Всем привет! Сегодня вы впервые попробуете написать свою собственную нейронную сеть и попробовать ее обучить. Мы будем работать с картинками, но пока что не совсем тем способом, которым лучше всего это делать, но должно получиться неплохо.

Будем работать с датасетом `Kuzushiji-MNIST` (`KMNIST`). Это рукописные буквы, изображения имеют размер (28, 28, 1) и разделены на 10 классов, по ссылке можно прочитать подробнее.

In [ ]:
```
import numpy as np
import torch
import matplotlib.pyplot as plt
from IPython.display import clear_output
```

Загрузка данных

Сейчас мы будем использовать встроенные данные, но в реальности приходится писать свой класс для датасета (Dataset), у которого реализовывать несколько обязательных методов (напр, `__getitem__`), но это обсудим уже потом.

In [ ]:
```
import torchvision
from torchvision.datasets import KMNIST


# Превращает картинки в тензоры
transform = torchvision.transforms.Compose(
    [torchvision.transforms.ToTensor()])

# Загрузим данные (в переменных лежат объекты типа `Dataset`)
# В аргумент `transform` мы передаем необходимые трансформации (ToTensor)
trainset = KMNIST(root="./KMNIST", train=True, download=True, transform=transform)
testset = KMNIST(root="./KMNIST", train=False, download=True, transform=transform)

clear_output()
```

Определим даталоадеры, они нужны, чтобы реализовывать стохастический градиентный спуск (то есть мы не хотим считывать в оперативную память все картинки сразу, а делать это батчами).

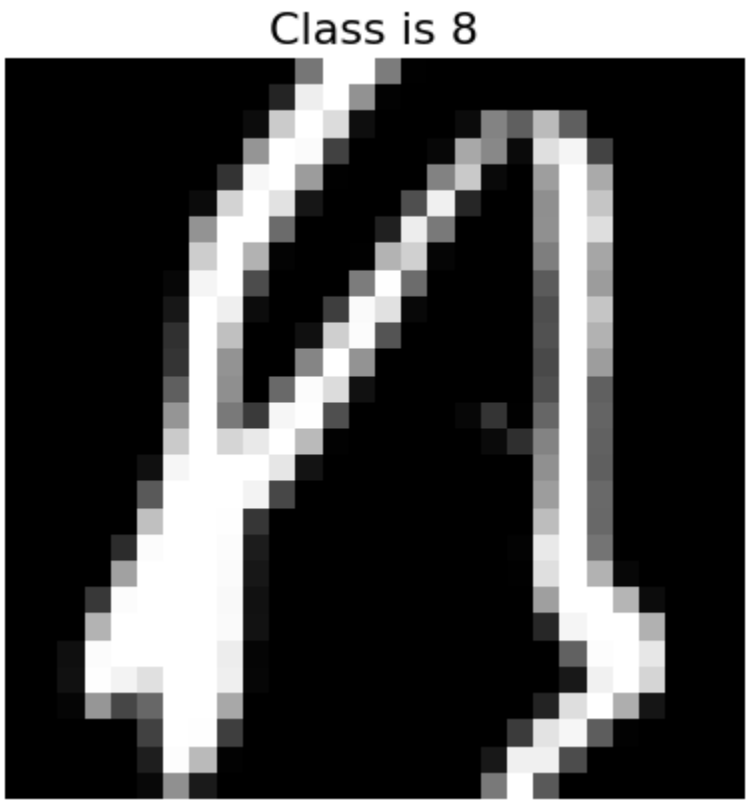
In [ ]:
```
from torch.utils.data import DataLoader


# Можно оставить таким
batch_size = 256

trainloader = DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=2)
testloader = DataLoader(testset, batch_size=batch_size, shuffle=False, num_workers=2)
```

Подумайте, как может влиять на скорость обучения параметр `batch_size`, почему вы так считаете?

