0069],
        [0.0741]])
Epoch 204 train loss: 0.3904, eval loss 0.6966419816017151
tensor([[0.1170],
        [0.7442],
        [0.7092],
        ...,
        [0.0396],
        [0.0069],
        [0.0746]])
Epoch 205 train loss: 0.4235, eval loss 0.6967056393623352
tensor([[0.1169],
        [0.7448],
        [0.7098],
        ...,
        [0.0395],
        [0.0069],
        [0.0747]])
Epoch 206 train loss: 0.3586, eval loss 0.6967155933380127
tensor([[0.1165],
        [0.7451],
        [0.7100],
        ...,
        [0.0394],
        [0.0069],
        [0.0746]])
Epoch 207 train loss: 0.3912, eval loss 0.6966533064842224
tensor([[0.1154],
        [0.7452],
        [0.7101],
        ...,
        [0.0391],
        [0.0069],
        [0.0741]])
Epoch 208 train loss: 0.3712, eval loss 0.6965171694755554
tensor([[0.1150],
        [0.7457],
        [0.7105],
        ...,
        [0.0390],
        [0.0068],
        [0.0741]])
Epoch 209 train loss: 0.3647, eval loss 0.6964734792709351
tensor([[0.1145],
        [0.7463],
        [0.7110],
        ...,
        [0.0389],
        [0.0068],
        [0.0741]])
Epoch 210 train loss: 0.3704, eval loss 0.6964406967163086
tensor([[0.1140],
        [0.7468],
        [0.7114],
        ...,
        [0.0387],
        [0.0068],
```

```
        [0.0740]])
Epoch 211 train loss: 0.3663, eval loss 0.6963816285133362
tensor([[0.1132],
        [0.7470],
        [0.7114],
        ...,
        [0.0386],
        [0.0067],
        [0.0738]])
Epoch 212 train loss: 0.3542, eval loss 0.6962978839874268
tensor([[0.1129],
        [0.7474],
        [0.7118],
        ...,
        [0.0386],
        [0.0067],
        [0.0739]])
Epoch 213 train loss: 0.3895, eval loss 0.6962766647338867
tensor([[0.1124],
        [0.7477],
        [0.7120],
        ...,
        [0.0385],
        [0.0067],
        [0.0740]])
Epoch 214 train loss: 0.3750, eval loss 0.6962338089942932
tensor([[0.1119],
        [0.7480],
        [0.7122],
        ...,
        [0.0384],
        [0.0067],
        [0.0740]])
Epoch 215 train loss: 0.3722, eval loss 0.6961844563484192
tensor([[0.1114],
        [0.7484],
        [0.7126],
        ...,
        [0.0382],
        [0.0067],
        [0.0739]])
Epoch 216 train loss: 0.3635, eval loss 0.6961223483085632
tensor([[0.1112],
        [0.7489],
        [0.7133],
        ...,
        [0.0382],
        [0.0067],
        [0.0740]])
Epoch 217 train loss: 0.3489, eval loss 0.6961228847503662
tensor([[0.1108],
        [0.7494],
        [0.7137],
        ...,
        [0.0382],
        [0.0067],
        [0.0742]])
Epoch 218 train loss: 0.3588, eval loss 0.696103036403656
tensor([[0.1102],
        [0.7496],
        [0.7138],
        ...,
        [0.0381],
        [0.0066],
        [0.0740]])
Epoch 219 train loss: 0.3666, eval loss 0.696008563041687
```

```
tensor([[0.1098],
        [0.7503],
        [0.7144],
        ...,
        [0.0379],
        [0.0066],
        [0.0739]])
Epoch 220 train loss: 0.3621, eval loss 0.6959713101387024
tensor([[0.1096],
        [0.7505],
        [0.7147],
        ...,
        [0.0379],
        [0.0066],
        [0.0740]])
Epoch 221 train loss: 0.3931, eval loss 0.6959525942802429
tensor([[0.1088],
        [0.7506],
        [0.7146],
        ...,
        [0.0379],
        [0.0066],
        [0.0738]])
Epoch 222 train loss: 0.3880, eval loss 0.6958574056625366
tensor([[0.1083],
        [0.7510],
        [0.7149],
        ...,
        [0.0377],
        [0.0065],
        [0.0736]])
Epoch 223 train loss: 0.3483, eval loss 0.6957809925079346
tensor([[0.1076],
        [0.7512],
        [0.7150],
        ...,
        [0.0375],
        [0.0065],
        [0.0734]])
Epoch 224 train loss: 0.3384, eval loss 0.695701003074646
tensor([[0.1079],
        [0.7518],
        [0.7157],
        ...,
        [0.0376],
        [0.0065],
        [0.0738]])
Epoch 225 train loss: 0.3767, eval loss 0.6957588791847229
tensor([[0.1075],
        [0.7523],
        [0.7162],
        ...,
        [0.0375],
        [0.0065],
        [0.0738]])
Epoch 226 train loss: 0.3895, eval loss 0.6957255005836487
tensor([[0.1069],
        [0.7526],
        [0.7163],
        ...,
        [0.0374],
        [0.0065],
        [0.0736]])
Epoch 227 train loss: 0.3416, eval loss 0.6956351399421692
tensor([[0.1062],
        [0.7532],
```

