# ТЕХНОЛОГИЧЕСКАЯ КАРТА ЗАНЯТИЯ

**Тема занятия:** Введение в библиотеку PyTorch.

**Аннотация к занятию:** на данном уроке обучающиеся продолжат знакомство с библиотекой PyTorch. Научатся создавать нейросеть. Обсудят каждый шаг цикла обучения нейросети^ от инициализации оптимизаторов до финальных метрик качества и визуализации прогнозов.

**Цель занятия:** сформировать у обучающихся представление о создании нейросети. Рассмотреть каждый цикл обучения нейросети от инициализации оптимизаторов до финальных метрик качества и визуализации прогнозов.

**Задачи занятия:**

- познакомить обучающихся с созданием нейросети;
- рассмотреть каждый шаг цикла обучения нейросети;
- познакомить с понятиями «оптимизатор» и «функция потерь»;
- сравнить метрики качества;
- визуализировать метрики качества;
- применить полученные знания на практике.

## Ход занятия

| Этап занятия | Время | Деятельность педагога | Комментарии, рекомендации для педагогов |
|---|---|---|---|
| **Организационный этап** | 2 мин. | Добрый день! Мы продолжаем работать с библиотекой PyTorch и с её помощью будем строить конвейер с использованием нейросетей. | Приветствие. Создание в классе атмосферы психологического комфорта |
| **Постановка цели и задач занятия. Мотивация учебной деятельности обучающихся** | 10 мин. | **Вопрос для обсуждения**<br>Как создать полноценную нейросеть?<br><br>**Возможные ответы обучающихся**<br>На занятии мы рассмотрим, как создаются нейросети, что такое оптимизаторы и функции потерь. Проведём цикл обучения и визуализируем наши прогнозы, сравним метрики качества. | Способствовать обсуждению мотивационных вопросов |
| **Изучение нового материала** | 50 мин. | Импортируем библиотеки для дальнейшей работы. | Для справки:<br>Файл для работы:<br><br>https://drive.google.com/file/d/1hD-DpZzDmlWGtGD384TrXl3HJp3rqd0W/view?usp=sharing |

```
1  import numpy as np
2  import pandas as pd
3  import matplotlib.pyplot as plt
4  import seaborn as sns
5
6  import torch
7  import torch.nn as nn
```

Данные и подготовка

В качестве данных будем использовать датасет рукописных символов 0-9 MNIST.
https://www.kaggle.com/c/digit-recognizer

Данные были предварительно скачаны с Kaggle и загружены на облачный сервер. Первая колонка данных — разметка числа от 0 до 9. Остальные 784 колонки — вектор, который был сформирован из картинки размером 28 на 28.

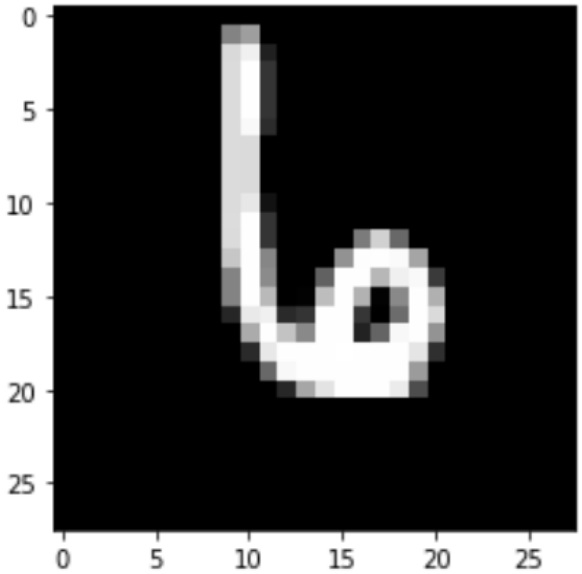
```
[ ]  1  train = pd.read_csv('https://dl.uploadgram.me/6171ac48b6053h?raw')
```

```
1  train
```

| | label | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **41995** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **41996** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Возьмём произвольный пример (123) и превратим это в матрицу, отобразим с помощью imshow. Видно изображение шестёрки.

```
[ ]  1  pixels = np.array(train.iloc[123])
     2  pixels = pixels[1:]
     3  pixels = pixels.reshape((28,28))
     4
     5  plt.imshow(pixels, cmap='gray');
```




Каждое изображение — набор целых чисел, яркостей пикселей от 0 до 255. Разделив все характеристики на 255, получим нормировку на диапазоне 0–1. Разделим данные на train/test в соотношении 80 на 20.

```
1  X_train = np.array(train.drop('label', 1)) / 255
2  y_train = np.array(train['label'])
3
4  from sklearn.model_selection import train_test_split
5
6  X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size=0
7
8  print('X_train.shape', X_train.shape)
9  print('y_train.shape', y_train.shape)
10 print('X_test.shape', X_test.shape)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: FutureWarning: In a fut
  """Entry point for launching an IPython kernel.
X_train.shape (33600, 784)
y_train.shape (33600,)
X_test.shape (8400, 784)
```

Обучение одного слоя
Рассмотрим пример, в котором каждое входное изображение умножается на 10 нейронов (10 векторов из 784 чисел). Так мы получаем 10 векторов предсказаний (predictions) для каждого изображения. Конвертировав веса нейронов в PyTorch Tensor и передав их в оптимизатор, мы даём оптимизатору доступ к области памяти, где хранятся веса, чтобы он мог получить доступ к рассчитанному функцией backward() градиенту (.grad у весов).

```python
%%time
# Веса всех нейронов
x0 = np.random.normal(size=(784, 10)) / np.sqrt(784)
neurons = torch.tensor(x0, requires_grad=True)

# Конвертируем датасет в читаемый торчом формат
X = torch.tensor(X_train)
y = torch.tensor(y_train)
# SGD
optimizer = torch.optim.SGD(params=[neurons], lr=1)

loss_history = []

# 100 шагов
for i in range(100):
    # считаем произведение матрицы параметров на матрицу (из 10 нейронов) весов
    predictions = X @ neurons
    # Вычисляем loss
    loss = torch.nn.functional.cross_entropy(predictions, y) # функция активации уже включе
    # Вычисляем градиент
    optimizer.zero_grad()
    loss.backward()
    # Делаем шаг
    optimizer.step()
    # логируем loss
    loss_history.append(loss.data.numpy())
```

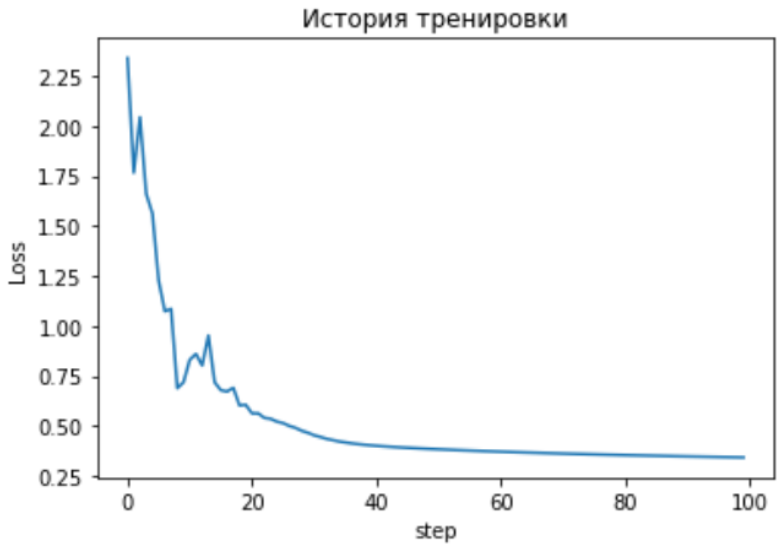
```
CPU times: user 9.83 s, sys: 11.5 ms, total: 9.84 s
Wall time: 9.92 s
```

В конце итерации добавляем рассчитанные значения loss в список для визуализации обучения.

Обучение 10 нейронов на 30 000 примерах заняло всего 15 секунд. Визуализируем график падения loss:

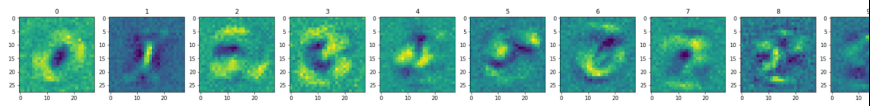
Весь датасет прошли за 15 сек!

```
plt.title('История тренировки')
plt.ylabel('Loss')
plt.xlabel('step')
plt.plot(loss_history);
```




Помимо этого, можно визуализировать веса нейронов, где явно видно очертания цифр:

```
fig, axes = plt.subplots(1, 10, figsize=(30,30))

for i in range(10):
    axes[i].set_title(i)
    neuron_img = neurons[:,i].data.numpy().reshape(28, 28)
    axes[i].imshow(neuron_img)
```



Важно отметить, что 10 нейронов и встроенный в функцию cross-entropy softmax не являются нейросетью, так как скрытых слоёв нет.

Найдём максимально активируемый нейрон (argmax) от вероятностей, полученных на тесте в результате матричного умножения.

```
[ ]
# Тот же код для метрик качества
def sigmoid(x):
    ex = np.exp(x)
    return ex / (1 + ex)

digit_probabilities = sigmoid((X @ neurons).data.numpy())
print('digit_probabilities.shape:', digit_probabilities.shape)
# Находим вероятности на test

from sklearn.metrics import f1_score

# Максимум
predictions = np.argmax(digit_probabilities, axis=1)
print('predictions.shape:', predictions.shape)
print('\nTrain f1:', f1_score(y_train, predictions, average='macro'))
```

```
digit_probabilities.shape: (33600, 10)
predictions.shape: (33600,)

Train f1: 0.9031460282614547
```

| | | | |
|---|---|---|---|
| | | Полученный результат в 10 нейронов даёт хороший результат в 0,9 по f1.<br><br>Обучение полносвязной нейросети<br>Рассмотрим создание полноценной нейросети. В примере выше в качестве слоя из 10 нейронов использовалась матрица размером 784 (входа) на 10 (выходов). На практике мы чаще всего используем готовые решения. Например, полносвязный слой linear подмодуля nn библиотеки PyTorch. В примере комментарием показано, что реализовать слой можно и таким образом. Класс нейросети требует два метода: init (конструктор) и forward (прямой проход / прогноз). В init происходит инициализация слоёв (весов нейросети). Forward обрабатывает пример(ы) с помощью инициализированных слоёв. | |

```python
# Будет использовать высокоуровневый API torch: nn
# Когда мы создаем нейронную сеть в pytorch, мы обычно создаем класс, который происходит от torch.nn.Module
# Это легко обучить и в дальнейшем использовать сеть, определенную таким образом
# Эта сеть имеет 210 нейронов
import torch.nn.functional as F

class MyFirstNN(nn.Module):
    def __init__(self, n_hidden_neurons=200):
        super().__init__()
        # Здесь мы определяем обучаемые параметры модели
        # Первый слой весов
        # init_1 = np.random.normal(size=(784, n_hidden_neurons)) / np.sqrt(784) # гуглите "xavier initialization
        # self.neurons_layer1 = torch.tensor(init_1, requires_grad=True) # веса для первого слоя нейронов
        # self.neurons_layer1 = nn.Parameter(self.neurons_layer1)
        self.neurons_layer1 = nn.Linear(784, n_hidden_neurons)

        # веса для второго слоя нейронов
        # init_2 = np.random.normal(size=(n_hidden_neurons, 10)) / np.sqrt(n_hidden_neurons)
        # self.neurons_layer2 = torch.tensor(init_2, requires_grad=True)
        # self.neurons_layer2 = nn.Parameter(self.neurons_layer2)
        self.neurons_layer2 = nn.Linear(n_hidden_neurons, 10)

    def forward(self, x):
        # Здесь мы делаем все вычисления
        # Первый слой
        # h = x @ self.neurons_layer1
        h = self.neurons_layer1(x)
        # Функция активации скрытого слоя
        # h = torch.relu(h)
        h = F.relu(h)

        # Выходной слой
        # out = h @ self.neurons_layer2
        out = self.neurons_layer2(h)
        return out
```

В примере у нас есть один скрытый слой из 200 нейронов, принимающий на вход картинку (в виде вектора из 784 чисел) и выходной слой, принимающий на вход вектор из 200 значений (после активаций скрытого) и выдающий 10 значений (вероятностное распределение по всем классам). После перемножения скрытый слой использует ReLU-активацию. В выходном слое softmax не используется, так как он уже включён в функции потерь.

Создадим объект нейросети и передадим (по ссылке) её параметры в оптимизатор (чтобы он мог сделать шаг, согласно устройству оптимизатора, по минус градиенту). Тут же задаём скорость обучения.

```
import numpy as np

model = MyFirstNN()
# инициализируем Adam для накопления импульса во время спуска
optimizer = torch.optim.Adam(params=model.parameters(), lr=0.001
```

**Обучение любой нейросети выглядит так:**

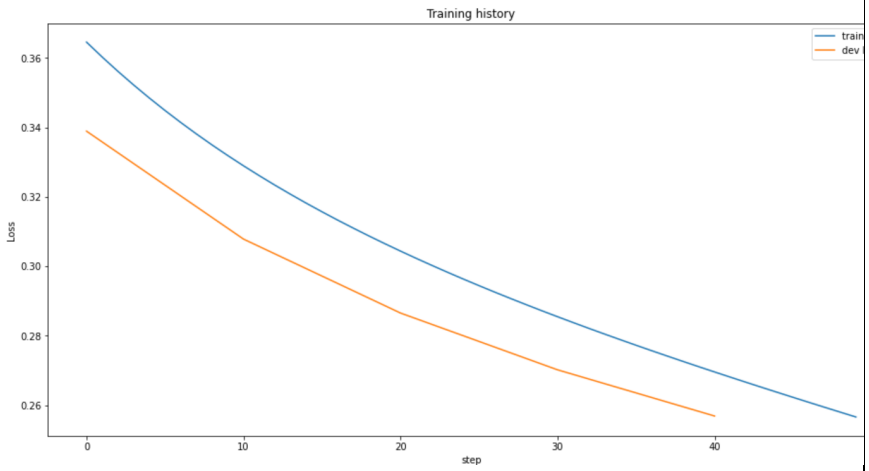
1. Получаем прогноз на наборе данных (или его части, см. batching) model(X_tr).
2. Рассчитываем функцию потерь cross_entropy (прогнозы и реальные значения). Стоит помнить, что в неё уже включён softmax.
3. Обнуляем градиент с прошлого запуска .zero_grad().
4. Рассчитываем градиент на основе обратного распространения ошибки.
5. .step() делает шаг по найденному градиенту (т.к. у оптимизатора есть доступ к весам).
6. Логируем loss.

```python
from tqdm import tqdm

N_EPOCHS = 50

# Конвертируем в торч тензор
X_tr = torch.tensor(X_train[1000:], dtype=torch.float32)
y_tr = torch.tensor(y_train[1000:], dtype=torch.long)
X_dev = torch.tensor(X_train[:1000], dtype=torch.float32)
y_dev = torch.tensor(y_train[:1000], dtype=torch.long)

train_loss_history = []
dev_loss_history = []

# Пройдемся 50 эпох
for i in tqdm(range(N_EPOCHS)):
    # Прямой проход (предсказания)
    predictions = model(X_tr)
    # Функция потерь
    loss = torch.nn.functional.cross_entropy(predictions, y_tr) # активация уже вкл
    # Градиенты
    optimizer.zero_grad()
    loss.backward()
    # Шаг спуска
    optimizer.step()
    train_loss_history.append(loss.item())

    # Валидируемся
    if i % 10 == 0:
        predictions = model(X_dev)
        loss = torch.nn.functional.cross_entropy(predictions, y_dev)
        dev_loss_history.append(loss.item())
```

```
100%|██████████| 50/50 [00:16<00:00,  3.06it/s]
```

Визуализируем loss на train и test:

```
plt.figure(figsize=(16, 8))
plt.title('Training history')
plt.ylabel('Loss')
plt.xlabel('step')
plt.plot(range(N_EPOCHS), train_loss_history, label='train loss')
plt.plot(range(N_EPOCHS)[::10], dev_loss_history, label='dev loss')
plt.legend();
```




Оценим метрики качества, найдя самый сильно активирующийся нейрон (argmax) из прогнозов на тесте. Визуализируем метрики качества, 0.93 по f1.

```python
# Метрики качества
digit_probabilities = model(X_dev).detach().numpy()
print('digit_probabilities.shape:', digit_probabilities.shape)
# Нашли вероятности принадлежности семплов test-а

from sklearn.metrics import classification_report

# Округлили
predictions = np.argmax(digit_probabilities, axis=1)
print('predictions.shape:', predictions.shape)
print('test results:')
print(classification_report(y_dev.detach().cpu().numpy(), predictions))
```

```
digit_probabilities.shape: (1000, 10)
predictions.shape: (1000,)
test results:
              precision    recall  f1-score   support

           0       0.94      0.95      0.95        86
           1       0.99      0.98      0.99       119
           2       0.93      0.93      0.93       103
           3       0.90      0.94      0.92        84
           4       0.93      0.94      0.93       109
           5       0.91      0.89      0.90        94
           6       0.96      0.98      0.97        92
           7       0.94      0.90      0.92       109
           8       0.93      0.88      0.90        96
           9       0.88      0.93      0.90       108

    accuracy                           0.93      1000
   macro avg       0.93      0.93      0.93      1000
weighted avg       0.93      0.93      0.93      1000
```

Рассмотрим пример, в котором мы добавим еще несколько слоёв. Помимо этого добавим .cuda() при инициализация объекта класса нейросети, что инициализирует веса на gpu.

Для справки: в задачах машинного обучения для оценки качества моделей и сравнения различных алгоритмов используются метрики.

```python
from torch.nn import ReLU
import torch.nn.functional as F

class MySecondNN(nn.Module):
    def __init__(self, n_hidden_neurons=200):
        # super позволяеть наследовать методы модуля nn
        super(MySecondNN,self).__init__()
        # создаем линейный слой
        self.linear1 = nn.Linear(784,256)
        self.linear2 = nn.Linear(256,n_hidden_neurons)
        self.linear3 = nn.Linear(n_hidden_neurons,n_hidden_neuron
        self.linear4 = nn.Linear(n_hidden_neurons,10)
        self.ReLU = nn.ReLU(inplace=True)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = F.relu(self.linear3(x))
        x = self.linear4(x)
        return x

import numpy as np

model = MySecondNN().cuda()
# инициализируем Adam для накопления импульса во время спуска
optimizer = torch.optim.Adam(params=model.parameters(), lr=0.001)
```

В дальнейшем обучение будет идти аналогично, кроме того что данные будут инициализироваться на gpu (чтоб

веса и примеры были на одном устройстве):

```
], dtype=torch.float32).cuda()
], dtype=torch.long).cuda()
], dtype=torch.float32).cuda()
], dtype=torch.long).cuda()
```

Классификация изображений
Рассмотрим обучение нейросети на примере картинок датасета cifar10, состоящего из 10 классов. Датасет встроен в библиотеку, его загрузка будет происходить в папку data из встроенной функции datasets.CIFAR10. Во время загрузки можно указать аргумент transform — какие преобразования производить со входными, «сырыми» данными. В нашем случае мы укажем конвертацию в PyTorch Tensor, однако часто тут видно преобразования размера изображения (у нас все изображения исходно одного размера), аугментации данных.

```
[ ]  1  # конвертируем сразу в pytorch tensor
     2  transform_train = transforms.Compose([
     3      transforms.ToTensor()
     4  ])
     5
     6  transform_val = transforms.Compose([
     7      transforms.ToTensor()
     8  ])
```

```
     1  # загрузим и сразу сконвертируем
     2  train_data = datasets.CIFAR10(root="./data", train=True, download=True, transform=transform_tra
     3  test_data = datasets.CIFAR10(root="./data", train=False, download=True, transform=transform_val
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100% 170498071/170498071 [00:13<00:00, 13891953.83it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
```

Переменные train_data и test_data указывают на область памяти, к которой необходимо произвести преобразования, указанные в transform_train. В переменной train_loader уже происходит вызов функции загрузки и разбиения на пачку из 16 примеров (batch_size).

```
[ ]  1  # загрузчики треин и тест части
     2  train_loader = torch.utils.data.DataLoader(train_data, batch_size=16, shuffle = True, pin_memory=True, num_workers=
     3  test_loader = torch.utils.data.DataLoader(test_data, batch_size=16, shuffle = False)
```

```
[ ]  1  train_loader
```

```
<torch.utils.data.dataloader.DataLoader at 0x7f99951d97d0>
```

Помимо этого аргумент shuffle = True указывает на то, что на каждой эпохе перемешка происходит по-разному. Аргумент num_worker указывает на количество потоков, которое будет использовано для динамической подгрузки (и конвертации в PyTorch Tensor) данных во время обучения. pin_memory = True указывает на то, что загрузка

будет происходить сразу на gpu, чтобы не тратить время на перенос данных из оперативной памяти на видеопамять во время обучения.

Для примера можно загрузить один элемент набора — данные и разметку. По форме тензора можно видеть, что пачка (батч) данных — это 16 трёхмерных тензоров (3 канала шириной и высотой по 32). Вектор ответов — 16 чисел.

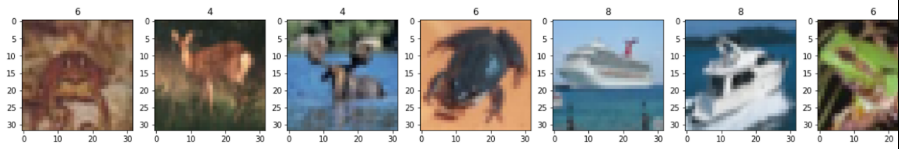
```
[ ]  1  dataiter = iter(train_loader)
     2  # батч картинок и батч ответов к картинкам
     3
     4  images, labels = dataiter.next()
```

```
[ ]  1  # размер датасета
     2  images.shape, labels.shape
```

```
(torch.Size([16, 3, 32, 32]), torch.Size([16]))
```

Помимо этого, можно прооперироваться по выбранному тензору и визуализировать с помощью функции imshow. Функция transpose здесь нужна, так как функция imshow ожидает другой порядок организации массива (не 32, 32, 3 а 3, 32, 32).

```python
# отобразим несколько картинок
def show_imgs(imgs, labels):
    f, axes= plt.subplots(1, 10, figsize=(30,5))
    for i, axis in enumerate(axes):
        # загружаем изображения как тип ndarray (Height * Width * Channels)
        # будьте внимательны при преобразовании dtype в np.uint8 [целое число (от 0 до 255)].
        # в этом примере я не использую метод ToTensor() из torchvision.transforms
        # поэтому вы можете преобразовать форму numpy ndarray в тензор в PyTorch (H, W, C) --> (C, H, W)
        axes[i].imshow(np.squeeze(np.transpose(imgs[i].numpy(), (1, 2, 0))), cmap='gray')
        axes[i].set_title(labels[i].numpy())
    plt.show()

show_imgs(images, labels)
```




Рассмотрим реализацию полносвязной нейросети для примера изображений. Перед тем как подавать её на линейный слой, необходимо «вытянуть» её в вектор. Для этого существует собственный класс Flatten. Мы вызываем его (функция forward класса Flatten) в функции forward класса SimpleNet. Далее вектор проходит через один скрытый (32*32*3 на вход, 256 на выход) слой и классификационный (256 на вход, 10 на выход):

```
[ ]
# класс для удобного перевода картинки из двумерного объекта в вектор
class Flatten(nn.Module):
    def forward(self, input):
        return input.view(input.size(0), -1)

class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = Flatten()
        self.fc1 = nn.Linear(32*32*3, 256) # полносвязные слои - вход и выхо
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        # forward pass сети

        # переводим входной объект из картинки в вектор
        x = self.flatten(x)
        # умножение на матрицу весов 1 слоя и применение функции активации
        x = F.relu(self.fc1(x))
        # умножение на матрицу весов 2 слоя и применение функции активации
        x = self.fc2(x)
        return x
```

Рассмотрим функцию тренировки нейросети. В первую очередь мы принимаем на вход объект класса нейросети и количество эпох обучения. Инициализируем функцию кросс-энтропии и оптимизатор, который принимает параметры алгоритма для дальнейшего шага по минус градиенту и скорость обучения. Помимо этого, инициализируем best_accuracy — лучшую на текущий момент accuracy, чтобы сохранять только те веса нейросети, которые обновили качество на тесте.

```python
def train(net, n_epoch=5):
    # выбираем функцию потерь
    loss_fn = torch.nn.CrossEntropyLoss()

    # выбираем алгоритм оптимизации и learning_rate
    learning_rate = 1e-3
    optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)

    # acc по test
    best_accuracy = 0
    # обучаем сеть 2 эпохи
```

Далее проитерируемся по эпохам и будем логировать текущий loss на train. Итерируясь по загрузчику, мы будем распаковывать обучающие пачки (батчи) примеров, которые далее распаковываются в признаки (X_batch) и ответы (y_batch). Затем обновляются градиенты с прошлых запусков (zero_grad). После этого получаем вектор прогнозов y_pred и с учётом ответов y_batch сравниваем их с помощью функции потерь. Вызывая backward, мы рассчитываем градиенты с учётом текущих весов и ошибок от текущей пачки. Далее, с учётом устройства алгоритма оптимизации, делаем шаг по (минус) антиградиенту. Далее происходит логирование и визуализация текущего loss.

```python
for epoch in tqdm(range(n_epoch)):

    running_loss = 0.0
    train_dataiter = iter(train_loader)
    for i, batch in enumerate(tqdm(train_dataiter)):
        # так получаем текущий батч
        X_batch, y_batch = batch

        # обнуляем градиент
        optimizer.zero_grad()

        # forward pass (получение ответов на батч картинок)
        y_pred = net(X_batch)
        # вычисление лосса от выданных сетью ответов и правильных ответов на ба
        loss = loss_fn(y_pred, y_batch)
        # backpropagation (вычисление градиентов)
        loss.backward()
        # обновление весов сети
        optimizer.step()

        # выведем текущий loss
        running_loss += loss.item()
        # выведем качество каждые 500 батчей
        if i % 500 == 499:
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 500))
            running_loss = 0.0
```

В конце эпохи происходит итерация по батчам теста без добавления градиентов. Далее происходит перенос результатов обратно с gpu на оперативную память для расчёта accuracy по пачке. Результаты accuracy всех пачек усредняются. Если она выросла по сравнению с прошлой эпохой, сохраняем модель.

```
# менеджер упарвления контекстом торча указывает на то, чтобы не обновлять параметры
with torch.no_grad():
    accuracy = []
    for batch in test_loader:
        x, y = batch
        y_pred = net(x)
        # loss = loss_fn(y_pred, y)
        # находим accuracy батча с теста
        accuracy.append(accuracy_score(y.detach().numpy(), np.argmax(y_pred.detach().numpy(), axis=
    # усредняем accuracy всех батчей на тесте
    accuracy = np.mean(np.array(accuracy))
    # если стало лучше - сохраняем на диск и обновляем лучшую метрику
    if accuracy > best_accuracy:
        print('New best model with test acc:', accuracy)
        torch.save(net.state_dict(), './best_model.pt')
        best_accuracy = accuracy
```

Результат обученной нейросети возвращается в функции. Осталось только инициализировать сеть и подать её функцию обучения.

```
[ ]  1  # объявляем сеть
     2  net = SimpleNet()
     3  # теперь обучить сеть м
     4  net = train(net)
```

После обучения можно перепроверить результат на батчах теста. Результат 0,44 можно улучшить путём добавления слоёв и увеличения количества эпох.

```
with torch.no_grad():
    accuracy = []
    for batch in test_loader:
        x, y = batch
        y_pred = net(x)
        accuracy.append(accuracy_score(y.numpy(), np.argmax(y_pred.detach().numpy(), axis=
    accuracy = np.mean(np.array(accuracy))

print('accuracy', accuracy)
```

```
accuracy 0.4424
```

Обучение на gpu происходит аналогично, только требует явного переноса данных на gpu в функции train:

```
# переносим его на видеопамять
# если точно уверены, что это г
X_batch = X_batch.to(device)
y_batch = y_batch.to(device)
# обнуляем веса
```

```
with torch.no_grad():
    accuracy = []
    for batch in test_loader:
        x, y = batch
        y_pred = model(x.to(device))
        accuracy.append(accuracy_score(y.numpy(), np.argmax(y_pred.detach().cpu().numpy(), axis=1
    accuracy = np.mean(np.array(accuracy))

print('accuracy', accuracy)
```

```
accuracy 0.4506
```

Перепроверим качество — получаем небольшой прирост accuracy. На практике обработка изображений чаще всего

| | | | |
|---|---|---|---|
| | | происходит с помощью свёрточных нейросетей, где результаты можно довести до 0,9. | |
| **Закрепление изученного материала** | 15 мин. | **Вопросы для обсуждения**<br>● Как создать нейросеть?<br>● С помощью каких функций можно обучить нейросеть? | Педагог организует беседу по вопросам |
| **Этап подведения итогов занятия (рефлексия)** | 8 мин. | **Вопросы для обсуждения**<br>● Чему я научился?<br>● С какими трудностями я столкнулся?<br>● Какие вопросы остались? Что осталось непонятным? | Педагог способствует размышлению обучающихся над вопросами |
| **Информация о домашнем задании, инструктаж по его применению** | 5 мин. | - | |

**Рекомендуемые ресурсы для дополнительного изучения:**


1. PyTorch. [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/334380/
2. Глубокое обучение, тонкая настройка нейронной сети. [Электронный ресурс] – Режим доступа: https://habr.com/ru/company/wunderfund/blog/315476/
3. Регуляризация. [Электронный ресурс] – Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Регуляризация
4. Batch normalization для ускорения обучения нейронных сетей. [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/309302/