Ответ: в целом, чем больше `batch_size` тем быстрее происходит обучение модели, так как за один раз она сможет просмотреть больше данных, но при достижении порога по оперативной памяти, увеличение этого параметра не будет приводить к ускорению обучения, а лишь ухудшит качетсво модели из-за переобучения

Посмотрим на какую-нибудь картинку:

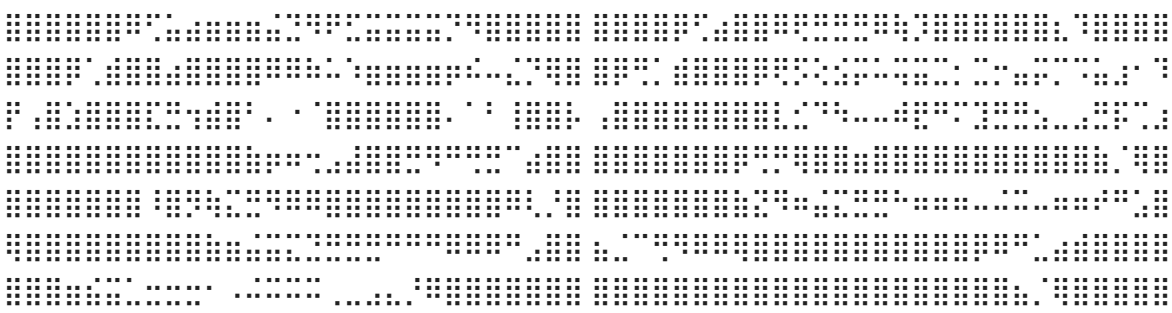
In [ ]:
```
plt.imshow(trainset[0][0].view(28, 28).numpy(), cmap="gray")
plt.axis("off")
plt.title(f"Class is {trainset[0][1]}", fontsize=16);
```



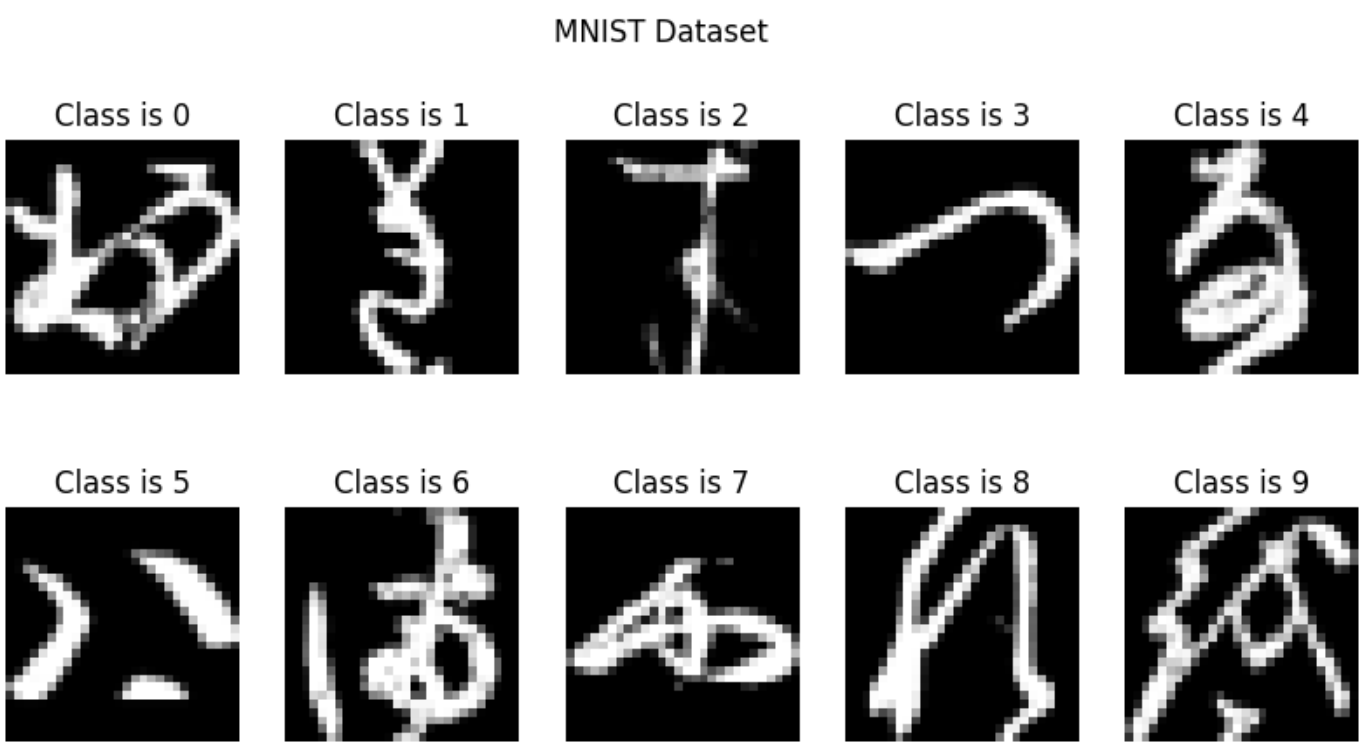
Задание 1. Смотрим на картинки

2 балла

Нарисуйте на одном графике изображения всех 10 классов:



In [ ]:
```
fig, axs = plt.subplots(2, 5, figsize=(10, 5))
fig.suptitle('MNIST Dataset')

for i in range(10):
  k = 0
  while trainset[k][1] != i:
    k+=1
  axs[i//5][i%5].imshow(trainset[k][0].view(28, 28).numpy(), cmap="gray")
  axs[i//5][i%5].axis("off")
  axs[i//5][i%5].set_title(f"Class is {trainset[k][1]}", fontsize=12)
```



Задание 2. Строим свой первый MLP

4 балла

MLP (multilayer perceptron) или нейронная сеть из полносвязных (линейных) слоев, это мы уже знаем.

Опишите структуру сети: 3 полносвязных слоя + функции активации на ваш выбор. **Подумайте** про активацию после последнего слоя!

Сеть на выходе 1 слоя должна иметь 256 признаков, на выходе из 2 128 признаков, на выходе из последнего столько, сколько у вас классов.

https://pytorch.org/docs/stable/nn.html?highlight=activation#non-linear-activations-weighted-sum-nonlinearity

In [ ]:
```python
import torch.nn as nn
import torch.nn.functional as F


class FCNet(nn.Module):
    def __init__(self):
        super().__init__() #  это надо помнить!
        self.fc1 = nn.Linear(in_features=28*28, out_features=256)
        self.fc2 = nn.Linear(in_features=256, out_features=128)
        self.fc3 = nn.Linear(in_features=128, out_features=10)


    def forward(self, x):  # Forward вызывается внутри метода __call__ родительского кла
        ## x -> тензор размерности (BATCH_SIZE, N_CHANNELS, WIDTH, HEIGHT)
        x = x.view(-1, 28*28)
        ## надо подумать над тем, что у нас полносвязные слои принимают векторы

        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        logits = self.fc3(x)
```

```
        return logits
```

```
    # This is formatted as code
```

Сколько обучаемых параметров у вашей модели (весов и смещений)?

Ответ: Первый слой: (28*28*256+256) = 200960 весов+смещений, второй - 256*128 + 128 = 32896, *третий* - 128*10 + 10 = 1674. Таким образом - 203530 обучаемых параметра

In [ ]:
```
```

Задание 3. Напишите код для обучения модели

5 баллов

Можно (и нужно) подглядывать в код семинара по пайторчу. Вам нужно создать модель, определить функцию потерь и оптимизатор (начнем с `SGD`). Дальше нужно обучать модель, при помощи тренировочного `Dataloader'a` и считать лосс на тренировочном и тестовом `Dataloader'ax`.

Напишем функцию для рассчета `accuracy`:

In [ ]:
```
def get_accuracy(model, dataloader):
    """
    model - обученная нейронная сеть
    dataloader - даталоадер, на котором вы хотите посчитать accuracy
    """
    correct = 0
    total = 0
    with torch.no_grad(): # Тензоры внутри этого блока будут иметь requires_grad=False
        for images, labels in dataloader:
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = correct / total

    return accuracy
```

Основной цикл обучения

Этот код можно (и зачастую нужно) выносить в отдельную функцию, но пока что можете это не делать, все по желанию)

In [ ]:
```
# Создадим объект модели
fc_net = FCNet()
# Определим функцию потерь
loss_function = nn.CrossEntropyLoss()
# Создадим оптимизатор для нашей сети
lr = 0.001 # скорость обучения
optimizer = torch.optim.Adam(fc_net.parameters(), lr=3e-4)
```

Напишите цикл обучения. Для начала хватит 10 эпох. Какое значение `accuracy` на тестовой выборке удалось получить?

In [ ]:
```
%%time
```

```
n_epochs = 10
loss_history = []

for epoch in range(n_epochs):
    epoch_loss = 0
    for images, labels in trainloader: # Получаем батч тренировочных картинок
        optimizer.zero_grad() # чтобы не было как в лог регрессии, когда мы не обнуляли
        outputs = fc_net(images) # делаем предсказания
        loss = loss_function(outputs, labels) # считаем лосс
        loss.backward() # считаем градиенты
        optimizer.step() # делаем шаг градиентного спуска

        epoch_loss += loss.item()

    loss_history.append(epoch_loss/len(trainloader))

    print(f"Epoch={epoch+1} loss={loss_history[epoch]:.4f}")
```

```
Epoch=1 loss=0.6880
Epoch=2 loss=0.3042
Epoch=3 loss=0.2256
Epoch=4 loss=0.1768
Epoch=5 loss=0.1423
Epoch=6 loss=0.1165
Epoch=7 loss=0.0950
Epoch=8 loss=0.0772
Epoch=9 loss=0.0635
Epoch=10 loss=0.0518
CPU times: user 30.3 s, sys: 3.23 s, total: 33.5 s
Wall time: 2min 38s
```

In [ ]:
```
get_accuracy(fc_net, testloader)
```

Out[ ]:
```
0.8952
```

Задание 4. Изучение влияния нормализации

3 балла

Вы могли заметить, что мы забыли провести нормализацию наших данных, а для нейронных сетей это может быть очень критично.

Нормализуйте данные.

- Подсчитайте среднее значение и стандартное отклонение интенсивности пикселей для всех тренировочных данных
- Нормализуйте данные с использованием этих параметров (используйте трансформацию `Normalize`)

Оцените влияние нормировки данных.

In [ ]:
```
mean = (trainset.data.view(-1, 28*28*60000)/255).mean()
std = (trainset.data.view(-1, 28*28*60000)/255).std()
print(mean, std)

transform_with_norm = torchvision.transforms.Compose([
                torchvision.transforms.ToTensor(),
                torchvision.transforms.Normalize(mean, std)
            ])
```

```
trainset.transform = transform_with_norm
testset.transform = transform_with_norm
```

```
tensor(0.1918) tensor(0.3483)
```

In [ ]:
```
fc_net = FCNet()
loss_function = nn.CrossEntropyLoss()
lr = 0.001
optimizer = torch.optim.Adam(fc_net.parameters(), lr=3e-4)
```

In [ ]:
```
n_epochs = 10
loss_history = []

for epoch in range(n_epochs):
    epoch_loss = 0
    for images, labels in trainloader: # Получаем батч тренировочных картинок
        optimizer.zero_grad() # чтобы не было как в лог регрессии, когда мы не обнуляли
        outputs = fc_net(images) # делаем предсказания
        loss = loss_function(outputs, labels) # считаем лосс
        loss.backward() # считаем градиенты
        optimizer.step() # делаем шаг градиентного спуска

        epoch_loss += loss.item()

    loss_history.append(epoch_loss/len(trainloader))

    print(f"Epoch={epoch+1} loss={loss_history[epoch]:.4f}")
```

```
Epoch=1 loss=0.6865
Epoch=2 loss=0.3055
Epoch=3 loss=0.2237
Epoch=4 loss=0.1745
Epoch=5 loss=0.1401
Epoch=6 loss=0.1131
Epoch=7 loss=0.0931
Epoch=8 loss=0.0773
Epoch=9 loss=0.0620
Epoch=10 loss=0.0506
```

In [ ]:
```
get_accuracy(fc_net, testloader)
```

Out[ ]:
```
0.899
```

Как изменилась `accuracy` после нормализации?

Accuracy стала чуть больше

Задание 5. Изучение влияния функции активации

3 балла

Исследуйте влияние функций активации на скорость обучения и точность предсказаний модели.