```
        [0.7167],
        ...,
        [0.0373],
        [0.0064],
        [0.0735]])
Epoch 228 train loss: 0.3597, eval loss 0.6955839395523071
tensor([[0.1060],
        [0.7538],
        [0.7174],
        ...,
        [0.0374],
        [0.0064],
        [0.0737]])
Epoch 229 train loss: 0.3873, eval loss 0.6956047415733337
tensor([[0.1056],
        [0.7543],
        [0.7179],
        ...,
        [0.0372],
        [0.0064],
        [0.0735]])
Epoch 230 train loss: 0.3743, eval loss 0.6955472826957703
tensor([[0.1053],
        [0.7547],
        [0.7182],
        ...,
        [0.0372],
        [0.0064],
        [0.0736]])
Epoch 231 train loss: 0.3708, eval loss 0.6955159902572632
tensor([[0.1052],
        [0.7550],
        [0.7185],
        ...,
        [0.0372],
        [0.0064],
        [0.0738]])
Epoch 232 train loss: 0.3560, eval loss 0.6955137848854065
tensor([[0.1047],
        [0.7555],
        [0.7189],
        ...,
        [0.0370],
        [0.0064],
        [0.0737]])
Epoch 233 train loss: 0.3724, eval loss 0.6954585909843445
tensor([[0.1045],
        [0.7559],
        [0.7194],
        ...,
        [0.0370],
        [0.0064],
        [0.0737]])
Epoch 234 train loss: 0.3680, eval loss 0.6954541206359863
tensor([[0.1034],
        [0.7559],
        [0.7192],
        ...,
        [0.0368],
        [0.0063],
        [0.0732]])
Epoch 235 train loss: 0.3745, eval loss 0.6952952146530151
tensor([[0.1029],
        [0.7563],
        [0.7195],
        ...,
```

```
        [0.0368],
        [0.0063],
        [0.0732]])
Epoch 236 train loss: 0.3776, eval loss 0.6952669620513916
tensor([[0.1023],
        [0.7565],
        [0.7194],
        ...,
        [0.0366],
        [0.0063],
        [0.0731]])
Epoch 237 train loss: 0.3603, eval loss 0.6951935291290283
tensor([[0.1024],
        [0.7569],
        [0.7200],
        ...,
        [0.0367],
        [0.0063],
        [0.0734]])
Epoch 238 train loss: 0.3628, eval loss 0.6952232718467712
tensor([[0.1021],
        [0.7573],
        [0.7204],
        ...,
        [0.0366],
        [0.0062],
        [0.0734]])
Epoch 239 train loss: 0.3744, eval loss 0.6951811909675598
tensor([[0.1019],
        [0.7580],
        [0.7211],
        ...,
        [0.0366],
        [0.0062],
        [0.0736]])
Epoch 240 train loss: 0.3497, eval loss 0.6952009201049805
tensor([[0.1015],
        [0.7583],
        [0.7213],
        ...,
        [0.0365],
        [0.0062],
        [0.0736]])
Epoch 241 train loss: 0.3876, eval loss 0.6951614618301392
tensor([[0.1008],
        [0.7584],
        [0.7214],
        ...,
        [0.0364],
        [0.0062],
        [0.0734]])
Epoch 242 train loss: 0.3586, eval loss 0.6950662136077881
tensor([[0.1005],
        [0.7590],
        [0.7218],
        ...,
        [0.0364],
        [0.0062],
        [0.0734]])
Epoch 243 train loss: 0.3865, eval loss 0.6950457096099854
tensor([[0.1000],
        [0.7593],
        [0.7219],
        ...,
        [0.0363],
        [0.0061],
```

```
        [0.0733]])
Epoch 244 train loss: 0.3586, eval loss 0.6949782967567444
tensor([[0.0993],
        [0.7593],
        [0.7218],
        ...,
        [0.0361],
        [0.0061],
        [0.0732]])
Epoch 245 train loss: 0.4036, eval loss 0.694895327091217
tensor([[0.0991],
        [0.7598],
        [0.7223],
        ...,
        [0.0362],
        [0.0061],
        [0.0733]])
Epoch 246 train loss: 0.4103, eval loss 0.6948965191841125
tensor([[0.0989],
        [0.7603],
        [0.7227],
        ...,
        [0.0363],
        [0.0061],
        [0.0736]])
Epoch 247 train loss: 0.3650, eval loss 0.6949183940887451
tensor([[0.0984],
        [0.7606],
        [0.7228],
        ...,
        [0.0362],
        [0.0061],
        [0.0736]])
Epoch 248 train loss: 0.3872, eval loss 0.6948816776275635
tensor([[0.0981],
        [0.7611],
        [0.7233],
        ...,
        [0.0361],
        [0.0060],
        [0.0737]])
Epoch 249 train loss: 0.3829, eval loss 0.6948571801185608
tensor([[0.0979],
        [0.7613],
        [0.7235],
        ...,
        [0.0361],
        [0.0060],
        [0.0738]])
Epoch 250 train loss: 0.3517, eval loss 0.694835901260376
tensor([[0.0974],
        [0.7614],
        [0.7235],
        ...,
        [0.0362],
        [0.0060],
        [0.0738]])
Epoch 251 train loss: 0.3525, eval loss 0.6947822570800781
tensor([[0.0967],
        [0.7617],
        [0.7235],
        ...,
        [0.0361],
        [0.0060],
        [0.0736]])
Epoch 252 train loss: 0.3766, eval loss 0.694707989692688
```

```
tensor([[0.0964],
        [0.7622],
        [0.7240],
        ...,
        [0.0360],
        [0.0060],
        [0.0735]])
Epoch 253 train loss: 0.3464, eval loss 0.6946719884872437
tensor([[0.0961],
        [0.7627],
        [0.7246],
        ...,
        [0.0359],
        [0.0059],
        [0.0736]])
Epoch 254 train loss: 0.3510, eval loss 0.6946708559989929
tensor([[0.0958],
        [0.7631],
        [0.7248],
        ...,
        [0.0360],
        [0.0059],
        [0.0736]])
Epoch 255 train loss: 0.3809, eval loss 0.6946452856063843
tensor([[0.0954],
        [0.7635],
        [0.7251],
        ...,
        [0.0358],
        [0.0059],
        [0.0735]])
Epoch 256 train loss: 0.3572, eval loss 0.6945996880531311
tensor([[0.0952],
        [0.7639],
        [0.7255],
        ...,
        [0.0358],
        [0.0059],
        [0.0736]])
Epoch 257 train loss: 0.3867, eval loss 0.694597065448761
tensor([[0.0948],
        [0.7638],
        [0.7253],
        ...,
        [0.0358],
        [0.0059],
        [0.0735]])
Epoch 258 train loss: 0.3850, eval loss 0.6945335268974304
tensor([[0.0943],
        [0.7641],
        [0.7255],
        ...,
        [0.0358],
        [0.0059],
        [0.0735]])
Epoch 259 train loss: 0.3755, eval loss 0.694492518901825
tensor([[0.0939],
        [0.7644],
        [0.7257],
        ...,
        [0.0357],
        [0.0059],
        [0.0734]])
Epoch 260 train loss: 0.3452, eval loss 0.6944348812103271
tensor([[0.0934],
        [0.7646],
```

```
        [0.7258],
        ...,
        [0.0355],
        [0.0058],
        [0.0732]])
Epoch 261 train loss: 0.3700, eval loss 0.6943623423576355
tensor([[0.0931],
        [0.7646],
        [0.7258],
        ...,
        [0.0355],
        [0.0058],
        [0.0732]])
Epoch 262 train loss: 0.3489, eval loss 0.6943272948265076
tensor([[0.0930],
        [0.7651],
        [0.7262],
        ...,
        [0.0355],
        [0.0058],
        [0.0736]])
Epoch 263 train loss: 0.3420, eval loss 0.6943539381027222
tensor([[0.0926],
        [0.7654],
        [0.7265],
        ...,
        [0.0355],
        [0.0058],
        [0.0735]])
Epoch 264 train loss: 0.3614, eval loss 0.6943228840827942
tensor([[0.0923],
        [0.7655],
        [0.7266],
        ...,
        [0.0354],
        [0.0058],
        [0.0735]])
Epoch 265 train loss: 0.3464, eval loss 0.6942926645278931
tensor([[0.0920],
        [0.7658],
        [0.7267],
        ...,
        [0.0353],
        [0.0058],
        [0.0736]])
Epoch 266 train loss: 0.3735, eval loss 0.6942712664604187
tensor([[0.0916],
        [0.7660],
        [0.7269],
        ...,
        [0.0353],
        [0.0058],
        [0.0736]])
Epoch 267 train loss: 0.3592, eval loss 0.6942514181137085
tensor([[0.0914],
        [0.7664],
        [0.7273],
        ...,
        [0.0353],
        [0.0057],
        [0.0737]])
Epoch 268 train loss: 0.3714, eval loss 0.6942546963691711
tensor([[0.0911],
        [0.7665],
        [0.7273],
        ...,
```

```
        [0.0353],
        [0.0057],
        [0.0738]])
Epoch 269 train loss: 0.3844, eval loss 0.6942117214202881
tensor([[0.0906],
        [0.7666],
        [0.7271],
        ...,
        [0.0353],
        [0.0057],
        [0.0737]])
Epoch 270 train loss: 0.3428, eval loss 0.694126307964325
tensor([[0.0903],
        [0.7668],
        [0.7273],
        ...,
        [0.0353],
        [0.0057],
        [0.0738]])
Epoch 271 train loss: 0.3747, eval loss 0.6941080093383789
tensor([[0.0899],
        [0.7670],
        [0.7272],
        ...,
        [0.0352],
        [0.0057],
        [0.0737]])
Epoch 272 train loss: 0.3603, eval loss 0.694038450717926
tensor([[0.0893],
        [0.7673],
        [0.7273],
        ...,
        [0.0350],
        [0.0056],
        [0.0735]])
Epoch 273 train loss: 0.3228, eval loss 0.6939612627029419
tensor([[0.0890],
        [0.7677],
        [0.7277],
        ...,
        [0.0350],
        [0.0056],
        [0.0734]])
Epoch 274 train loss: 0.3518, eval loss 0.6938946843147278
tensor([[0.0887],
        [0.7679],
        [0.7278],
        ...,
        [0.0350],
        [0.0056],
        [0.0735]])
Epoch 275 train loss: 0.3583, eval loss 0.6938655376434326
tensor([[0.0887],
        [0.7681],
        [0.7281],
        ...,
        [0.0351],
        [0.0056],
        [0.0737]])
Epoch 276 train loss: 0.3609, eval loss 0.693889856338501
tensor([[0.0886],
        [0.7684],
        [0.7283],
        ...,
        [0.0351],
        [0.0056],
```

```
        [0.0739]])
Epoch 277 train loss: 0.3886, eval loss 0.6938905119895935
tensor([[0.0886],
        [0.7692],
        [0.7291],
        ...,
        [0.0352],
        [0.0056],
        [0.0741]])
Epoch 278 train loss: 0.3609, eval loss 0.6939295530319214
tensor([[0.0882],
        [0.7691],
        [0.7289],
        ...,
        [0.0351],
        [0.0055],
        [0.0741]])
Epoch 279 train loss: 0.3896, eval loss 0.6938502788543701
tensor([[0.0882],
        [0.7696],
        [0.7295],
        ...,
        [0.0351],
        [0.0055],
        [0.0743]])
Epoch 280 train loss: 0.3941, eval loss 0.6938927173614502
tensor([[0.0878],
        [0.7697],
        [0.7294],
        ...,
        [0.0350],
        [0.0055],
        [0.0742]])
Epoch 281 train loss: 0.3588, eval loss 0.6938362121582031
tensor([[0.0875],
        [0.7700],
        [0.7297],
        ...,
        [0.0350],
        [0.0055],
        [0.0742]])
Epoch 282 train loss: 0.3642, eval loss 0.6937964558601379
tensor([[0.0868],
        [0.7701],
        [0.7295],
        ...,
        [0.0349],
        [0.0055],
        [0.0739]])
Epoch 283 train loss: 0.3283, eval loss 0.6937114000320435
tensor([[0.0866],
        [0.7703],
        [0.7296],
        ...,
        [0.0349],
        [0.0055],
        [0.0740]])
Epoch 284 train loss: 0.3731, eval loss 0.6936900615692139
tensor([[0.0864],
        [0.7707],
        [0.7300],
        ...,
        [0.0349],
        [0.0055],
        [0.0740]])
Epoch 285 train loss: 0.3689, eval loss 0.6936761736869812
```

```
tensor([[0.0863],
        [0.7710],
        [0.7302],
        ...,
        [0.0349],
        [0.0055],
        [0.0741]])
Epoch 286 train loss: 0.3714, eval loss 0.693686306476593
tensor([[0.0859],
        [0.7711],
        [0.7302],
        ...,
        [0.0349],
        [0.0054],
        [0.0741]])
Epoch 287 train loss: 0.3449, eval loss 0.6936401724815369
tensor([[0.0858],
        [0.7715],
        [0.7305],
        ...,
        [0.0350],
        [0.0054],
        [0.0744]])
Epoch 288 train loss: 0.3683, eval loss 0.6936689615249634
tensor([[0.0856],
        [0.7715],
        [0.7304],
        ...,
        [0.0350],
        [0.0054],
        [0.0744]])
Epoch 289 train loss: 0.3482, eval loss 0.6936400532722473
tensor([[0.0853],
        [0.7719],
        [0.7307],
        ...,
        [0.0350],
        [0.0054],
        [0.0743]])
Epoch 290 train loss: 0.3541, eval loss 0.6936181783676147
tensor([[0.0845],
        [0.7722],
        [0.7307],
        ...,
        [0.0348],
        [0.0054],
        [0.0739]])
Epoch 291 train loss: 0.3547, eval loss 0.6934856176376343
tensor([[0.0842],
        [0.7724],
        [0.7307],
        ...,
        [0.0347],
        [0.0054],
        [0.0738]])
Epoch 292 train loss: 0.3483, eval loss 0.6934320330619812
tensor([[0.0839],
        [0.7726],
        [0.7307],
        ...,
        [0.0348],
        [0.0053],
        [0.0739]])
Epoch 293 train loss: 0.3539, eval loss 0.6934288144111633
tensor([[0.0841],
        [0.7732],
```

```
        [0.7315],
        ...,
        [0.0349],
        [0.0053],
        [0.0744]])
Epoch 294 train loss: 0.3692, eval loss 0.6934909224510193
tensor([[0.0841],
        [0.7736],
        [0.7321],
        ...,
        [0.0349],
        [0.0054],
        [0.0746]])
Epoch 295 train loss: 0.3577, eval loss 0.6935422420501709
tensor([[0.0835],
        [0.7737],
        [0.7320],
        ...,
        [0.0348],
        [0.0053],
        [0.0743]])
Epoch 296 train loss: 0.3834, eval loss 0.6934536099433899
tensor([[0.0833],
        [0.7740],
        [0.7323],
        ...,
        [0.0348],
        [0.0053],
        [0.0744]])
Epoch 297 train loss: 0.3402, eval loss 0.6934208273887634
tensor([[0.0831],
        [0.7741],
        [0.7323],
        ...,
        [0.0348],
        [0.0053],
        [0.0745]])
Epoch 298 train loss: 0.3626, eval loss 0.6933882236480713
tensor([[0.0827],
        [0.7743],
        [0.7324],
        ...,
        [0.0346],
        [0.0053],
        [0.0742]])
Epoch 299 train loss: 0.3584, eval loss 0.6933301091194153
```

In [44]:

```
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}%")
```

```
tensor([[0.4202],
        [0.0154],
        [0.0871],
        ...,
        [0.0047],
        [0.5084],
        [0.8567]])
AUROC: 90.23%
F1: 68.53%
Precision: 63.49%
Recall: 74.43%
```

Wyniki wyglądają już dużo lepiej.

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

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

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

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

Zadanie 8 (0.5 punktu)

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

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

Zamień używany optymalizator na `AdamW`.

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

In [45]:
```
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, threshold: float = 0.5):
        y_pred_score = self.predict_proba(x)
        return (y_pred_score > threshold).to(torch.int32)
```

In [46]:
```
from sklearn.utils.class_weight import compute_class_weight

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

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

In [47]:
```
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:
        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"]
        steps_without_improvement = 0
        best_model = deepcopy(model)
        best_threshold = valid_metrics["optimal_threshold"]
    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['l
```

```
tensor([[0.1231],
        [0.8902],
        [0.8421],
        ...,
        [0.1011],
        [0.0100],
        [0.2652]])
Epoch 0 train loss: 0.5201, eval loss 0.8227417469024658
tensor([[0.0798],
        [0.8783],
        [0.8352],
        ...,
        [0.0608],
        [0.0044],
        [0.2085]])
Epoch 1 train loss: 0.5896, eval loss 0.8165199160575867
tensor([[0.0611],
        [0.8733],
        [0.8531],
        ...,
        [0.0539],
        [0.0029],
        [0.2224]])
Epoch 2 train loss: 0.5483, eval loss 0.81540447473526
tensor([[0.0691],
        [0.8761],
        [0.8551],
        ...,
        [0.0484],
        [0.0023],
        [0.2428]])
Epoch 3 train loss: 0.5475, eval loss 0.8143898844718933
tensor([[0.0616],
        [0.8717],
        [0.8552],
        ...,
        [0.0365],
        [0.0025],
        [0.2353]])
Epoch 4 train loss: 0.5595, eval loss 0.8119218349456787
tensor([[0.0507],
        [0.8732],
        [0.8292],
        ...,
        [0.0371],
        [0.0019],
        [0.2376]])
Epoch 5 train loss: 0.4677, eval loss 0.8116204142570496
tensor([[0.0556],
        [0.8835],
        [0.8194],
        ...,
        [0.0382],
        [0.0017],
        [0.2157]])
Epoch 6 train loss: 0.5563, eval loss 0.8120542764663696
tensor([[0.0618],
        [0.8857],
        [0.8174],
        ...,
        [0.0425],
        [0.0019],
        [0.2386]])
Epoch 7 train loss: 0.5043, eval loss 0.8109076023101807
tensor([[0.0467],
```

```
        [0.8803],
        [0.8235],
        ...,
        [0.0415],
        [0.0016],
        [0.2678]])
Epoch 8 train loss: 0.4945, eval loss 0.8109753727912903
tensor([[0.0580],
        [0.8879],
        [0.8177],
        ...,
        [0.0321],
        [0.0011],
        [0.2436]])
Epoch 9 train loss: 0.5311, eval loss 0.8108074069023132
tensor([[6.8400e-02],
        [8.8817e-01],
        [8.3138e-01],
        ...,
        [3.3434e-02],
        [5.1629e-04],
        [2.7240e-01]])
Epoch 10 train loss: 0.5584, eval loss 0.810680627822876
tensor([[5.1143e-02],
        [8.8917e-01],
        [8.2164e-01],
        ...,
        [2.8838e-02],
        [8.3705e-04],
        [2.3823e-01]])
Epoch 11 train loss: 0.4964, eval loss 0.8105016946792603
tensor([[4.7653e-02],
        [8.8125e-01],
        [8.1814e-01],
        ...,
        [2.7056e-02],
        [6.8931e-04],
        [2.2095e-01]])
Epoch 12 train loss: 0.5161, eval loss 0.8096075057983398
tensor([[5.2995e-02],
        [8.8936e-01],
        [8.1724e-01],
        ...,
        [3.5555e-02],
        [5.5341e-04],
        [2.5422e-01]])
Epoch 13 train loss: 0.4856, eval loss 0.8086854219436646
tensor([[5.4828e-02],
        [8.8327e-01],
        [8.1441e-01],
        ...,
        [3.6252e-02],
        [7.7365e-04],
        [3.0478e-01]])
Epoch 14 train loss: 0.5021, eval loss 0.8098427653312683
tensor([[5.4128e-02],
        [8.8811e-01],
        [8.1387e-01],
        ...,
        [3.1692e-02],
        [8.1109e-04],
        [2.7414e-01]])
Epoch 15 train loss: 0.5009, eval loss 0.8091221451759338
tensor([[4.3599e-02],
        [8.8134e-01],
        [8.2901e-01],
```

```
        ...,
        [2.4832e-02],
        [4.1398e-04],
        [2.6567e-01]])
Epoch 16 train loss: 0.5263, eval loss 0.8093746900558472
tensor([[4.2367e-02],
        [8.9041e-01],
        [8.1626e-01],
        ...,
        [1.4587e-02],
        [4.2889e-04],
        [2.8596e-01]])
```

In [48]:
```
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}%")
```

```
tensor([[0.5535],
        [0.0026],
        [0.1730],
        ...,
        [0.0043],
        [0.7110],
        [0.9636]])
AUROC: 90.73%
F1: 69.79%
Precision: 65.21%
Recall: 75.06%
```

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 [49]:
```
import time


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

        self.mlp = nn.Sequential(
            nn.Linear(input_size, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.Linear(256, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(dropout_p),
```

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

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

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

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


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

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

        step_counter += 1

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

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

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[49], line 39
     35         y_pred_score = self.predict_proba(x)
     36         return (y_pred_score > threshold).to(torch.int32)
---> 39 model = CudaMLP(X_train.shape[1]).to('cuda')
     41 optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate, weight_decay
=1e-4)
     43 # note that we are using loss function with sigmoid built in

File G:\anaconda3\Lib\site-packages\torch\nn\modules\module.py:1160, in Module.to(self,
 *args, **kwargs)
   1156         return t.to(device, dtype if t.is_floating_point() or t.is_complex() els
e 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 No
ne, non_blocking)
-> 1160 return self._apply(convert)

File G:\anaconda3\Lib\site-packages\torch\nn\modules\module.py:810, in Module._apply(sel
f, fn, recurse)
    808 if recurse:
    809     for module in self.children():
--> 810         module._apply(fn)
    812 def compute_should_use_set_data(tensor, tensor_applied):
    813     if torch._has_compatible_shallow_copy_type(tensor, tensor_applied):
    814         # If the new tensor has compatible tensor type as the existing tensor,
    815         # the current behavior is to change the tensor in-place using `.data =`,
   (...)
    820         # global flag to let the user control whether they want the future
    821         # behavior of overwriting the existing tensor or not.

File G:\anaconda3\Lib\site-packages\torch\nn\modules\module.py:810, in Module._apply(sel
f, fn, recurse)
    808 if recurse:
    809     for module in self.children():
--> 810         module._apply(fn)
    812 def compute_should_use_set_data(tensor, tensor_applied):
    813     if torch._has_compatible_shallow_copy_type(tensor, tensor_applied):
    814         # If the new tensor has compatible tensor type as the existing tensor,
    815         # the current behavior is to change the tensor in-place using `.data =`,
   (...)
    820         # global flag to let the user control whether they want the future
    821         # behavior of overwriting the existing tensor or not.

File G:\anaconda3\Lib\site-packages\torch\nn\modules\module.py:833, in Module._apply(sel
f, fn, recurse)
    829 # Tensors stored in modules are graph leaves, and we don't want to
    830 # track autograd history of `param_applied`, so we have to use
    831 # `with torch.no_grad():`
    832 with torch.no_grad():
--> 833     param_applied = fn(param)
    834 should_use_set_data = compute_should_use_set_data(param, param_applied)
    835 if should_use_set_data:

File G:\anaconda3\Lib\site-packages\torch\nn\modules\module.py:1158, in Module.to.<local
s>.convert(t)
   1155 if convert_to_format is not None and t.dim() in (4, 5):
   1156     return t.to(device, dtype if t.is_floating_point() or t.is_complex() else No
ne,
   1157                 non_blocking, memory_format=convert_to_format)
-> 1158 return t.to(device, dtype if t.is_floating_point() or t.is_complex() else None,
 non_blocking)

File G:\anaconda3\Lib\site-packages\torch\cuda\__init__.py:289, in _lazy_init()
    284     raise RuntimeError(
    285         "Cannot re-initialize CUDA in forked subprocess. To use CUDA with "
    286         "multiprocessing, you must use the 'spawn' start method"
    287     )
    288 if not hasattr(torch._C, "_cuda_getDeviceCount"):
--> 289     raise AssertionError("Torch not compiled with CUDA enabled")
    290 if _cudart is None:
    291     raise AssertionError(
    292         "libcudart functions unavailable. It looks like you have a broken build?
"
    293     )

AssertionError: Torch not compiled with CUDA enabled
```

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

tym szybszy.

Dla zainteresowanych polecamy tę serie artykułów

Zadanie dla chętnych

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

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

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

Przeszukaj wybrane zakresy dla hiperparametrów:

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

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

Przydatne materiały:

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

In [ ]: