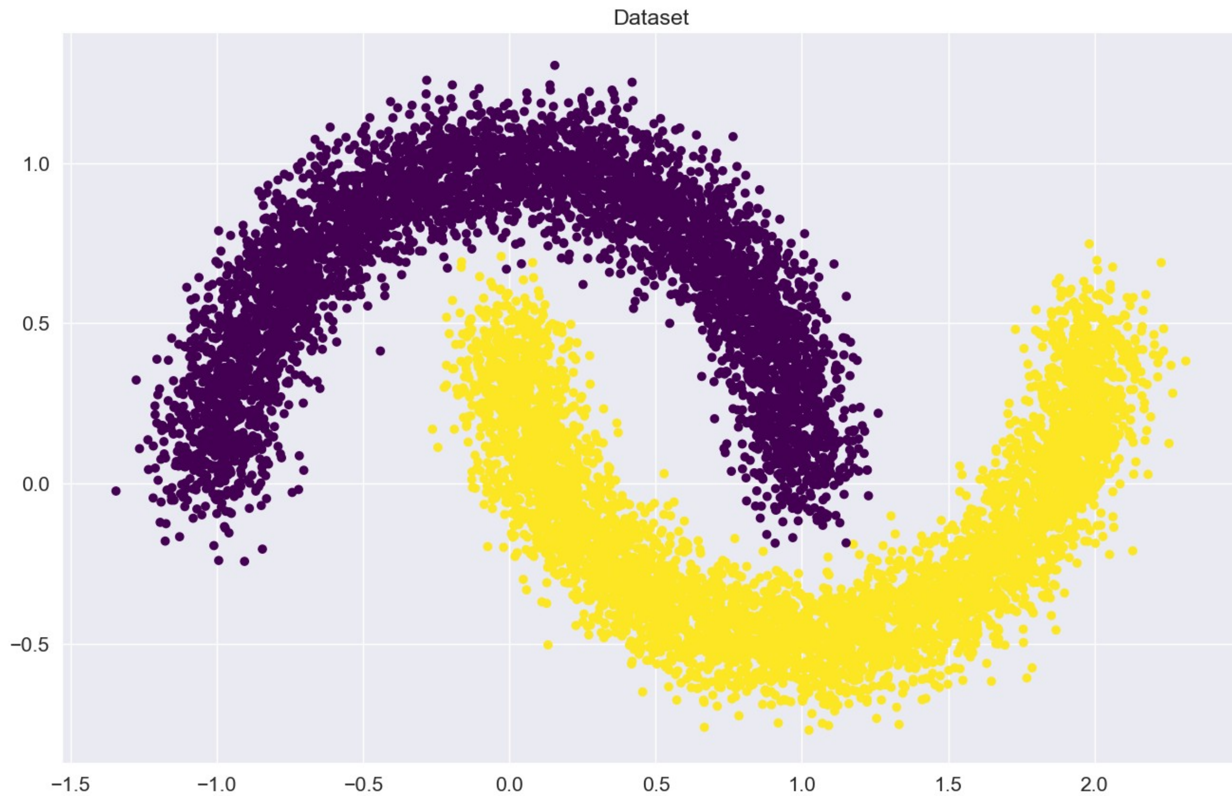
В этом занятии вам предстоит потренироваться построению нейронных сетей с помощью библиотеки Pytorch. Делать мы это будем на нескольких датасетах.

```
import numpy as np

import seaborn as sns
from matplotlib import pyplot as plt

from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split

import torch
from torch import nn
from torch.nn import functional as F

from torch.utils.data import TensorDataset, DataLoader

sns.set(style="darkgrid", font_scale=1.4)
```

Часть 1. Датасет moons

Давайте сгенерируем датасет и посмотрим на него!

```
X, y = make_moons(n_samples=10000, random_state=42, noise=0.1)

plt.figure(figsize=(16, 10))
plt.title("Dataset")
plt.scatter(X[:, 0], X[:, 1], c=y, cmap="viridis")
plt.show()
```



Сделаем train/test split

Загрузка данных

В PyTorch загрузка данных как правило происходит налету (иногда датасеты не помещаются в оперативную память). Для этого используются две сущности `Dataset` и `DataLoader`.

1. `Dataset` загружает каждый объект по отдельности.
2. `DataLoader` группирует объекты из `Dataset` в батчи.

Так как наш датасет достаточно маленький мы будем использовать `TensorDataset`. Все, что нам нужно, это перевести из массива numpy в тензор с типом `torch.float32`.

Задание. Создайте тензоры с обучающими и тестовыми данными

Создаем `Dataset` и `DataLoader`.

```
X_train, X_val, y_train, y_val = train_test_split(X, y,
random_state=42, test_size=0.2)

X_train_t = torch.tensor(X_train, dtype=torch.float32)
y_train_t = torch.tensor(y_train, dtype=torch.float32)
X_val_t = torch.tensor(X_val, dtype=torch.float32)
y_val_t = torch.tensor(y_val, dtype=torch.float32)
```

```
train_dataset = TensorDataset(X_train_t, y_train_t)
val_dataset = TensorDataset(X_val_t, y_val_t)
train_dataloader = DataLoader(train_dataset, batch_size=128)
val_dataloader = DataLoader(val_dataset, batch_size=128)
print(len(train_dataloader), len(val_dataloader))
```

```
63 16
```

Logistic regression is my profession

Напоминание Давайте вспоним, что происходит в логистической регрессии. На входе у нас есть матрица объект-признак X и столбец-вектор y – метки из $\{0,1\}$ для каждого объекта. Мы хотим найти такую матрицу весов W и смещение b (bias), что наша модель $XW+b$ будет каким-то образом предсказывать класс объекта. Как видно на выходе наша модель может выдавать число в интервале от $(-\infty;\infty)$. Этот выход как правило называют "логитами" (logits). Нам необходимо перевести его на интервал от $[0;1)$ для того, чтобы он выдавал нам вероятность принадлежности объекта к кассу один, также лучше, чтобы эта функция была монотонной, быстро считалась, имела производную и на $-\infty$ имела значение 0, а на $+\infty$ имела значение 1. Такой класс функций называется сигмоидыю. Чаще всего в качестве сигмоида берут

$$\sigma(x)=\frac{1}{1+e^{-x}}.$$

Задание. Реализация логистической регрессии

Вам необходимо написать модуль на PyTorch реализующий $logits=XW+b$, где W и b – параметры (`nn.Parameter`) модели. Иначе говоря, здесь мы реализуем своими руками модуль `nn.Linear` (в этом пункте его использование запрещено). Инициализируйте веса нормальным распределением (`torch.randn`).

```
class LinearRegression(nn.Module):
    def __init__(self, in_features: int, out_features: int, bias: bool
= True):
        super().__init__()
        self.weights =
nn.Parameter(torch.randn(in_features,out_features))
        self.bias = bias
        if bias:
            self.bias_term = nn.Parameter(torch.rand(out_features))

    def forward(self, x):
        x = x @ self.weights
        if self.bias:
            x += self.bias_term
        return x
```

Вопрос 1. Сколько обучаемых параметров у получившейся модели? Имеется в виду суммарное количество отдельных числовых переменных, а не количество тензоров.

Поскольку сеть полносвязная то будем перемножать количество нейрнов в слоях между собой, таким образов именно "классических" весов будет in_features * out_features. Если у нас еще и bias'ы используются то сюда прибавим еще раз количество out_feautures. Итого: out_features (in_feautures + 1) обучаемых параметров.

Train loop

Вот псевдокод, который поможет вам разобраться в том, что происходит во время обучения

```
for epoch in range(max_epochs):  # <----------- итерируемся по датасету несколько раз
    for x_batch, y_batch in dataset:  # <------ итерируемся по датасету. Так как мы используем SGD а не GD, то берем батчи заданного размера
        optimizer.zero_grad()  # <------------- обуляем градиенты модели
        outp = model(x_batch)  # <------------- получаем "логиты" из модели
        loss = loss_func(outp, y_batch)  # <--- считаем "лосс" для логистической регрессии
        loss.backward()  # <------------------- считаем градиенты
        optimizer.step()  # <------------------ делаем шаг градиентного спуска
        if convergence:  # <------------------- в случае сходимости выходим из цикла
            break
```

В коде ниже добавлено логирование `accuracy` и `loss`.

Задание. Реализация цикла обучения

```
from sklearn.metrics import accuracy_score


linear_regression = LinearRegression(2, 1)
loss_function = nn.BCEWithLogitsLoss()
optimizer = torch.optim.SGD(linear_regression.parameters(), lr=0.01,
weight_decay=1e-5)


train_losses = []
train_accuracies = []
val_losses = []
val_accuracies = []
max_epochs = 100
```

```
best_val_loss = float('inf')
patience = 10
loss_tol = 0.001
patience_counter = 0
best_weights = None

for epoch in range(max_epochs):
    epoch_train_loss = 0
    epoch_train_preds = []
    epoch_train_reals = []
    for X_batch, y_batch in train_dataloader:
        outp = linear_regression(X_batch).squeeze()
        loss = loss_function(outp, y_batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        epoch_train_loss += loss.item()

        probabilities = torch.sigmoid(outp)
        preds = (probabilities > 0.5).type(torch.long)
        epoch_train_preds.extend(preds.cpu().numpy())
        epoch_train_reals.extend(y_batch.cpu().numpy())

    epoch_train_loss /= len(train_dataloader)
    train_losses.append(epoch_train_loss)
    epoch_train_accuracy = accuracy_score(epoch_train_reals,
epoch_train_preds)
    train_accuracies.append(epoch_train_accuracy)


    epoch_val_loss = 0
    epoch_val_preds = []
    epoch_val_reals = []
    with torch.no_grad():
        for X_val_batch, y_val_batch in val_dataloader:
            val_output = linear_regression(X_val_batch).squeeze()
            epoch_val_loss += loss_function(val_output,
y_val_batch).item()

            probabilities = torch.sigmoid(outp)
            val_preds = (probabilities > 0.5).type(torch.long)
            epoch_val_preds.extend(val_preds.cpu().numpy())
            epoch_val_reals.extend(y_batch.cpu().numpy())

    epoch_val_loss /= len(val_dataloader)
    val_losses.append(epoch_val_loss)
    epoch_val_accuracy = accuracy_score(epoch_val_reals,
epoch_val_preds)
    val_accuracies.append(epoch_val_accuracy)
```

```
    print(f"Epoch: {epoch}")
    print(f"train/val acc: {train_accuracies[-1]:.4f} /
{val_accuracies[-1]:.4f}")
    print(f"train/val loss: {train_losses[-1]:.5f} / {val_losses[-
1]:.5f}")

    if epoch_val_loss < best_val_loss + loss_tol:
        best_val_loss = epoch_val_loss
        patience_counter = 0
        best_weights = linear_regression.state_dict()
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping triggered. Restoring best weights.")
            linear_regression.load_state_dict(best_weights)  #
Восстанавливаем лучшие веса
            break
```

```
Epoch: 0
train/val acc: 0.1614 / 0.1562
train/val loss: 1.51740 / 1.46872
Epoch: 1
train/val acc: 0.1826 / 0.2188
train/val loss: 1.37045 / 1.32528
Epoch: 2
train/val acc: 0.2009 / 0.2344
train/val loss: 1.24033 / 1.19861
Epoch: 3
train/val acc: 0.2260 / 0.2812
train/val loss: 1.12608 / 1.08770
Epoch: 4
train/val acc: 0.2645 / 0.2969
train/val loss: 1.02663 / 0.99144
Epoch: 5
train/val acc: 0.3078 / 0.2656
train/val loss: 0.94079 / 0.90853
Epoch: 6
train/val acc: 0.3500 / 0.2969
train/val loss: 0.86722 / 0.83753
Epoch: 7
train/val acc: 0.4019 / 0.3594
train/val loss: 0.80446 / 0.77696
Epoch: 8
train/val acc: 0.4447 / 0.4062
train/val loss: 0.75107 / 0.72534
Epoch: 9
train/val acc: 0.4755 / 0.3750
train/val loss: 0.70563 / 0.68128
Epoch: 10
```

```
train/val acc: 0.5089 / 0.3906
train/val loss: 0.66688 / 0.64356
Epoch: 11
train/val acc: 0.5403 / 0.4062
train/val loss: 0.63370 / 0.61111
Epoch: 12
train/val acc: 0.5723 / 0.4688
train/val loss: 0.60514 / 0.58304
Epoch: 13
train/val acc: 0.6058 / 0.5312
train/val loss: 0.58042 / 0.55861
Epoch: 14
train/val acc: 0.6332 / 0.5625
train/val loss: 0.55888 / 0.53722
Epoch: 15
train/val acc: 0.6558 / 0.5938
train/val loss: 0.53998 / 0.51835
Epoch: 16
train/val acc: 0.6739 / 0.6094
train/val loss: 0.52330 / 0.50162
Epoch: 17
train/val acc: 0.6887 / 0.6250
train/val loss: 0.50848 / 0.48668
Epoch: 18
train/val acc: 0.7045 / 0.6406
train/val loss: 0.49524 / 0.47328
Epoch: 19
train/val acc: 0.7147 / 0.6406
train/val loss: 0.48334 / 0.46118
Epoch: 20
train/val acc: 0.7244 / 0.6406
train/val loss: 0.47259 / 0.45021
Epoch: 21
train/val acc: 0.7322 / 0.6562
train/val loss: 0.46282 / 0.44020
Epoch: 22
train/val acc: 0.7406 / 0.6875
train/val loss: 0.45391 / 0.43105
Epoch: 23
train/val acc: 0.7464 / 0.6875
train/val loss: 0.44574 / 0.42263
Epoch: 24
train/val acc: 0.7522 / 0.7031
train/val loss: 0.43822 / 0.41487
Epoch: 25
train/val acc: 0.7569 / 0.7344
train/val loss: 0.43128 / 0.40768
Epoch: 26
train/val acc: 0.7620 / 0.7344
```

```
train/val loss: 0.42484 / 0.40099
Epoch: 27
train/val acc: 0.7684 / 0.7344
train/val loss: 0.41885 / 0.39477
Epoch: 28
train/val acc: 0.7722 / 0.7344
train/val loss: 0.41326 / 0.38895
Epoch: 29
train/val acc: 0.7762 / 0.7344
train/val loss: 0.40803 / 0.38349
Epoch: 30
train/val acc: 0.7791 / 0.7500
train/val loss: 0.40312 / 0.37837
Epoch: 31
train/val acc: 0.7833 / 0.7500
train/val loss: 0.39851 / 0.37354
Epoch: 32
train/val acc: 0.7871 / 0.7656
train/val loss: 0.39415 / 0.36898
Epoch: 33
train/val acc: 0.7890 / 0.7656
train/val loss: 0.39004 / 0.36468
Epoch: 34
train/val acc: 0.7915 / 0.7656
train/val loss: 0.38615 / 0.36060
Epoch: 35
train/val acc: 0.7939 / 0.7656
train/val loss: 0.38245 / 0.35672
Epoch: 36
train/val acc: 0.7975 / 0.7656
train/val loss: 0.37894 / 0.35304
Epoch: 37
train/val acc: 0.7994 / 0.7812
train/val loss: 0.37560 / 0.34953
Epoch: 38
train/val acc: 0.8009 / 0.7812
train/val loss: 0.37241 / 0.34619
Epoch: 39
train/val acc: 0.8029 / 0.7812
train/val loss: 0.36937 / 0.34299
Epoch: 40
train/val acc: 0.8056 / 0.7812
train/val loss: 0.36646 / 0.33994
Epoch: 41
train/val acc: 0.8077 / 0.7969
train/val loss: 0.36368 / 0.33701
Epoch: 42
train/val acc: 0.8090 / 0.7969
train/val loss: 0.36101 / 0.33421
```

```
Epoch: 43
train/val acc: 0.8115 / 0.8125
train/val loss: 0.35845 / 0.33152
Epoch: 44
train/val acc: 0.8124 / 0.8125
train/val loss: 0.35599 / 0.32894
Epoch: 45
train/val acc: 0.8141 / 0.8438
train/val loss: 0.35362 / 0.32646
Epoch: 46
train/val acc: 0.8154 / 0.8438
train/val loss: 0.35135 / 0.32407
Epoch: 47
train/val acc: 0.8167 / 0.8438
train/val loss: 0.34915 / 0.32177
Epoch: 48
train/val acc: 0.8177 / 0.8438
train/val loss: 0.34704 / 0.31955
Epoch: 49
train/val acc: 0.8196 / 0.8438
train/val loss: 0.34500 / 0.31741
Epoch: 50
train/val acc: 0.8219 / 0.8438
train/val loss: 0.34303 / 0.31535
Epoch: 51
train/val acc: 0.8235 / 0.8594
train/val loss: 0.34113 / 0.31336
Epoch: 52
train/val acc: 0.8245 / 0.8594
train/val loss: 0.33928 / 0.31143
Epoch: 53
train/val acc: 0.8260 / 0.8750
train/val loss: 0.33750 / 0.30956
Epoch: 54
train/val acc: 0.8277 / 0.8750
train/val loss: 0.33578 / 0.30776
Epoch: 55
train/val acc: 0.8291 / 0.8750
train/val loss: 0.33411 / 0.30601
Epoch: 56
train/val acc: 0.8304 / 0.8906
train/val loss: 0.33249 / 0.30432
Epoch: 57
train/val acc: 0.8314 / 0.8906
train/val loss: 0.33092 / 0.30268
Epoch: 58
train/val acc: 0.8329 / 0.8906
train/val loss: 0.32939 / 0.30109
Epoch: 59
```

```
train/val acc: 0.8336 / 0.8906
train/val loss: 0.32791 / 0.29955
Epoch: 60
train/val acc: 0.8345 / 0.8906
train/val loss: 0.32648 / 0.29805
Epoch: 61
train/val acc: 0.8353 / 0.8906
train/val loss: 0.32508 / 0.29659
Epoch: 62
train/val acc: 0.8355 / 0.8906
train/val loss: 0.32372 / 0.29518
Epoch: 63
train/val acc: 0.8363 / 0.8906
train/val loss: 0.32240 / 0.29380
Epoch: 64
train/val acc: 0.8371 / 0.8906
train/val loss: 0.32112 / 0.29247
Epoch: 65
train/val acc: 0.8381 / 0.8906
train/val loss: 0.31987 / 0.29117
Epoch: 66
train/val acc: 0.8387 / 0.8906
train/val loss: 0.31866 / 0.28991
Epoch: 67
train/val acc: 0.8396 / 0.8906
train/val loss: 0.31747 / 0.28868
Epoch: 68
train/val acc: 0.8404 / 0.8906
train/val loss: 0.31632 / 0.28748
Epoch: 69
train/val acc: 0.8414 / 0.8906
train/val loss: 0.31520 / 0.28631
Epoch: 70
train/val acc: 0.8424 / 0.8906
train/val loss: 0.31410 / 0.28517
Epoch: 71
train/val acc: 0.8436 / 0.8906
train/val loss: 0.31304 / 0.28407
Epoch: 72
train/val acc: 0.8445 / 0.8906
train/val loss: 0.31199 / 0.28299
Epoch: 73
train/val acc: 0.8451 / 0.8906
train/val loss: 0.31098 / 0.28193
Epoch: 74
train/val acc: 0.8456 / 0.8906
train/val loss: 0.30999 / 0.28091
Epoch: 75
train/val acc: 0.8458 / 0.8906
```

```
train/val loss: 0.30902 / 0.27991
Epoch: 76
train/val acc: 0.8466 / 0.8906
train/val loss: 0.30808 / 0.27893
Epoch: 77
train/val acc: 0.8470 / 0.8906
train/val loss: 0.30716 / 0.27798
Epoch: 78
train/val acc: 0.8475 / 0.8906
train/val loss: 0.30626 / 0.27705
Epoch: 79
train/val acc: 0.8478 / 0.8906
train/val loss: 0.30538 / 0.27614
Epoch: 80
train/val acc: 0.8486 / 0.8906
train/val loss: 0.30452 / 0.27525
Epoch: 81
train/val acc: 0.8495 / 0.8906
train/val loss: 0.30369 / 0.27438
Epoch: 82
train/val acc: 0.8501 / 0.8906
train/val loss: 0.30287 / 0.27353
Epoch: 83
train/val acc: 0.8505 / 0.8906
train/val loss: 0.30207 / 0.27271
Epoch: 84
train/val acc: 0.8506 / 0.9062
train/val loss: 0.30128 / 0.27190
Epoch: 85
train/val acc: 0.8511 / 0.9062
train/val loss: 0.30052 / 0.27111
Epoch: 86
train/val acc: 0.8521 / 0.9062
train/val loss: 0.29977 / 0.27033
Epoch: 87
train/val acc: 0.8524 / 0.9062
train/val loss: 0.29904 / 0.26958
Epoch: 88
train/val acc: 0.8531 / 0.9062
train/val loss: 0.29832 / 0.26884
Epoch: 89
train/val acc: 0.8536 / 0.9062
train/val loss: 0.29762 / 0.26811
Epoch: 90
train/val acc: 0.8539 / 0.9062
train/val loss: 0.29694 / 0.26740
Epoch: 91
train/val acc: 0.8544 / 0.9062
train/val loss: 0.29626 / 0.26671
```

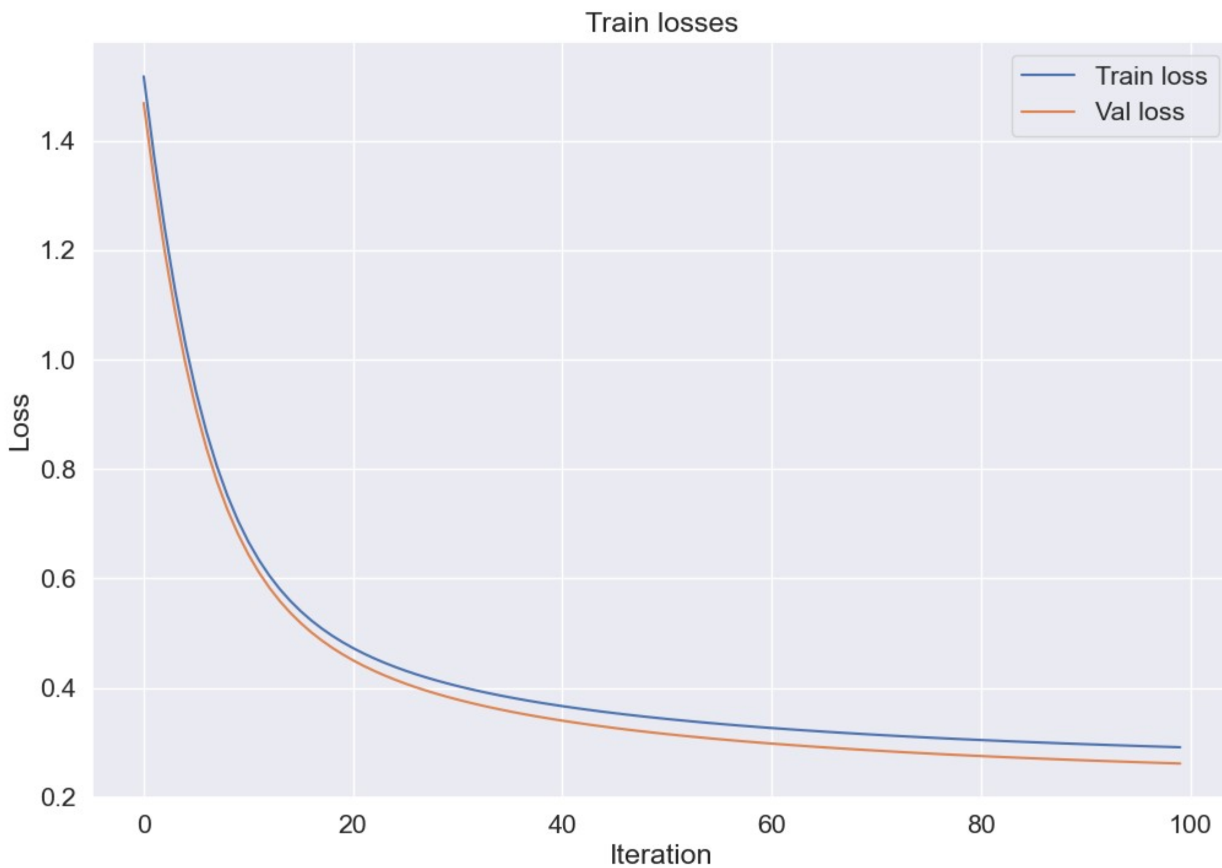
```
Epoch: 92
train/val acc: 0.8549 / 0.9062
train/val loss: 0.29561 / 0.26603
Epoch: 93
train/val acc: 0.8556 / 0.9062
train/val loss: 0.29497 / 0.26537
Epoch: 94
train/val acc: 0.8559 / 0.9062
train/val loss: 0.29434 / 0.26472
Epoch: 95
train/val acc: 0.8562 / 0.9062
train/val loss: 0.29372 / 0.26408
Epoch: 96
train/val acc: 0.8565 / 0.9062
train/val loss: 0.29311 / 0.26345
Epoch: 97
train/val acc: 0.8566 / 0.9062
train/val loss: 0.29252 / 0.26284
Epoch: 98
train/val acc: 0.8570 / 0.9062
train/val loss: 0.29194 / 0.26224
Epoch: 99
train/val acc: 0.8578 / 0.9062
train/val loss: 0.29138 / 0.26166
```

Вопрос 2. Сколько итераций потребовалось, чтобы алгоритм сошелся?

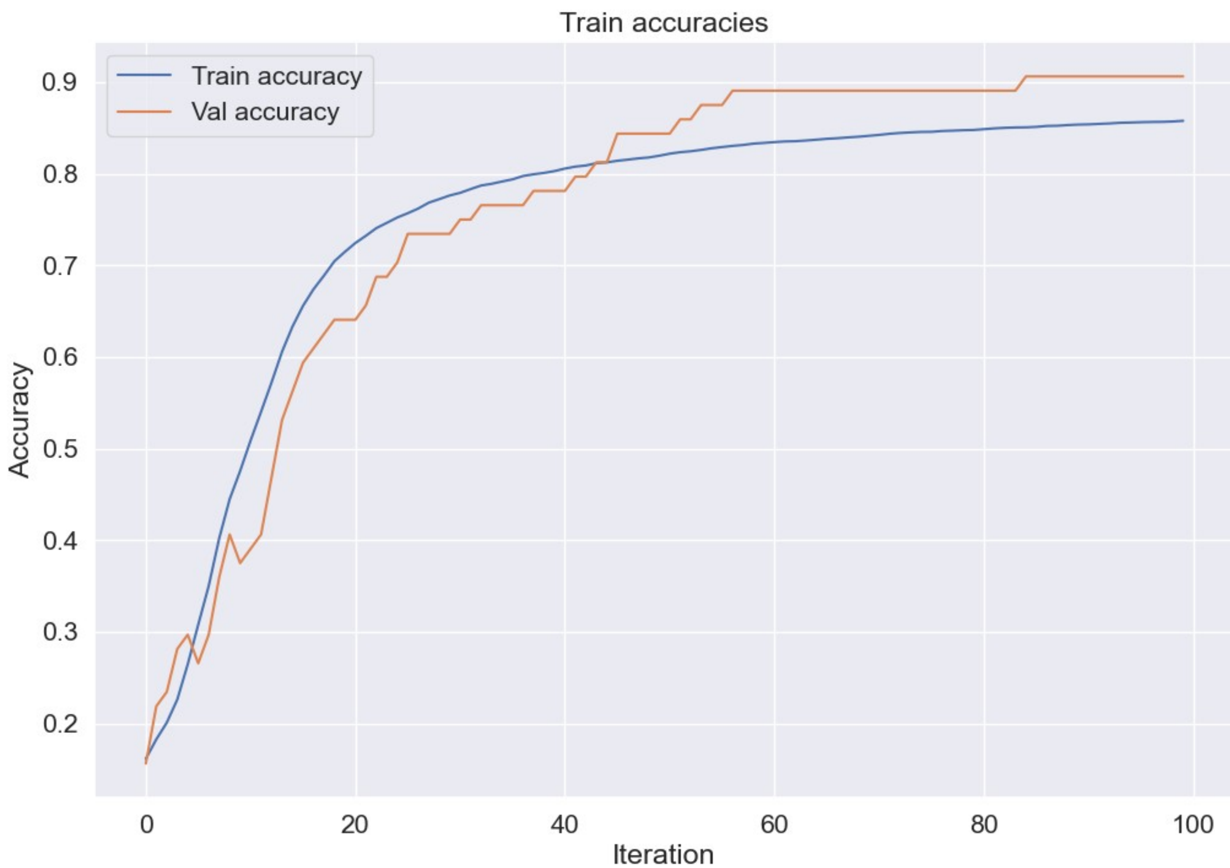
Ответ: 100 эпох

Визуализируем результаты

```
plt.figure(figsize=(12, 8))
plt.plot(range(len(train_losses)), train_losses, label='Train loss')
plt.plot(range(len(val_losses)), val_losses, label='Val loss')
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.legend()
plt.title("Train losses")
plt.show()

plt.figure(figsize=(12, 8))
plt.plot(range(len(train_accuracies)), train_accuracies, label='Train
accuracy')
plt.plot(range(len(val_accuracies)), val_accuracies, label='Val
accuracy')
plt.xlabel("Iteration")
plt.ylabel("Accuracy")
plt.legend()
plt.title("Train accuracies")
plt.show()
```

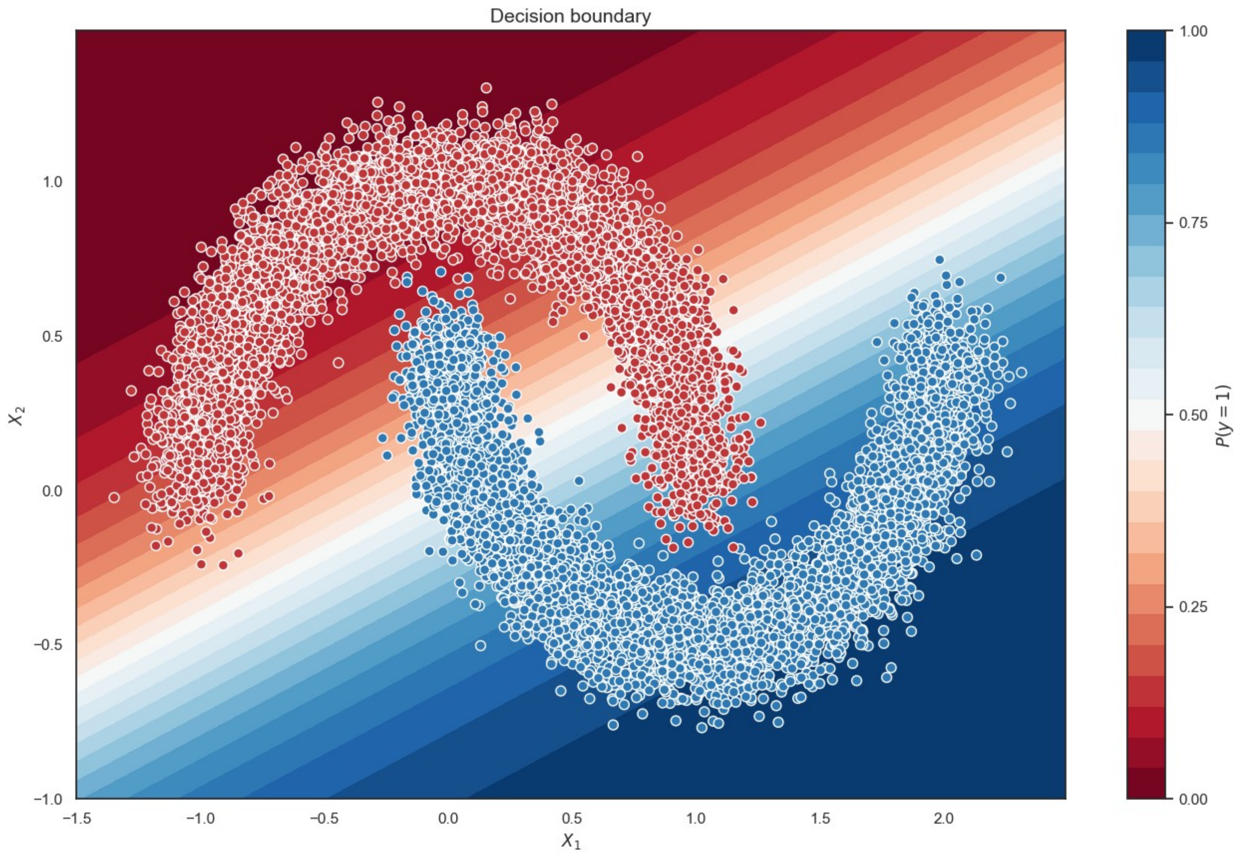


Train losses
Train loss
Val loss
Loss
Iteration
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
20
40
60
80
100




```
import numpy as np

sns.set(style="white")

xx, yy = np.mgrid[-1.5:2.5:.01, -1.:1.5:.01]
grid = np.c_[xx.ravel(), yy.ravel()]
batch = torch.from_numpy(grid).type(torch.float32)
with torch.no_grad():
    probs = torch.sigmoid(linear_regression(batch).reshape(xx.shape))
    probs = probs.numpy().reshape(xx.shape)

f, ax = plt.subplots(figsize=(16, 10))
ax.set_title("Decision boundary", fontsize=14)
contour = ax.contourf(xx, yy, probs, 25, cmap="RdBu",
                      vmin=0, vmax=1)
ax_c = f.colorbar(contour)
ax_c.set_label("$P(y = 1)$")
ax_c.set_ticks([0, .25, .5, .75, 1])

ax.scatter(X[100:,0], X[100:, 1], c=y[100:], s=50,
           cmap="RdBu", vmin=-.2, vmax=1.2,
           edgecolor="white", linewidth=1)
```

```
ax.set(xlabel="$X_1$", ylabel="$X_2$")
plt.show()
```



Задание. Реализуйте predict и посчитайте accuracy на test.

```
from sklearn.metrics import accuracy_score


@torch.no_grad()
def predict(dataloader, model):
    model.eval()
    predictions = np.array([])
    reals = np.array([])

    for x_batch, y_batch in dataloader:
        outp = model(x_batch).squeeze()
        probabilities = torch.sigmoid(outp)
        preds = (probabilities > 0.5).type(torch.long)
        predictions = np.hstack((predictions,
preds.cpu().numpy().flatten()))
        reals = np.hstack((reals, y_batch.cpu().numpy()))
    return (predictions.flatten(), reals)
```

```
predictions, reals = predict(val_dataloader, linear_regression)
total_accuracy = accuracy_score(reals, predictions)

print(f"Total Accuracy: {total_accuracy:.2f}")

Total Accuracy: 0.88
```

Вопрос 3

Какое `accuracy` получается после обучения?

Ответ: 0.88

Часть 2. Датасет MNIST

Датасет MNIST содержит рукописные цифры. Загрузим датасет и создадим DataLoader-ы. Пример можно найти в семинаре по полносвязным нейронным сетям.

```
import os
import random
from torchvision.datasets import MNIST
from torchvision import transforms as transforms
from torch.utils.data import DataLoader, Subset


data_root = os.getcwd()

data_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5), (0.5))
])

train_dataset = MNIST(root=data_root, train=True,
transform=data_transforms, download=True)
val_dataset = MNIST(root=data_root, train=False,
transform=data_transforms, download=True)

# Определяем процент данных
fraction = 0.5
num_train = len(train_dataset)
train_indices = random.sample(range(num_train), int(num_train *
fraction))

num_val = len(val_dataset)
val_indices = random.sample(range(num_val), int(num_val * fraction))
train_subset = Subset(train_dataset, train_indices)
val_subset = Subset(val_dataset, val_indices)

train_dataloader = DataLoader(train_subset, batch_size=256,
```

```
shuffle=True)
valid_dataloader = DataLoader(val_subset, batch_size=256,
shuffle=False)
```

Часть 2.1. Полносвязные нейронные сети

Сначала решим MNIST с помощью полносвязной нейронной сети.

```
class Identical(nn.Module):
    def forward(self, x):
        return x
```

Задание. Простая полносвязная нейронная сеть

Создайте полносвязную нейронную сеть с помощью класса Sequential. Сеть состоит из:

- Уплощения матрицы в вектор (nn.Flatten);
- Двух скрытых слоёв из 128 нейронов с активацией nn.ELU;
- Выходного слоя с 10 нейронами.

Задайте лосс для обучения (кросс-энтропия).

```
device = "cuda" if torch.cuda.is_available() else "cpu"
device = 'cpu'
device

'cpu'
```

Train loop (seriously)

Давайте разберемся с кодом ниже, который подойдет для 90% задач в будущем.

```
for epoch in range(max_epochs):  # <--------------- итерируемся по
датасету несколько раз
    for k, dataloader in loaders.items():  # <----- несколько
dataloader для train / valid / test
        for x_batch, y_batch in dataloader:  # <--- итерируемся по
датасету. Так как мы используем SGD а не GD, то берем батчи заданного
размера
            if k == "train":
                model.train()  # <------------------ переводим модель
в режим train
                optimizer.zero_grad()  # <--------- обнуляем градиенты
модели
                outp = model(x_batch)
                loss = criterion(outp, y_batch) # <-считаем "лосс" для
логистической регрессии
                loss.backward()  # <--------------- считаем градиенты
```

```
                optimizer.step()  # <-------------- делаем шаг
градиентного спуска
            else:  # <---------------------------- test/eval
                model.eval()  # <------------------ переводим модель в
режим eval
                with torch.no_grad():  # <--------- НЕ считаем
градиенты
                    outp = model(x_batch)  # <------------- получаем
"логиты" из модели
            count_metrics(outp, y_batch)  # <-------------- считаем
метрики
```

Задание. Дополните цикл обучения.

```
def plot_metrics(train_losses, val_losses, accuracy):
    epochs = range(1, len(train_losses) + 1)

    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(epochs, train_losses, label='Train Loss')
    plt.plot(epochs, val_losses, label='Validation Loss')
    plt.title('Loss per Epoch')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(epochs, accuracy["train"], label='Train Accuracy')
    plt.plot(epochs, accuracy["valid"], label='Validation Accuracy')
    plt.title('Accuracy per Epoch')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.tight_layout()
    plt.grid(True)
    plt.show()

def train(model, max_epochs=10, show_stats=False):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), weight_decay=1e-
5)

    train_accuracy, valid_accuracy = [], []
    train_losses, val_losses = [], []

    for epoch in range(max_epochs):
        model.train()  # Устанавливаем режим обучения
        train_epoch_loss, train_correct, train_total = 0, 0, 0
```

```
        for x_batch, y_batch in train_dataloader:
            optimizer.zero_grad()
            outp = model(x_batch)
            loss = criterion(outp, y_batch)
            loss.backward()
            optimizer.step()
            train_epoch_loss += loss.item()

            preds = outp.argmax(dim=1)
            train_correct += (preds == y_batch).sum().item()
            train_total += y_batch.size(0)

        train_loss_avg = train_epoch_loss / len(train_dataloader)
        train_accuracy.append(train_correct / train_total)
        train_losses.append(train_loss_avg)

        model.eval()  # Устанавливаем режим оценки
        val_epoch_loss, val_correct, val_total = 0, 0, 0

        with torch.no_grad():
            for x_batch, y_batch in valid_dataloader:
                outp = model(x_batch)
                loss = criterion(outp, y_batch)
                val_epoch_loss += loss.item()

                preds = outp.argmax(dim=1)
                val_correct += (preds == y_batch).sum().item()
                val_total += y_batch.size(0)

        val_loss_avg = val_epoch_loss / len(valid_dataloader)
        valid_accuracy.append(val_correct / val_total)
        val_losses.append(val_loss_avg)

        print(f"Epoch {epoch + 1}/{max_epochs}")
        print(f"Train Loss: {train_loss_avg:.4f}, Train Accuracy:
{train_accuracy[-1]:.4f}")
        print(f"Valid Loss: {val_loss_avg:.4f}, Valid Accuracy:
{valid_accuracy[-1]:.4f}")

    if show_stats:
        plot_metrics(train_losses, val_losses, {"train":
train_accuracy, "valid": valid_accuracy})

    return train_losses, val_losses, {"train": train_accuracy,
"valid": valid_accuracy}
```

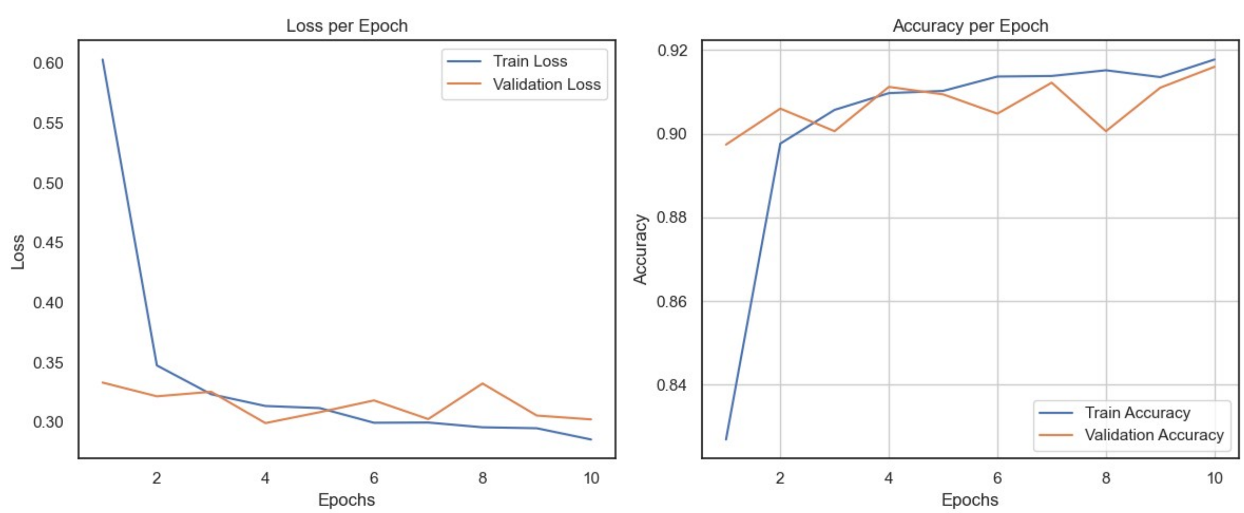
Задание. Протестируйте разные функции активации.

Попробуйте разные функции активации. Для каждой функции активации посчитайте массив validation accuracy. Лучше реализовать это в виде функции, берущей на вход активацию и получающей массив из accuracies.

```python
accuracies = dict()

plain_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 128),
    nn.Linear(128, 128),
    nn.Linear(128, 10)
)

ReLU_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 128),
    nn.ReLU(),
    nn.Linear(128, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)

ELU_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 128),
    nn.ELU(),
    nn.Linear(128, 128),
    nn.ELU(),
    nn.Linear(128, 10)
)

LeakyReLU_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 128),
    nn.LeakyReLU(),
    nn.Linear(128, 128),
    nn.LeakyReLU(),
    nn.Linear(128, 10)
)

Sigmoid_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 128),
    nn.Sigmoid(),
    nn.Linear(128, 128),
    nn.Sigmoid(),
    nn.Linear(128, 10)
)
```

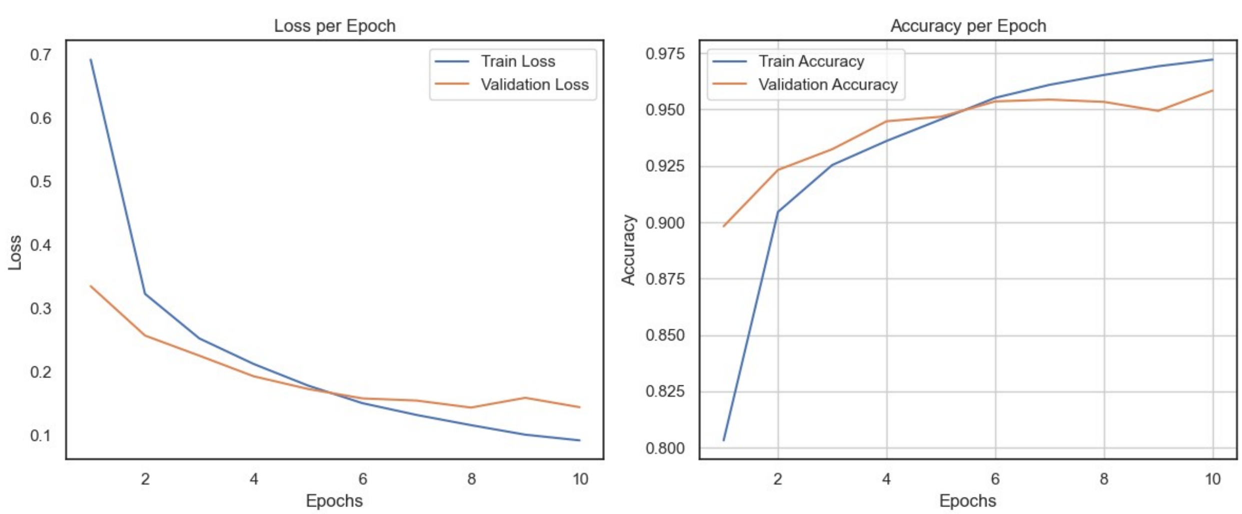
```python
plain_train_losses, plain_val_losses, plain_accuracy =
train(plain_model, show_stats=True)
```

```
Epoch 1/10
Train Loss: 0.6032, Train Accuracy: 0.8269
Valid Loss: 0.3331, Valid Accuracy: 0.8974
Epoch 2/10
Train Loss: 0.3474, Train Accuracy: 0.8976
Valid Loss: 0.3216, Valid Accuracy: 0.9060
Epoch 3/10
Train Loss: 0.3233, Train Accuracy: 0.9057
Valid Loss: 0.3254, Valid Accuracy: 0.9006
Epoch 4/10
Train Loss: 0.3135, Train Accuracy: 0.9097
Valid Loss: 0.2992, Valid Accuracy: 0.9112
Epoch 5/10
Train Loss: 0.3117, Train Accuracy: 0.9102
Valid Loss: 0.3083, Valid Accuracy: 0.9094
Epoch 6/10
Train Loss: 0.2995, Train Accuracy: 0.9137
Valid Loss: 0.3182, Valid Accuracy: 0.9048
Epoch 7/10
Train Loss: 0.2997, Train Accuracy: 0.9138
Valid Loss: 0.3026, Valid Accuracy: 0.9122
Epoch 8/10
Train Loss: 0.2957, Train Accuracy: 0.9152
Valid Loss: 0.3323, Valid Accuracy: 0.9006
Epoch 9/10
Train Loss: 0.2949, Train Accuracy: 0.9135
Valid Loss: 0.3055, Valid Accuracy: 0.9110
Epoch 10/10
Train Loss: 0.2855, Train Accuracy: 0.9177
Valid Loss: 0.3023, Valid Accuracy: 0.9160
```



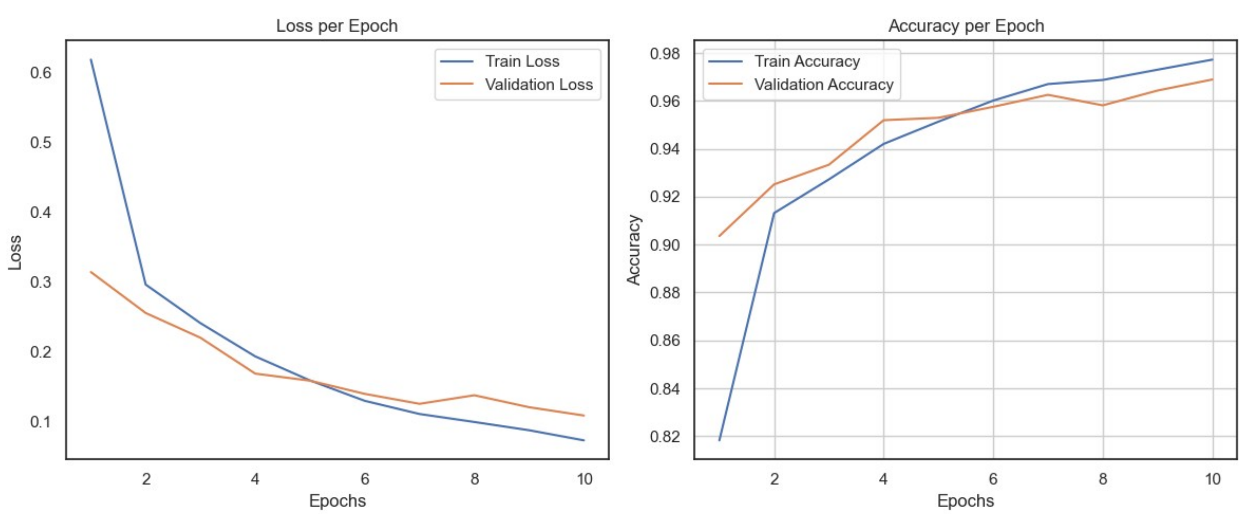
```python
ReLU_train_losses, ReLU_val_losses, ReLU_accuracy = train(ReLU_model,
show_stats=True)
```

```
Epoch 1/10
Train Loss: 0.6911, Train Accuracy: 0.8032
Valid Loss: 0.3346, Valid Accuracy: 0.8982
Epoch 2/10
Train Loss: 0.3225, Train Accuracy: 0.9046
Valid Loss: 0.2569, Valid Accuracy: 0.9232
Epoch 3/10
Train Loss: 0.2523, Train Accuracy: 0.9254
Valid Loss: 0.2252, Valid Accuracy: 0.9324
Epoch 4/10
Train Loss: 0.2121, Train Accuracy: 0.9361
Valid Loss: 0.1928, Valid Accuracy: 0.9448
Epoch 5/10
Train Loss: 0.1782, Train Accuracy: 0.9456
Valid Loss: 0.1728, Valid Accuracy: 0.9468
Epoch 6/10
Train Loss: 0.1505, Train Accuracy: 0.9552
Valid Loss: 0.1579, Valid Accuracy: 0.9536
Epoch 7/10
Train Loss: 0.1319, Train Accuracy: 0.9610
Valid Loss: 0.1545, Valid Accuracy: 0.9544
Epoch 8/10
Train Loss: 0.1158, Train Accuracy: 0.9653
Valid Loss: 0.1435, Valid Accuracy: 0.9534
Epoch 9/10
Train Loss: 0.1008, Train Accuracy: 0.9692
Valid Loss: 0.1589, Valid Accuracy: 0.9494
Epoch 10/10
Train Loss: 0.0919, Train Accuracy: 0.9721
Valid Loss: 0.1441, Valid Accuracy: 0.9584
```

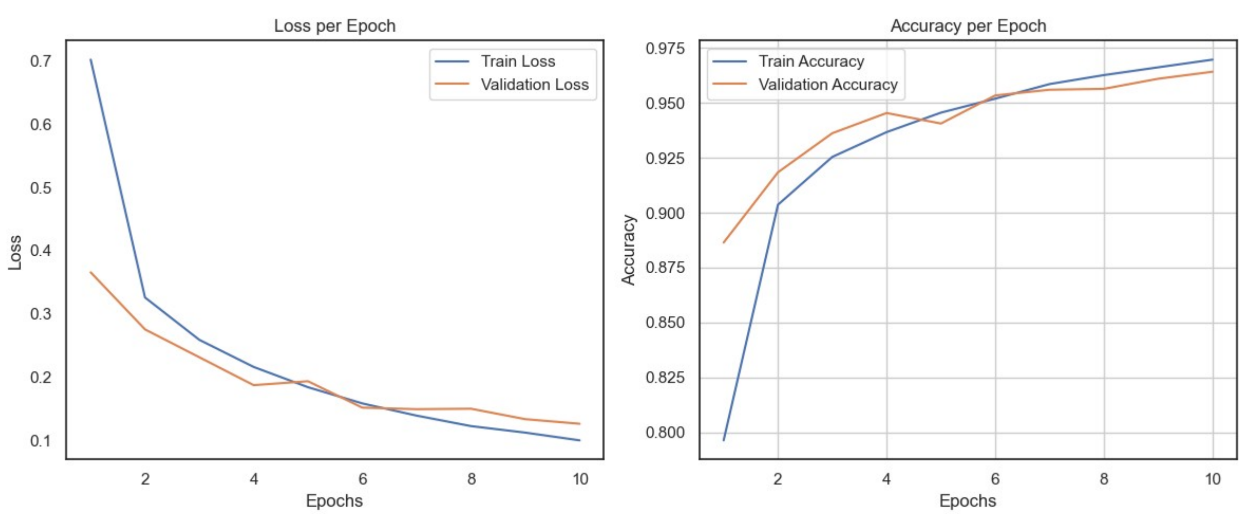


```
ELU_train_losses, ELU_val_losses, ELU_accuracy = train(ELU_model,
show_stats=True)
```

```
Epoch 1/10
Train Loss: 0.6185, Train Accuracy: 0.8182
Valid Loss: 0.3144, Valid Accuracy: 0.9036
Epoch 2/10
Train Loss: 0.2966, Train Accuracy: 0.9132
Valid Loss: 0.2559, Valid Accuracy: 0.9252
Epoch 3/10
Train Loss: 0.2413, Train Accuracy: 0.9273
Valid Loss: 0.2204, Valid Accuracy: 0.9334
Epoch 4/10
Train Loss: 0.1938, Train Accuracy: 0.9421
Valid Loss: 0.1691, Valid Accuracy: 0.9520
Epoch 5/10
Train Loss: 0.1593, Train Accuracy: 0.9514
Valid Loss: 0.1590, Valid Accuracy: 0.9530
Epoch 6/10
Train Loss: 0.1301, Train Accuracy: 0.9602
Valid Loss: 0.1401, Valid Accuracy: 0.9576
Epoch 7/10
Train Loss: 0.1113, Train Accuracy: 0.9671
Valid Loss: 0.1258, Valid Accuracy: 0.9626
Epoch 8/10
Train Loss: 0.0998, Train Accuracy: 0.9688
Valid Loss: 0.1381, Valid Accuracy: 0.9582
Epoch 9/10
Train Loss: 0.0880, Train Accuracy: 0.9731
Valid Loss: 0.1210, Valid Accuracy: 0.9644
Epoch 10/10
Train Loss: 0.0735, Train Accuracy: 0.9773
Valid Loss: 0.1091, Valid Accuracy: 0.9690
```

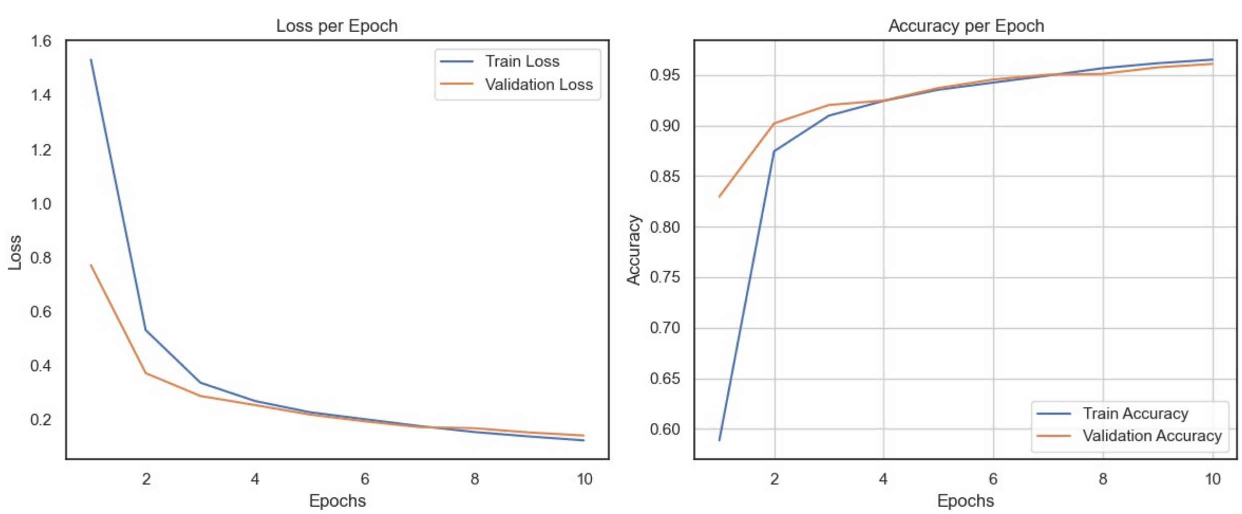


```
LeakyReLU_train_losses, LeakyReLU_val_losses, LeakyReLU_accuracy =
train(LeakyReLU_model, show_stats=True)

Epoch 1/10
Train Loss: 0.7022, Train Accuracy: 0.7963
Valid Loss: 0.3656, Valid Accuracy: 0.8864
Epoch 2/10
Train Loss: 0.3258, Train Accuracy: 0.9036
Valid Loss: 0.2753, Valid Accuracy: 0.9184
Epoch 3/10
Train Loss: 0.2588, Train Accuracy: 0.9254
Valid Loss: 0.2312, Valid Accuracy: 0.9362
Epoch 4/10
Train Loss: 0.2159, Train Accuracy: 0.9367
Valid Loss: 0.1870, Valid Accuracy: 0.9454
Epoch 5/10
Train Loss: 0.1838, Train Accuracy: 0.9456
Valid Loss: 0.1932, Valid Accuracy: 0.9406
Epoch 6/10
Train Loss: 0.1581, Train Accuracy: 0.9519
Valid Loss: 0.1514, Valid Accuracy: 0.9534
Epoch 7/10
Train Loss: 0.1388, Train Accuracy: 0.9586
Valid Loss: 0.1490, Valid Accuracy: 0.9560
Epoch 8/10
Train Loss: 0.1223, Train Accuracy: 0.9627
Valid Loss: 0.1498, Valid Accuracy: 0.9564
Epoch 9/10
Train Loss: 0.1119, Train Accuracy: 0.9662
Valid Loss: 0.1332, Valid Accuracy: 0.9610
Epoch 10/10
Train Loss: 0.0997, Train Accuracy: 0.9697
Valid Loss: 0.1260, Valid Accuracy: 0.9642
```



```
Sigmoid_train_losses, Sigmoid_val_losses, Sigmoid_accuracy =
train(Sigmoid_model, show_stats=True)
```

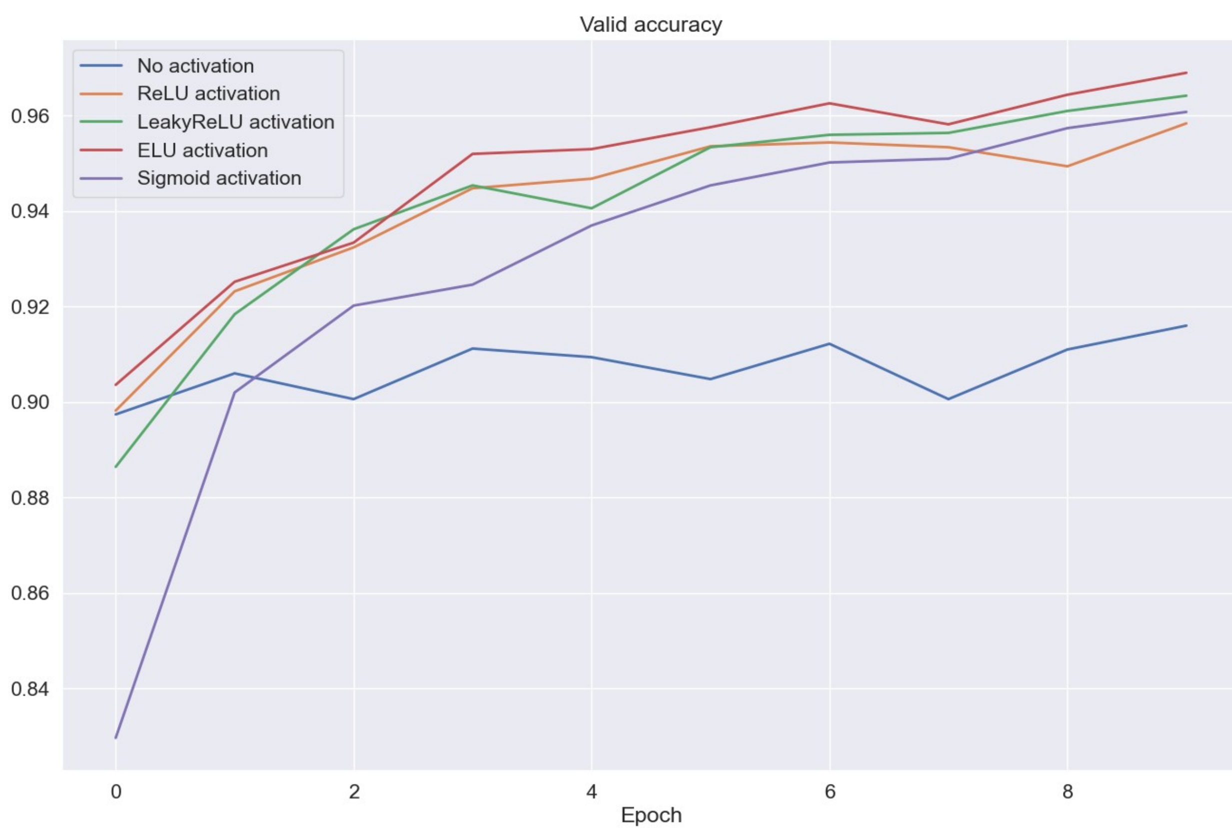
```
Epoch 1/10
Train Loss: 1.5318, Train Accuracy: 0.5887
Valid Loss: 0.7709, Valid Accuracy: 0.8296
Epoch 2/10
Train Loss: 0.5307, Train Accuracy: 0.8744
Valid Loss: 0.3715, Valid Accuracy: 0.9020
Epoch 3/10
Train Loss: 0.3359, Train Accuracy: 0.9097
Valid Loss: 0.2867, Valid Accuracy: 0.9202
Epoch 4/10
Train Loss: 0.2678, Train Accuracy: 0.9242
Valid Loss: 0.2530, Valid Accuracy: 0.9246
Epoch 5/10
Train Loss: 0.2266, Train Accuracy: 0.9353
Valid Loss: 0.2180, Valid Accuracy: 0.9370
Epoch 6/10
Train Loss: 0.2008, Train Accuracy: 0.9424
Valid Loss: 0.1930, Valid Accuracy: 0.9454
Epoch 7/10
Train Loss: 0.1755, Train Accuracy: 0.9493
Valid Loss: 0.1715, Valid Accuracy: 0.9502
Epoch 8/10
Train Loss: 0.1534, Train Accuracy: 0.9566
Valid Loss: 0.1676, Valid Accuracy: 0.9510
Epoch 9/10
Train Loss: 0.1369, Train Accuracy: 0.9615
Valid Loss: 0.1517, Valid Accuracy: 0.9574
Epoch 10/10
Train Loss: 0.1224, Train Accuracy: 0.9650
Valid Loss: 0.1404, Valid Accuracy: 0.9608
```



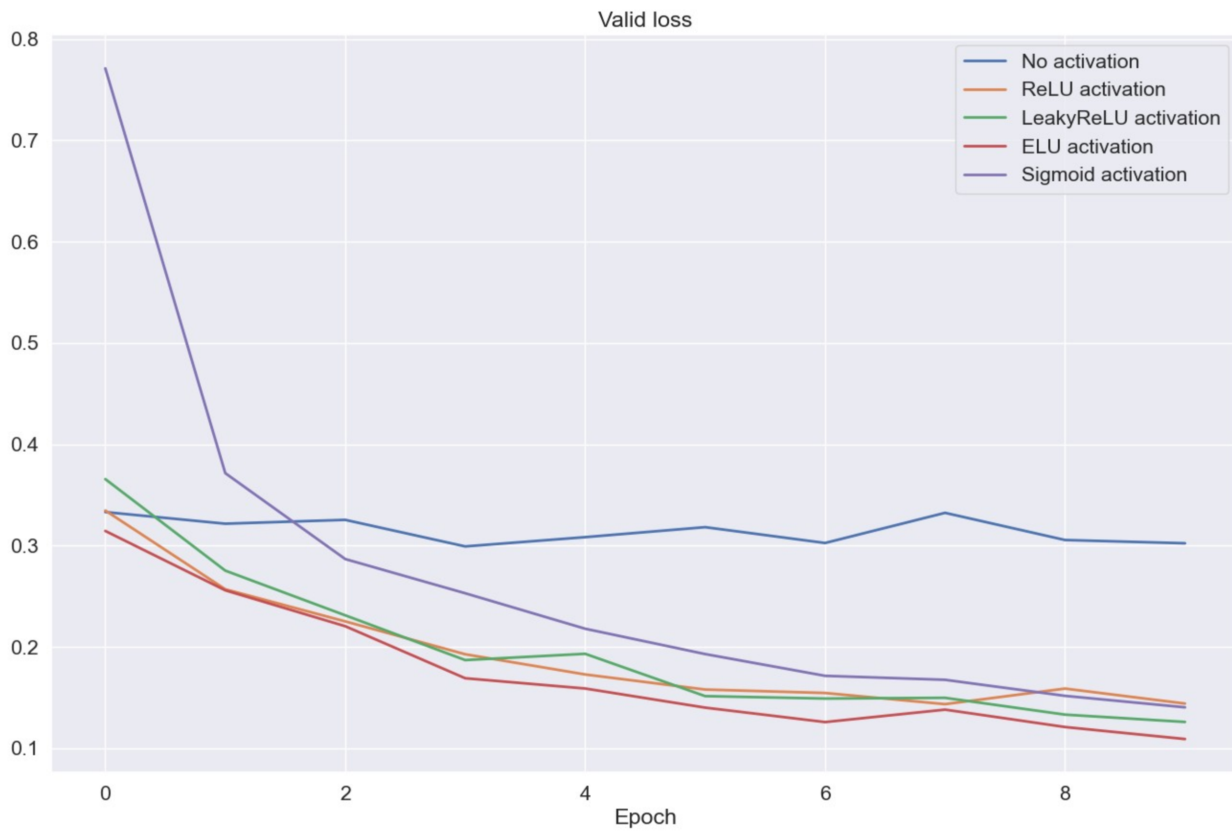
Accuracy

Построим график accuracy/epoch для каждой функции активации.

```
max_epochs = len(plain_accuracy['valid'])
# sns.set(style="darkgrid", font_scale=1.4)

plt.figure(figsize=(16, 10))
plt.title("Valid accuracy")
plt.plot(range(max_epochs), plain_accuracy['valid'], label="No activation", linewidth=2)
plt.plot(range(max_epochs), ReLU_accuracy['valid'], label="ReLU activation", linewidth=2)
plt.plot(range(max_epochs), LeakyReLU_accuracy['valid'], label="LeakyReLU activation", linewidth=2)
plt.plot(range(max_epochs), ELU_accuracy['valid'], label="ELU activation", linewidth=2)
plt.plot(range(max_epochs), Sigmoid_accuracy['valid'], label="Sigmoid activation", linewidth=2)
plt.legend()
plt.grid(True)
plt.xlabel("Epoch")
plt.show()
```

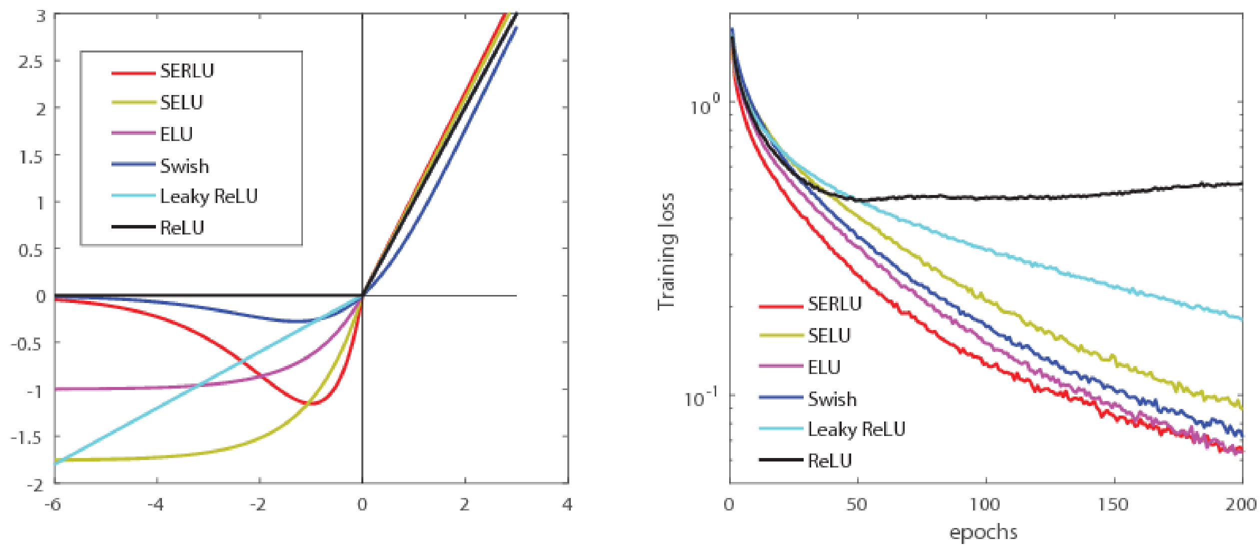


```
plt.figure(figsize=(16, 10))
plt.title("Valid loss")
plt.plot(range(max_epochs), plain_val_losses, label="No activation",
linewidth=2)
plt.plot(range(max_epochs), ReLU_val_losses, label="ReLU activation",
linewidth=2)
plt.plot(range(max_epochs), LeakyReLU_val_losses, label="LeakyReLU
activation", linewidth=2)
plt.plot(range(max_epochs), ELU_val_losses, label="ELU activation",
linewidth=2)
plt.plot(range(max_epochs), Sigmoid_val_losses, label="Sigmoid
activation", linewidth=2)
plt.legend()
plt.grid(True)
plt.xlabel("Epoch")
plt.show()
```



Вопрос 4. Какая из активаций показала наивысший `accuracy` к концу обучения?

Ответ: Очевидно ELU как я и считал до эксперимента



(a): Different activation functions

(b): Performance on CIFAR10 without dropout

Часть 2.2 Сверточные нейронные сети

Ядра

Сначала немного поработам с самим понятием ядра свёртки.

```
import cv2
sns.set(style="white")
img = cv2.imread("sample_photo.jpg")
RGB_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.figure(figsize=(12, 8))
plt.imshow(RGB_img)
plt.axis('off')
plt.show()
```

Попробуйте посмотреть как различные свертки влияют на фото. Например, попробуйте

А)

```
[0, 0, 0],
[0, 1, 0],
[0, 0, 0]
```

Б)

```
[0, 1, 0],
[0, -2, 0],
[0, 1, 0]
```

В)

```
[0, 0, 0],
[1, -2, 1],
[0, 0, 0]
```

Г)

```
[0, 1, 0],
[1, -4, 1],
[0, 1, 0]
```

Д)

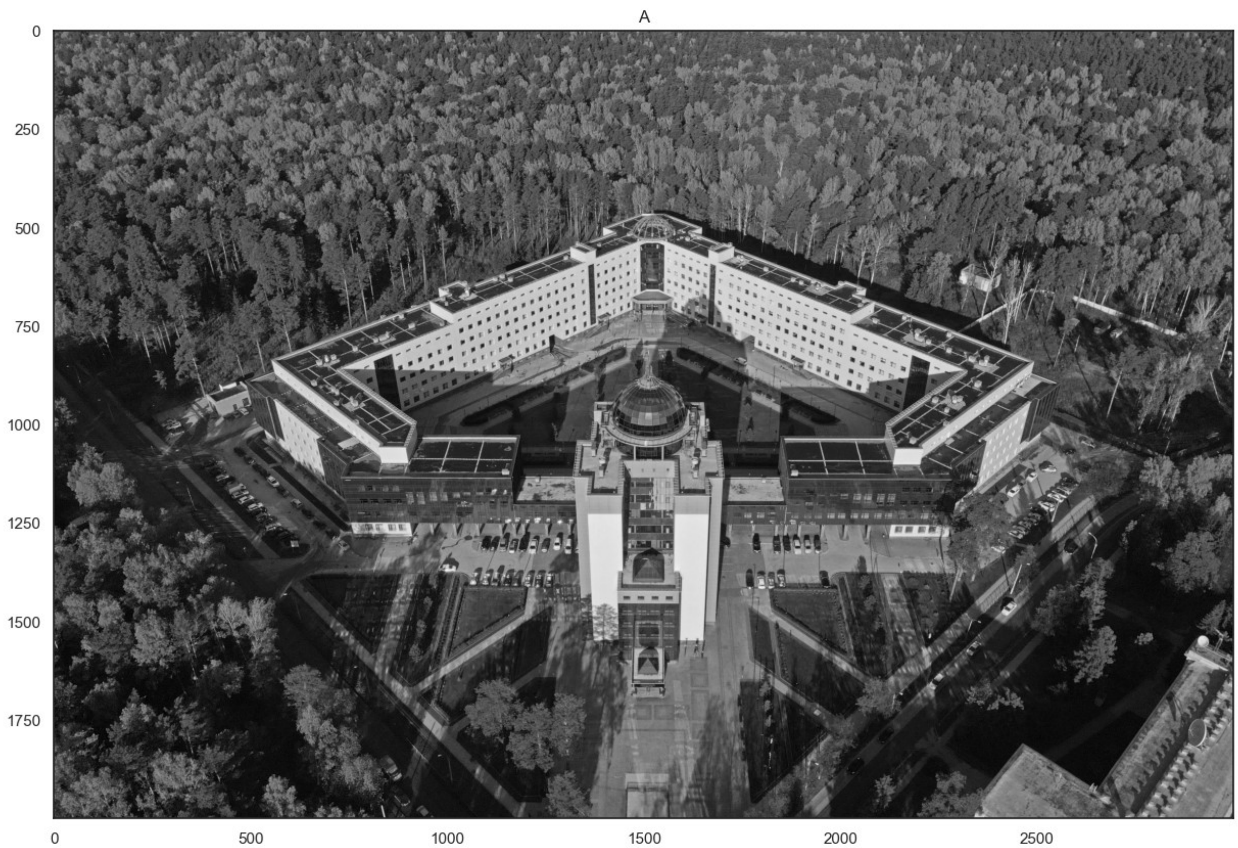
```
[0, -1, 0],
[-1, 5, -1],
[0, -1, 0]
```

Е)

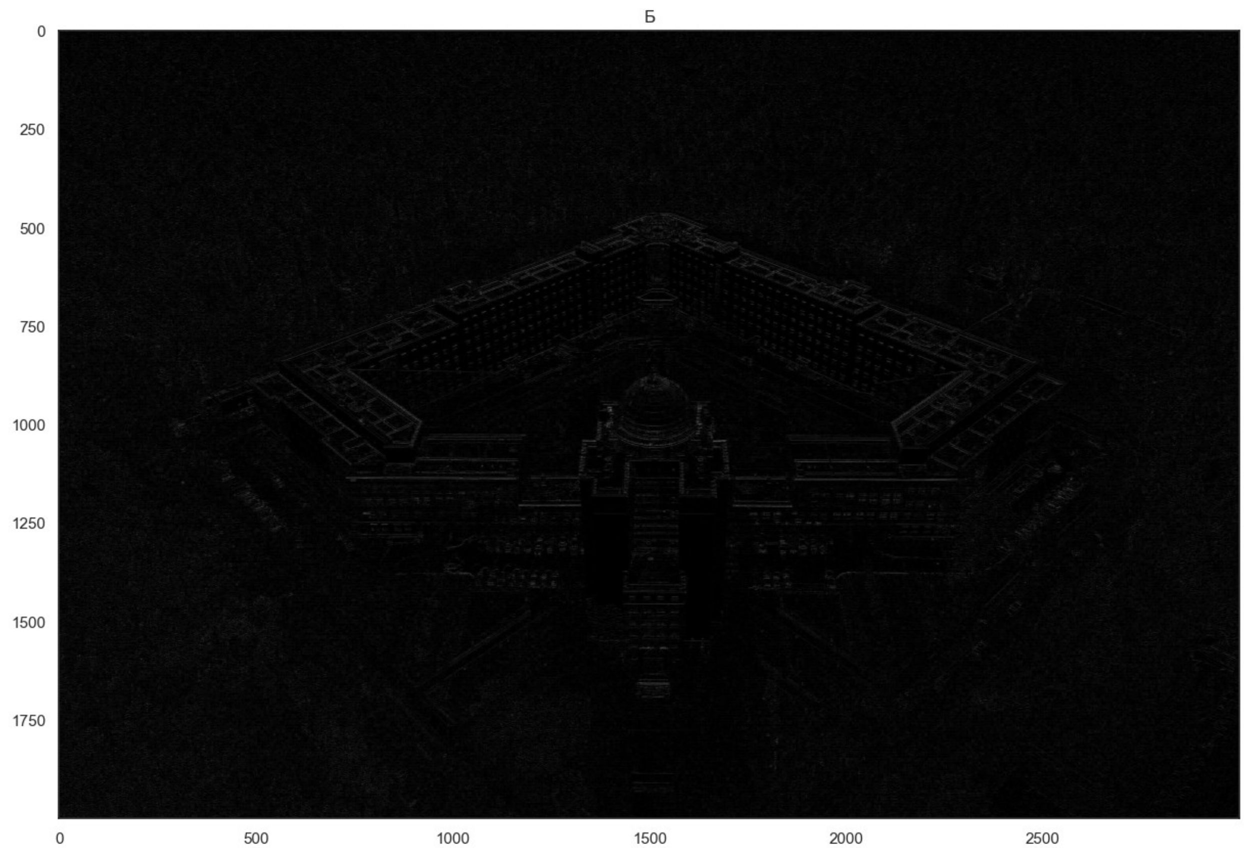
```
[0.0625, 0.125, 0.0625],
[0.125, 0.25, 0.125],
[0.0625, 0.125, 0.0625]
```

Не стесняйтесь пробовать свои варианты!

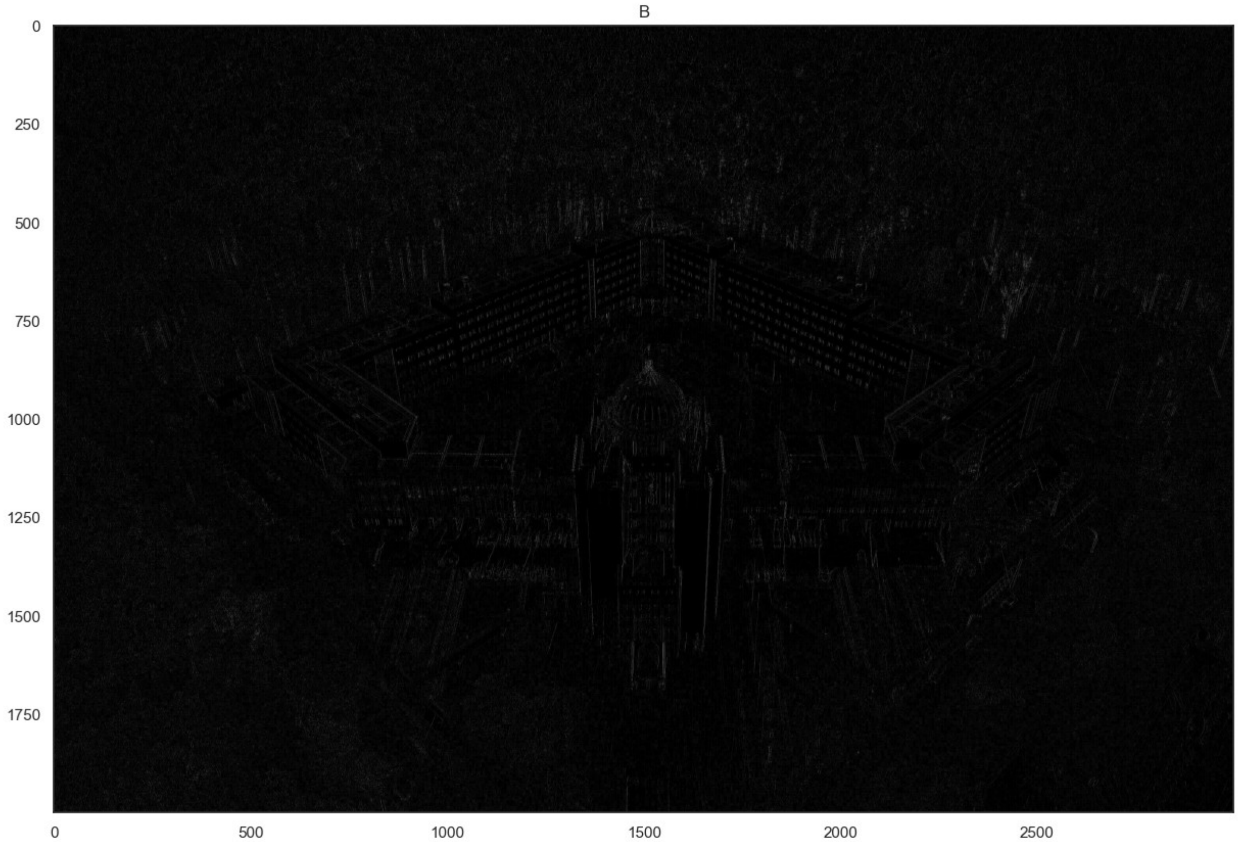
```
img_t = torch.from_numpy(RGB_img).type(torch.float32).unsqueeze(0)

kernels = {
    "А": torch.tensor([[0, 0, 0], [0, 1, 0], [0, 0, 0]],
dtype=torch.float32).reshape(1, 1, 3, 3),
    "Б": torch.tensor([[0, 1, 0], [0, -2, 0], [0, 1, 0]],
dtype=torch.float32).reshape(1, 1, 3, 3),
    "В": torch.tensor([[0, 0, 0], [1, -2, 1], [0, 0, 0]],
dtype=torch.float32).reshape(1, 1, 3, 3),
    "Г": torch.tensor([[0, 1, 0], [1, -4, 1], [0, 1, 0]],
dtype=torch.float32).reshape(1, 1, 3, 3),
    "Д": torch.tensor([[0, -1, 0], [1, 5, 1], [0, -1, 0]],
dtype=torch.float32).reshape(1, 1, 3, 3),
    "Е": torch.tensor([[0.0625, 0.125, 0.0625], [0.125, 0.25, 0.125],
[0.0625, 0.125, 0.0625]], dtype=torch.float32).reshape(1, 1, 3, 3),
}

img_t = img_t.permute(0, 3, 1, 2)  # [BS, H, W, C] -> [BS, C, H, W]
for name, kernel in kernels.items():
    kernel = kernel.repeat(3, 3, 1, 1)
    filtered_img = nn.ReflectionPad2d(1)(img_t)

    result = F.conv2d(filtered_img, kernel)[0]
    result_np = result.permute(1, 2, 0).numpy() / 256 / 3

    plt.figure(figsize=(15, 10))
    plt.imshow(result_np)
    plt.title(name)
    plt.show()
```



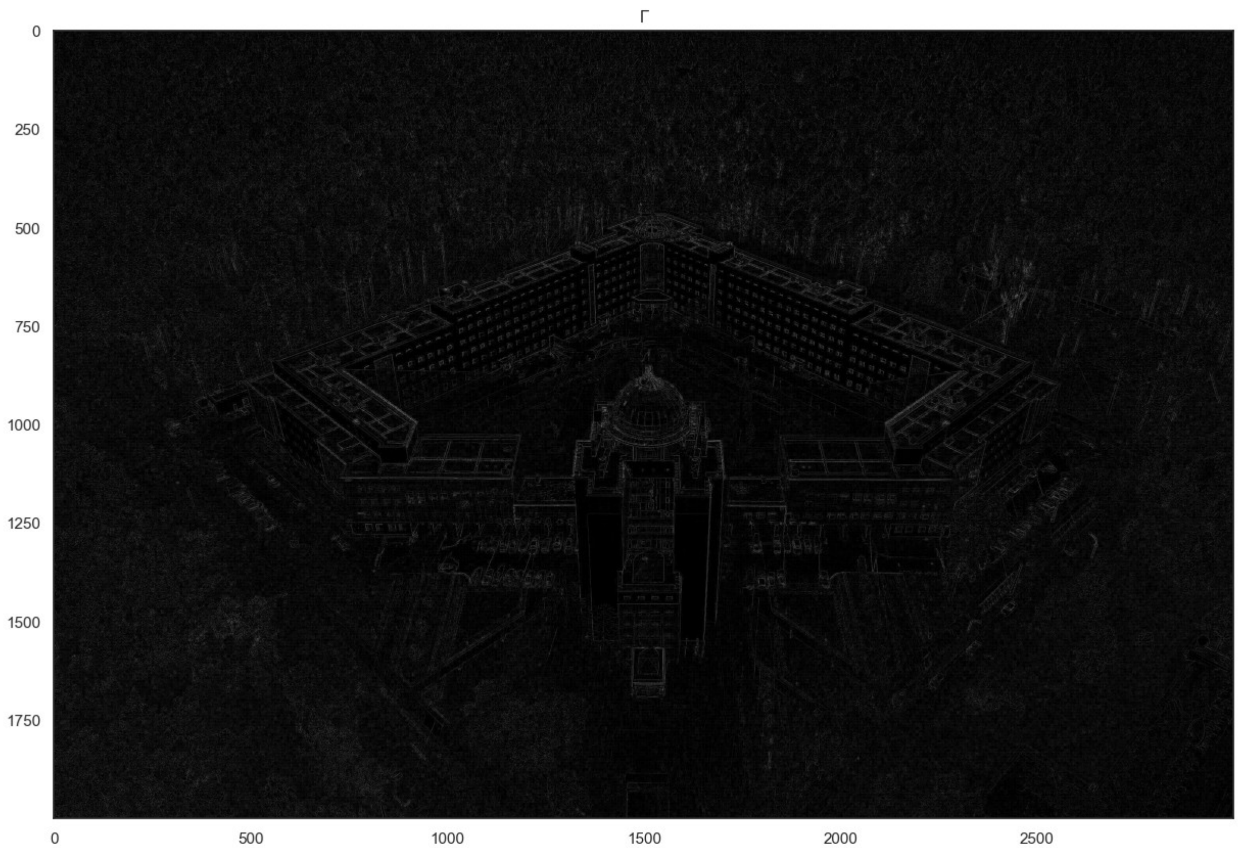

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers). Got range [-
1.2539062..1.1614584].
```



```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers). Got range [-
1.1653646..1.0507812].
```

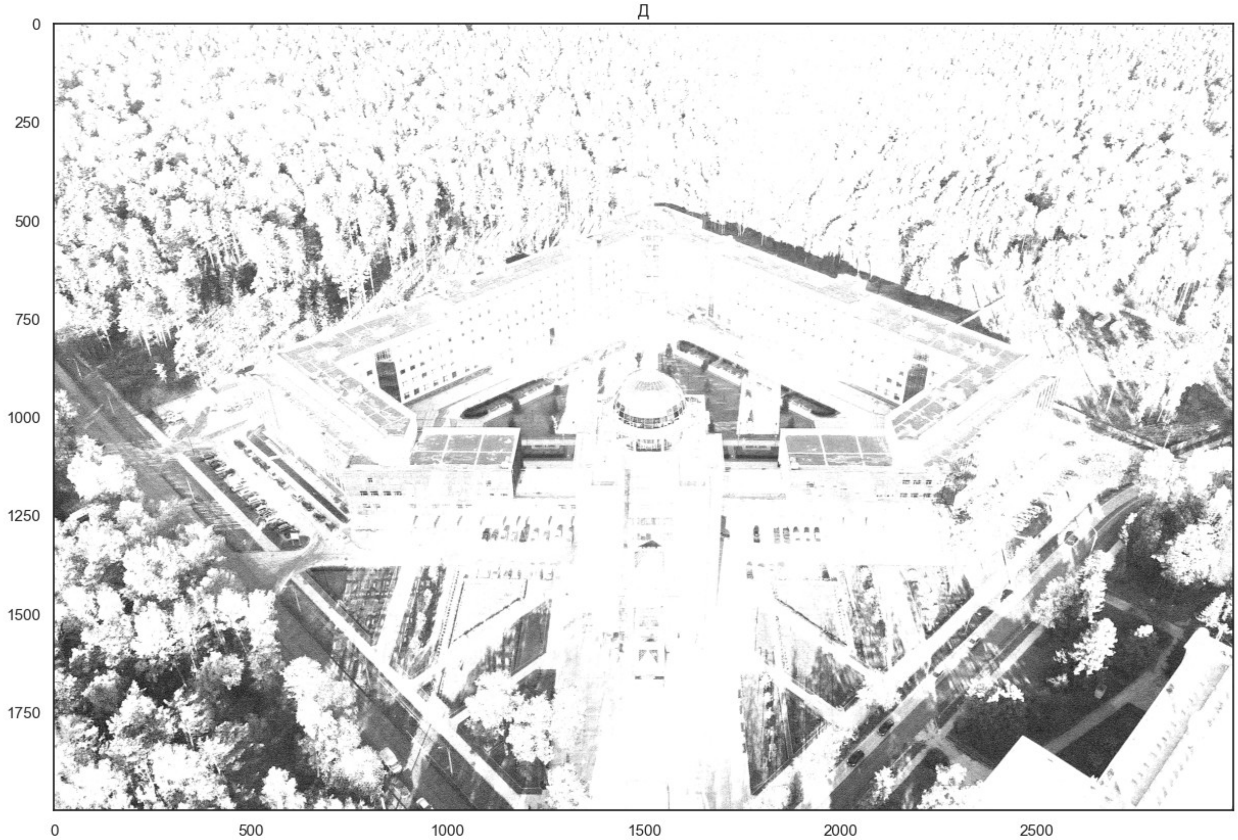


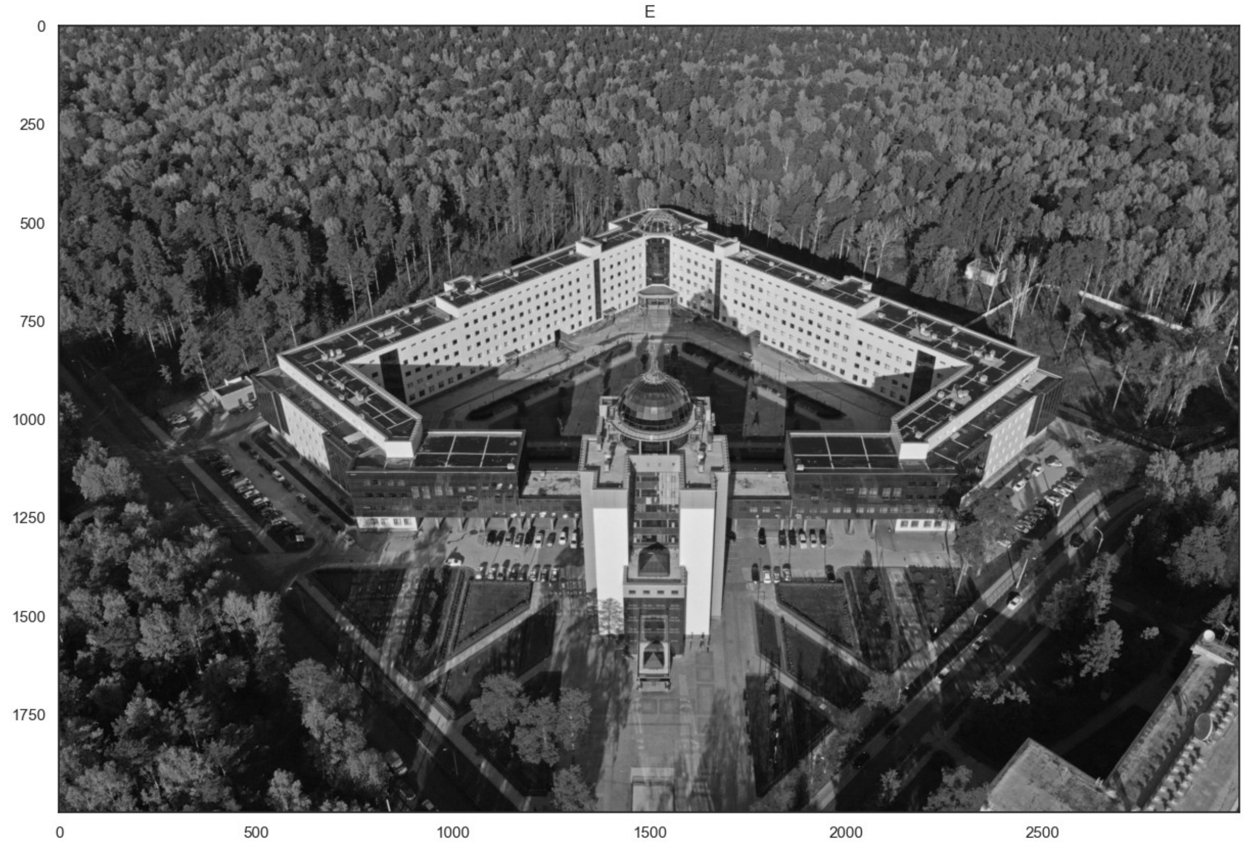

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers). Got range [-
1.7565104..1.5585938].
```




```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers). Got range [-
0.3997396..5.7421875].
```

д





Вопрос 5. Как можно описать действия ядер, приведенных выше? Сопоставьте для каждой буквы число.

1) Размытие: Е

2) Увеличение резкости: Д

3) Тождественное преобразование: А

4) Выделение вертикальных границ: В

5) Выделение горизонтальных границ: Б

6) Выделение границ: Г

Задание. Реализуйте LeNet

Если мы сделаем параметры сверток обучаемыми, то можем добиться хороших результатов для задач компьютерного зрения. Реализуйте архитектуру LeNet, предложенную еще в 1998 году! На этот раз используйте модульную структуру (без помощи класса Sequential).

Наша нейронная сеть будет состоять из

- Свёртки 3x3 (1 карта на входе, 6 на выходе) с активацией ReLU;
- MaxPooling-а 2x2;

- Свёртки 3x3 (6 карт на входе, 16 на выходе) с активацией ReLU;
- MaxPooling-а 2x2;
- Уплощения (nn.Flatten);
- Полносвязного слоя со 120 нейронами и активацией ReLU;
- Полносвязного слоя с 84 нейронами и активацией ReLU;
- Выходного слоя из 10 нейронов.

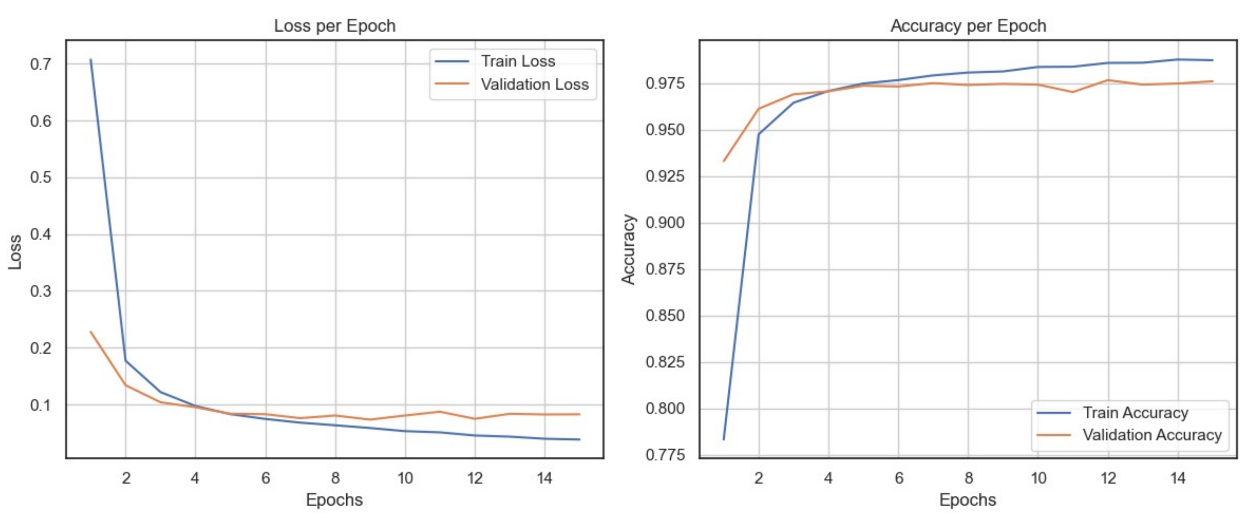
```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 6, 3), #[1, 28, 28] -> [6, 26, 26]
            nn.MaxPool2d(2),    #[6, 26, 26] -> [6, 13, 13]
            nn.Conv2d(6, 16, 3),#[6, 13, 13] -> [16, 11, 11]
            nn.MaxPool2d(2),    #[16, 11, 11] -> [16, 5, 5]
            nn.Flatten(),       #[16, 5, 5] -> [120]
            nn.Linear(16 * 5 * 5, 120),
            nn.Linear(120, 84),
            nn.Linear(84, 10),
        )

    def forward(self, x):
        return self.model(x)
```

Задание. Обучите CNN

Используйте код обучения, который вы написали для полносвязной нейронной сети.

```
leNET = LeNet().to(device)
leNET_train_losses, leNET_val_losses, leNET_accuracy = train(leNET,
max_epochs=15, show_stats=True)

Epoch 1/15
Train Loss: 0.7076, Train Accuracy: 0.7833
Valid Loss: 0.2278, Valid Accuracy: 0.9330
Epoch 2/15
Train Loss: 0.1770, Train Accuracy: 0.9475
Valid Loss: 0.1338, Valid Accuracy: 0.9612
Epoch 3/15
Train Loss: 0.1218, Train Accuracy: 0.9645
Valid Loss: 0.1038, Valid Accuracy: 0.9690
Epoch 4/15
Train Loss: 0.0972, Train Accuracy: 0.9708
Valid Loss: 0.0951, Valid Accuracy: 0.9706
Epoch 5/15
Train Loss: 0.0828, Train Accuracy: 0.9748
Valid Loss: 0.0837, Valid Accuracy: 0.9736
Epoch 6/15
Train Loss: 0.0745, Train Accuracy: 0.9766
```

```
Valid Loss: 0.0828, Valid Accuracy: 0.9732
Epoch 7/15
Train Loss: 0.0679, Train Accuracy: 0.9792
Valid Loss: 0.0758, Valid Accuracy: 0.9750
Epoch 8/15
Train Loss: 0.0633, Train Accuracy: 0.9807
Valid Loss: 0.0805, Valid Accuracy: 0.9740
Epoch 9/15
Train Loss: 0.0585, Train Accuracy: 0.9813
Valid Loss: 0.0732, Valid Accuracy: 0.9746
Epoch 10/15
Train Loss: 0.0529, Train Accuracy: 0.9837
Valid Loss: 0.0805, Valid Accuracy: 0.9742
Epoch 11/15
Train Loss: 0.0508, Train Accuracy: 0.9838
Valid Loss: 0.0872, Valid Accuracy: 0.9702
Epoch 12/15
Train Loss: 0.0453, Train Accuracy: 0.9859
Valid Loss: 0.0747, Valid Accuracy: 0.9766
Epoch 13/15
Train Loss: 0.0432, Train Accuracy: 0.9860
Valid Loss: 0.0835, Valid Accuracy: 0.9742
Epoch 14/15
Train Loss: 0.0395, Train Accuracy: 0.9877
Valid Loss: 0.0823, Valid Accuracy: 0.9748
Epoch 15/15
Train Loss: 0.0382, Train Accuracy: 0.9873
Valid Loss: 0.0826, Valid Accuracy: 0.9760
```



Сравним с предыдущем пунктом

Вопрос 6 Какое `accuracy` получается после обучения с точностью до двух знаков после запятой?

Ответ: 0.9766, тогда как у MLP максимум был 0.9678, видим разницу менее чем в 0.01. Короче для задачи MNIST можно было и без CNN обойтись)