Используйте три функции:

- Sigmoid
- GELU
- Tanh

In [ ]:
```
class FCNet_sigm(nn.Module):
    def __init__(self):
```

```
        super().__init__() #  это надо помнить!
        self.fc1 = nn.Linear(in_features=28*28, out_features=256)
        self.fc2 = nn.Linear(in_features=256, out_features=128)
        self.fc3 = nn.Linear(in_features=128, out_features=10)


    def forward(self, x):  # Forward вызывается внутри метода __call__ родительского кла
        ## x -> тензор размерности (BATCH_SIZE, N_CHANNELS, WIDTH, HEIGHT)
        x = x.view(-1, 28*28)
        ## надо подумать над тем, что у нас полносвязные слои принимают векторы

        x = self.fc1(x)
        x = F.sigmoid(x)
        x = self.fc2(x)
        x = F.sigmoid(x)
        logits = self.fc3(x)

        return logits

class FCNet_gelu(nn.Module):
    def __init__(self):
        super().__init__() #  это надо помнить!
        self.fc1 = nn.Linear(in_features=28*28, out_features=256)
        self.fc2 = nn.Linear(in_features=256, out_features=128)
        self.fc3 = nn.Linear(in_features=128, out_features=10)


    def forward(self, x):  # Forward вызывается внутри метода __call__ родительского кла
        ## x -> тензор размерности (BATCH_SIZE, N_CHANNELS, WIDTH, HEIGHT)
        x = x.view(-1, 28*28)
        ## надо подумать над тем, что у нас полносвязные слои принимают векторы

        x = self.fc1(x)
        x = F.gelu(x)
        x = self.fc2(x)
        x = F.gelu(x)
        logits = self.fc3(x)

        return logits


class FCNet_tanh(nn.Module):
    def __init__(self):
        super().__init__() #  это надо помнить!
        self.fc1 = nn.Linear(in_features=28*28, out_features=256)
        self.fc2 = nn.Linear(in_features=256, out_features=128)
        self.fc3 = nn.Linear(in_features=128, out_features=10)


    def forward(self, x):  # Forward вызывается внутри метода __call__ родительского кла
        ## x -> тензор размерности (BATCH_SIZE, N_CHANNELS, WIDTH, HEIGHT)
        x = x.view(-1, 28*28)
        ## надо подумать над тем, что у нас полносвязные слои принимают векторы

        x = self.fc1(x)
        x = F.tanh(x)
        x = self.fc2(x)
        x = F.tanh(x)
        logits = self.fc3(x)

        return logits
```

In [ ]:

```
fc_net = FCNet_sigm()
loss_function = nn.CrossEntropyLoss()
lr = 0.001
```

```
optimizer = torch.optim.Adam(fc_net.parameters(), lr=3e-4)

n_epochs = 10
loss_history = []

for epoch in range(n_epochs):
    epoch_loss = 0
    for images, labels in trainloader: # Получаем батч тренировочных картинок
        optimizer.zero_grad() # чтобы не было как в лог регрессии, когда мы не обнуляли
        outputs = fc_net(images) # делаем предсказания
        loss = loss_function(outputs, labels) # считаем лосс
        loss.backward() # считаем градиенты
        optimizer.step() # делаем шаг градиентного спуска

        epoch_loss += loss.item()

    loss_history.append(epoch_loss/len(trainloader))

    print(f"Epoch={epoch+1} loss={loss_history[epoch]:.4f}")

print('Sigm')
print(get_accuracy(fc_net, testloader))
print('========')

fc_net = FCNet_gelu()
loss_function = nn.CrossEntropyLoss()
lr = 0.001
optimizer = torch.optim.Adam(fc_net.parameters(), lr=3e-4)

n_epochs = 10
loss_history = []

for epoch in range(n_epochs):
    epoch_loss = 0
    for images, labels in trainloader: # Получаем батч тренировочных картинок
        optimizer.zero_grad() # чтобы не было как в лог регрессии, когда мы не обнуляли
        outputs = fc_net(images) # делаем предсказания
        loss = loss_function(outputs, labels) # считаем лосс
        loss.backward() # считаем градиенты
        optimizer.step() # делаем шаг градиентного спуска

        epoch_loss += loss.item()

    loss_history.append(epoch_loss/len(trainloader))

    print(f"Epoch={epoch+1} loss={loss_history[epoch]:.4f}")

print('Gelu')
print(get_accuracy(fc_net, testloader))
print('========')


fc_net = FCNet_tanh()
loss_function = nn.CrossEntropyLoss()
lr = 0.001
optimizer = torch.optim.Adam(fc_net.parameters(), lr=3e-4)

n_epochs = 10
loss_history = []

for epoch in range(n_epochs):
    epoch_loss = 0
    for images, labels in trainloader: # Получаем батч тренировочных картинок
        optimizer.zero_grad() # чтобы не было как в лог регрессии, когда мы не обнуляли
        outputs = fc_net(images) # делаем предсказания
        loss = loss_function(outputs, labels) # считаем лосс
```

```
        loss.backward() # считаем градиенты
        optimizer.step() # делаем шаг градиентного спуска

        epoch_loss += loss.item()

    loss_history.append(epoch_loss/len(trainloader))

    print(f"Epoch={epoch+1} loss={loss_history[epoch]:.4f}")

print('Tanh')
print(get_accuracy(fc_net, testloader))
print('========')
```

```
Epoch=1 loss=1.5356
Epoch=2 loss=0.7564
Epoch=3 loss=0.5496
Epoch=4 loss=0.4456
Epoch=5 loss=0.3760
Epoch=6 loss=0.3220
Epoch=7 loss=0.2790
Epoch=8 loss=0.2440
Epoch=9 loss=0.2144
Epoch=10 loss=0.1886
Sigm
0.857
========
Epoch=1 loss=0.6649
Epoch=2 loss=0.2794
Epoch=3 loss=0.1987
Epoch=4 loss=0.1511
Epoch=5 loss=0.1185
Epoch=6 loss=0.0953
Epoch=7 loss=0.0757
Epoch=8 loss=0.0615
Epoch=9 loss=0.0497
Epoch=10 loss=0.0378
Gelu
0.902
========
Epoch=1 loss=0.7754
Epoch=2 loss=0.3861
Epoch=3 loss=0.2613
Epoch=4 loss=0.1879
Epoch=5 loss=0.1394
Epoch=6 loss=0.1041
Epoch=7 loss=0.0788
Epoch=8 loss=0.0584
Epoch=9 loss=0.0432
Epoch=10 loss=0.0315
Tanh
0.8992
========
```

С использованием какой функции активации удалось досчить наибольшей `accuracy`?

Лучше всего себя показала функция Gelu

Задание 6. Другие оптимизаторы

4 балла

Исследуйте влияние оптимизаторов на скорость обучения и точность предсказаний модели.

Попробуйте следующие:

- Adam
- RMSprop
- Adagrad

Вам нужно снова обучить 3 модели и сравнить их перформанс (функцию активации используйте ту, которая показала себя лучше всего).

In [ ]:
```
import timeit

fc_net = FCNet_gelu()
optimizers = {
    'Adam' :torch.optim.Adam(fc_net.parameters(), lr=3e-4),
    'RMSprop': torch.optim.RMSprop(fc_net.parameters(), lr=3e-4),
    'Adagrad': torch.optim.Adagrad(fc_net.parameters(), lr=3e-4)
    }

results = {optimizer: {} for optimizer in optimizers.keys()}
loss_function = nn.CrossEntropyLoss()

lr = 0.001
n_epochs = 10

for optimizer_name in optimizers.keys():

  optimizer = optimizers[optimizer_name]

  loss_history = []
  start = timeit.default_timer()
  for epoch in range(n_epochs):
      epoch_loss = 0
      for images, labels in trainloader: # Получаем батч тренировочных картинок
          optimizer.zero_grad() # чтобы не было как в лог регрессии, когда мы не обнулял
          outputs = fc_net(images) # делаем предсказания
          loss = loss_function(outputs, labels) # считаем лосс
          loss.backward() # считаем градиенты
          optimizer.step() # делаем шаг градиентного спуска

          epoch_loss += loss.item()

      loss_history.append(epoch_loss/len(trainloader))

      print(f"Epoch={epoch+1} loss={loss_history[epoch]:.4f}")

  end = timeit.default_timer()

  accur = get_accuracy(fc_net, testloader)
  results[optimizer_name] = {'time': end-start, 'accuracy': accur}
```

```
Epoch=1 loss=0.6691
Epoch=2 loss=0.2804
Epoch=3 loss=0.2023
Epoch=4 loss=0.1554
Epoch=5 loss=0.1217
Epoch=6 loss=0.0968
Epoch=7 loss=0.0790
Epoch=8 loss=0.0628
Epoch=9 loss=0.0507
Epoch=10 loss=0.0399
Epoch=1 loss=0.0460
Epoch=2 loss=0.0277
Epoch=3 loss=0.0207
```

```
Epoch=4 loss=0.0178
Epoch=5 loss=0.0130
Epoch=6 loss=0.0124
Epoch=7 loss=0.0078
Epoch=8 loss=0.0064
Epoch=9 loss=0.0086
Epoch=10 loss=0.0069
Epoch=1 loss=0.0017
Epoch=2 loss=0.0011
Epoch=3 loss=0.0010
Epoch=4 loss=0.0009
Epoch=5 loss=0.0009
Epoch=6 loss=0.0009
Epoch=7 loss=0.0008
Epoch=8 loss=0.0008
Epoch=9 loss=0.0008
Epoch=10 loss=0.0007
```

In [ ]:
```
results
```

Out[ ]:
```
{'Adam': {'time': 164.8207710960014, 'accuracy': 0.8983},
 'RMSprop': {'time': 162.1102809139993, 'accuracy': 0.9016},
 'Adagrad': {'time': 164.12448333499924, 'accuracy': 0.9066}}
```

Как видим, самым быстрым оптимизатором оказался RMSprop, а самым точным Adagrad. Adam оказался худшим по обоим параметрам.

Задание 7. Реализация ReLU

4 балла

Самостоятельно реализуйте функцию активации ReLU. Замените в уже обученной модели функцию активации на вашу. Убедитесь что ничего не изменилась.

In [ ]:
```
#переобучим выше модель с обычным ReLU

class CustomReLU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        # YOUR CODE HERE
        # если элемент x < 0, то 0, если >= 0, то x
        x = (x > 0)*x
        return x

cust_relu = CustomReLU()
```

In [ ]:
```
import types
def new_forward(self, x):

  x = x.view(-1, 28*28)
  x = self.fc1(x)
  x = cust_relu(x)
  x = self.fc2(x)
  x = cust_relu(x)
  logits = self.fc3(x)
  return logits


fc_net.forward = types.MethodType(new_forward, fc_net)
FCNet.forward = new_forward
```

In [ ]:
```
get_accuracy(fc_net, testloader)
```

Out[ ]:
```
0.9011
```

Как видим - новая функция исправно работает

Заново обучите модель и проверьте правильность реализации `CustomReLU`.

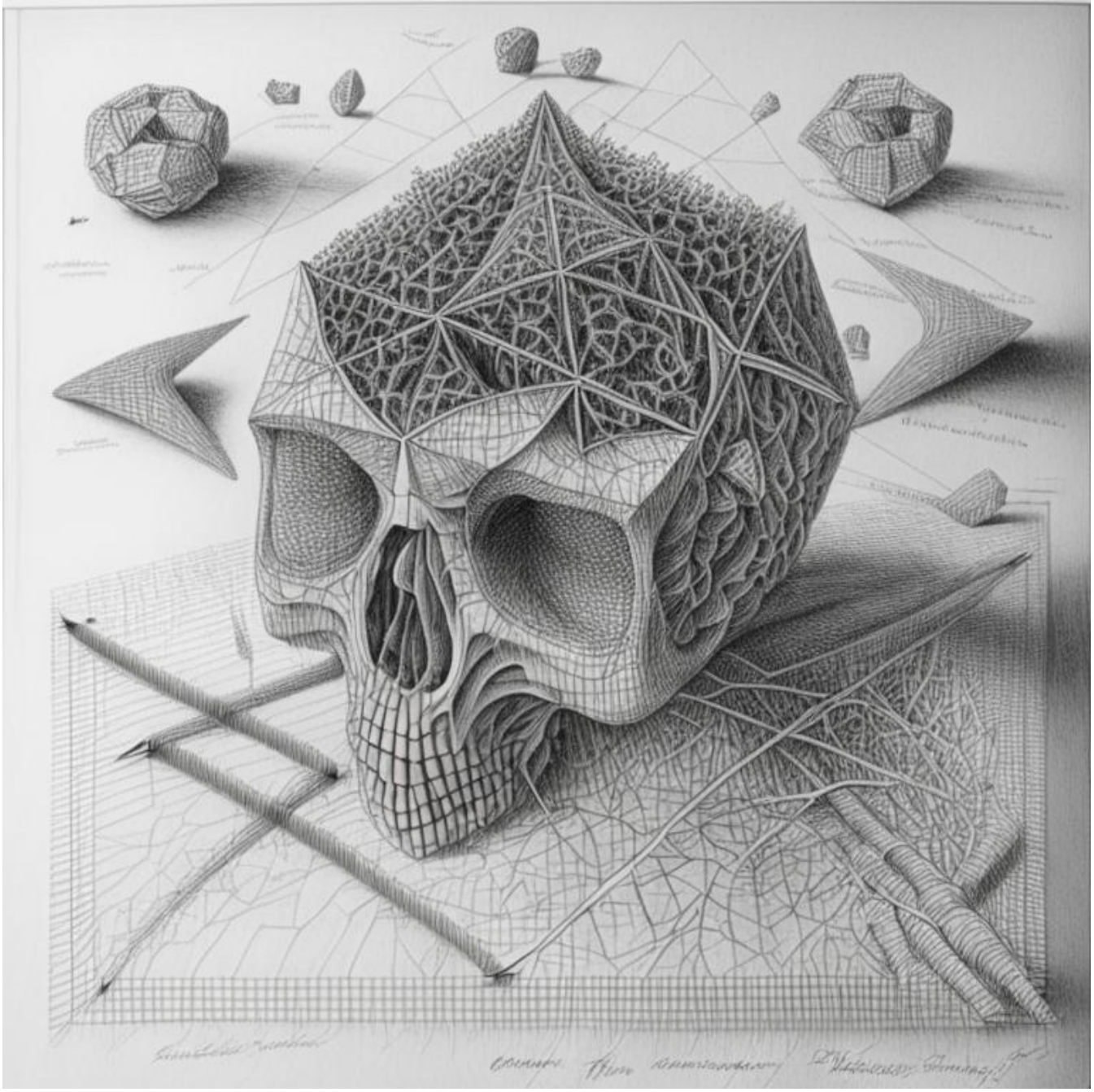
In [ ]:
```
fc_net_new = FCNet()
loss_function = nn.CrossEntropyLoss()
lr = 0.001
optimizer = torch.optim.Adam(fc_net_new.parameters(), lr=3e-4)


n_epochs = 10
loss_history = []

for epoch in range(n_epochs):
    epoch_loss = 0
    for images, labels in trainloader: # Получаем батч тренировочных картинок
        optimizer.zero_grad() # чтобы не было как в лог регрессии, когда мы не обнуляли
        outputs = fc_net_new(images) # делаем предсказания
        loss = loss_function(outputs, labels) # считаем лосс
        loss.backward() # считаем градиенты
        optimizer.step() # делаем шаг градиентного спуска

        epoch_loss += loss.item()

    loss_history.append(epoch_loss/len(trainloader))

    print(f"Epoch={epoch+1} loss={loss_history[epoch]:.4f}")
```

```
Epoch=1 loss=0.6861
Epoch=2 loss=0.3076
Epoch=3 loss=0.2278
Epoch=4 loss=0.1769
Epoch=5 loss=0.1420
Epoch=6 loss=0.1170
Epoch=7 loss=0.0951
Epoch=8 loss=0.0785
Epoch=9 loss=0.0640
Epoch=10 loss=0.0516
```

Задание 8. Генерация картинок

3 балла

Так как вы снова работаете в командах, то придумайте 3 предложения и сгенерируйте при помощи них 3 картинки, используя телеграм бота ruDALLE. Прикрепите сюда ваши картины.



Смерть от топологической статистики

Катманду на закате

Светлячки в стиле khokhloma

In [ ]: