ТЕХНОЛОГИЧЕСКАЯ КАРТА ЗАНЯТИЯ

Тема занятия: Введение в библиотеку PyTorch.

Аннотация к занятию: на данном уроке обучающиеся продолжат знакомство с библиотекой PyTorch и обсудят, как создаются кастомизированные загрузчики данных.

Цель занятия: сформировать у обучающихся представление о кастомизированных загрузчиках данных.

Задачи занятия:

- познакомить с кастомизированными загрузчиками данных;
- рассмотреть модули библиотеки PyTorch;
- применить полученные знания на практике.

Ход занятия

Этап занятия	Время	Деятельность педагога	Комментарии, рекомендации для педагогов
Организационный этап	2 мин.	Здравствуйте! Продолжаем знакомство с библиотекой PyTorch	Приветствие. Создание в классе атмосферы психологического комфорта
Постановка цели и задач занятия. Мотивация учебной деятельности обучающихся	10 мин.	На этом занятии мы обсудим, как создаются кастомизированные загрузчики данных, такие, которые подходят именно вашим данным, и их особенности. Посмотрим на концепцию, в которой вы создаёте нейросетевые блоки, из которых можно создать глубокую нейросеть. Давайте приступать	
Изучение нового материала	50 мин.	Кастомные загрузчики и готовые реализации На этом занятии мы рассмотрим, как загружать собственные данные из пачки (батчи) на примере датасета «Коты против собак». Датасет предполагает бинарную классификацию изображений котов и собак. Загрузим его с помощью команды wget и распакуем в рабочую директорию.	Для справки: Файл для работы: https://drive.google.com/file/d/1pjfTdwRURTVFQ4_N62KtlfDhQdH184ot/view?usp=sharing .

```
[ ]  1  # команда wget скачивает файлы из интернета по ссылке
     2  ! wget https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip
     3  # команда unzip разархивирует zip-архив
     4  ! unzip cats_and_dogs_filtered.zip
```

Видено, что в распакованной папке присутствуют папки train и validation. В каждой из них есть папки по соответствующим названиям классов с нужными изображениями.

```
[ ]  1  # команда ls выводит список файлов в текущей директории
     2  ! ls
```

```
cats_and_dogs_filtered  cats_and_dogs_filtered.zip  sample_data
```

```
[ ]  1  # команда ls ИМЯ_ПАПКИ выводит список файлов в указанной папке
     2  ! ls cats_and_dogs_filtered/
```

```
train  validation  vectorize.py
```

```
[ ]  1  ! ls cats_and_dogs_filtered/train
```

```
cats  dogs
```

Импортируем библиотеки и функции для загрузки изображений. Поскольку изображения для обучения разного разрешения, мы укажем в функции compose (композиции преобразований) функцию resize до разрешения 224 на 224 и конвертации значений яркости пикселей (224*224 высота на ширину, 3 канала: красный, синий и зеленый) в PyTorch Tensor.

```
[ ]  1  # модули библиотеки PyTorch
     2  import torch
     3  from torchvision import datasets, transforms
     4  # метрика качества
     5  from sklearn.metrics import accuracy_score
```

```
     1  # можно из коробки приводить все изображения к
     2  transform_train = transforms.Compose([
     3          transforms.Resize((224, 224)),
     4          transforms.ToTensor(),
     5  ])
     6  # отдельный для теста
     7  transform_val = transforms.Compose([
     8          transforms.Resize((224, 224)),
     9          transforms.ToTensor(),
    10  ])
```

Функция ImageFolder автоматически индексирует все изображения в папках и разбивает их на классы (в зависимости от названия подпапки, в нашем случае cat и dog). В аргументах мы укажем transform: какие преобразования мы хотим применять к данным в момент вызова функции загрузки.

```
[ ]  1  # при загрузке сразу указываем как предобрабатывать
     2  train_data = datasets.ImageFolder("cats_and_dogs_filtered/train/", transform=transform_train, )
     3  test_data = datasets.ImageFolder("cats_and_dogs_filtered/validation", transform=transform_val)
     4  # там же в transforms с версии торча 1.8+ появилась и аугментация
```

Теперь создадим загрузчик, который будет непосредственно загружать несколько изображений в 2 потока до нужного размера пачки.

```
[ ]  1  # разбиваем сразу на батчи (обычно берут под максимум, который влезает в память)
     2  train_loader = torch.utils.data.DataLoader(train_data, batch_size=32, shuffle = True, pin_memory=True, num_workers=2)
     3  test_loader = torch.utils.data.DataLoader(test_data, batch_size=32, shuffle = False, pin_memory=True, num_workers=2)
```

Для проверки мы загрузим одну пачку и отобразим форму тензора (изображений и разметок).

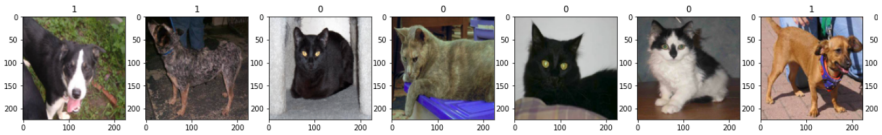
```
[ ]  1  dataiter = iter(train_loader)
     2  # батч картинок и батч ответов к картинкам
     3  images, labels = dataiter.next()
```

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

```
(torch.Size([32, 3, 224, 224]), torch.Size([32]))
```

Помимо этого, визуализируем батч картинок и разметку с помощью функции imshow.

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

# посмотрим на пару из них
def show_imgs(imgs, labels):
    f, axes= plt.subplots(1, 10, figsize=(30,5))
    for i, axis in enumerate(axes):
      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)
```



Класс flatten будет реализовывать «выпрямление» трёхмерного тензора изображения в вектор. В функции train мы инициализируем функцию потерь и оптимизатор, которому доступны параметры алгоритма. Мы итерируемся по эпохам, в каждой из которых итерируемся по батчам train. Распаковывая батч, мы переносим изображения и разметку батча на gpu.

Для справки: функции потерь являются важным компонентом нейронной сети. Взаимодействуя между прямым и обратным переходом в модели глубокого обучения, они эффективно вычисляют, насколько плохо работает модель (насколько велики её потери).

```python
# класс для удобного перевода картинки из 3l объекта в вектор
class Flatten(nn.Module):
    def forward(self, input):
        return input.view(input.size(0), -1)

def train(net, n_epoch=10):
    # 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
    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
            # переносим его на видеопамять
            # если точно уверены, что это гпу - можно написать .cuda()
            X_batch = X_batch.to(device)
            y_batch = y_batch.to(device)
            # обнуляем веса
            optimizer.zero_grad()
```

Получим прогнозы на train. Сравним прогнозы с разметкой с помощью функции потерь. Вычислим с помощью backward градиенты. Сделаем шаг по минус градиенту с

		помощью step. Результаты логируем с помощью .item(). Функция открепляет значения от вычислительного графа (не даёт вычислять по ним градиент при следующих запусках).
		`# forward pass (получение ответов на батч картинок)`
		`y_pred = net(X_batch)`
		`# вычисление лосса от выданных сетью ответов и правильных ответов на батч`
		`loss = loss_fn(y_pred, y_batch)`
		`# bsckpropagation (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 рассчитаем метрики качества на батчах теста. Для этого как прогнозы, так и разметку необходимо перенести с gpu на оперативную память (с помощью .cpu()), открепить от вычислительного графа (detach) и конвертировать в NumPy.
			Для справки: в задачах машинного обучения для оценки качества моделей и сравнения различных алгоритмов используются метрики.

```python
# менеджер упарвления контекстом торча указывает на то, чтобы не обновлять параметры
with torch.no_grad():
    accuracy = []
    for batch in test_loader:
        x, y = batch
        # переносим на гпу
        x = x.to(device)
        y = y.to(device)
        # прогнозируем
        y_pred = net(x)
        # loss = loss_fn(y_pred, y)
        # находим accuracy батча с теста
        accuracy.append(accuracy_score(y.detach().cpu().numpy(), np.argmax(y_pred.detach().cpu().numpy(), axis=1)))
    # 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
```

Для примера мы инициализируем достаточно крупную, многослойную модель с помощью функционала sequentional, который позволяет просто перечислить слои и функции активации в порядке обработки тензора. Будем использовать dropout (случайное отключение нейронов с указанной вероятностью для стабилизации обучения).

Очистим кэш, хранящийся на gpu, и соберём мусор в оперативной памяти с помощью gc.collect().

```python
model = nn.Sequential(
    Flatten(),
    nn.Linear(3*224*224, 102
    nn.ReLU(),
    nn.Dropout(p=0.3),
    nn.Linear(1024, 512),
    nn.ReLU(),
    nn.Linear(512, 512),
    nn.ReLU(),
    nn.Linear(512, 256),
    nn.ReLU(),
    nn.Linear(256, 256),
    nn.ReLU(),
    nn.Linear(256, 256),
    nn.ReLU(),
    nn.Linear(256, 2)
).to(device)
```

```python
torch.cuda.empty_cache()
import gc
gc.collect()
```

```
543
```

Вызовем функцию обучения модели:

```
model = train(model)
```

В конце обучения проитерируемся по тестовым данным для получения метрик качества. Доля верных ответов (accuracy) показывает, что модель ничем не лучше случайного гадания. Это обусловлено сложной структурой изображений реального мира и высоким разрешением (размер входного вектора — 3*224*224) входного вектора. Линейные слои в данном случае не подходят для такого типа данных, что может быть исправлено, например, сверточными нейросетями.

```
labels_net = []
labels_true = []
for images, labels in test_loader:
    labels_true.extend(labels)
    labels_net.extend(model.forward(images.cuda()).detach().cpu().numpy())
```

+ Код + Текст

```
accuracy_score(labels_true, np.argmax(np.array(labels_net), axis=1))
```

```
0.5
```

Кастомные загрузчики данных (даталоадеры)

Вопрос для обсуждения

		Что делать, если структура набора данных отличается от датасета not mnist? **Ответы обучающихся** Для этого рассмотрим, как написать свой собственный загрузчик. Функция load_notmnist занимается загрузкой датасета notmnist, состоящего из изображений букв с разными шрифтами. Функция загружает данные в виде NumPy-массива и сохраняет их на диск. Рассмотрим вид кастомизированного загрузчика. Он включает в себя 3 функции: конструктор (init), оценку размера данных (len), получение примера по индексу (getitem). Обычно в init мы инициализируем пути к файлам (например, с помощью Pandas-датафреймов) или напрямую сам датасет с разметкой, как в этом примере. Такой вариант работает в случае, если его загруженный размер меньше, чем доступная оперативная память. Обычно в случае датасетов из глубокого обучения нам необходимы десятки тысяч примеров, и лучше инициализировать только пути к файлам, которые getitem будет динамически подгружать по номеру (индексу index) во время обучения. Обычно в функции len мы указываем размер датафрейма, инициализированного в init.	

```python
# imageFolder не сможет нормально обработать выборки, которые не помещаются в оперативку
class DatasetMNIST(Dataset):
    def __init__(self, path='./notMNIST_small', letters='ABCDEFGHIJ', transform=None):
        # мы можем использовать imageFolder для загрузки какой-то части датасета
        self.data, self.labels, _ ,_  = load_notmnist(path=path, letters=letters, test_size=0)
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        image = self.data[index].transpose(1, 2, 0) # формат каналов торча
        label = self.labels[index]

        if self.transform is not None:
            image = self.transform(image)

        return image, label
```

Функция getitem подгружает данные по индексу (обычно определённому в конструкторе). В примере видно, что наш датасет уже загружен в NumPy-массив и мы используем индексацию. На практике можно использовать любые сторонние библиотеки для загрузки (это касается не только типа данных изображений, для которых часто можно увидеть cv2 или skimage), которые эффективнее всего работают именно с вашим типом данных. Помимо загрузки данных, можно произвести различные аугментации (или композиции аугментаций) для обогащения набора данных, например, повернутыми изображениями, увеличенными или уменьшенными.

Создадим объект класса датасета, загрузив только 2 класса (буквы A и B). Для примера вызовем один раз получение примера (getitem). Форма загруженного тензора 28 на 28 на 1 — одно чёрно-белое изображение. Полученный

результат — это NumPy-массив, которые необходимо преобразовать в PyTorch Tensor.

```
[ ]  1  full_dataset = DatasetMNIST('./notMNIST_small', 'AB', transform=None)
```

```
Parsing...
found broken img: ./notMNIST_small/A/RGVtb2NyYXRpY2FCb2xkT2xkc3R5bGUgQm9sZC5
Done
```

```
[ ]  1  # встроенный метод получает нужнуй элемент по индексу
     2  img, lab = full_dataset.__getitem__(0)
     3
     4  print(img.shape)
     5  print(type(img))
```

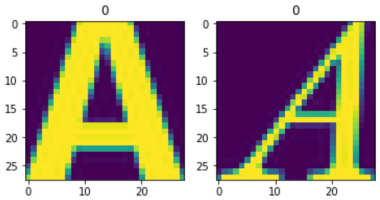
```
(28, 28, 1)
<class 'numpy.ndarray'>
```

```
     1  a = torchvision.transforms.ToTensor()
     2
     3  a(img).shape
```

```
torch.Size([1, 28, 28])
```

Для примера мы визуализируем полученные изображения с помощью imshow.

```
[ ]  1  # отобразим пару букв
     2  inds = np.random.randint(len(full_dataset), size=2)
     3
     4  for i in range(2):
     5      plt.subplot(1, 2, i + 1)
     6      plt.imshow(full_dataset[inds[i]][0].reshape([28,28]))
     7      plt.title(str(full_dataset[inds[i]][1]))
```




```
[ ]  1  # создаем привычный загрузчик
     2  train_loader = DataLoader(full_dataset, batch_size=8, shuffle=True, pin_memory=True, num_workers=2)
```

Затем можно подать полученный объект класса в класс DataLoader библиотеки PyTorch. Именно с его помощью можно реализовать асинхронную загрузку (в нашем случае в 2 потока) данных на gpu (pin_memory) сразу батчами (в нашем случае по 8 картинок). Отобразив тип данных класса, мы видим MultiProcessingDataLoaderIter — асинхронный загрузчик. Для примера вызовем одну итерацию с помощью next(). Видим, что в пачке 8 чёрно-белых изображений формы 28 на 28 на 1 и вектор из 8 ответов.

```
1
2  # мы можем использовать dataloader в качестве итератора с помощь
3  train_iter = iter(train_loader)
4  print(type(train_iter))
```

```
<class 'torch.utils.data.dataloader._MultiProcessingDataLoaderIter'>
```

```
[ ]  1  # мы можем просматривать изображения и метки пакетного размера,
     2  images, labels = train_iter.next()
     3
     4  print('images shape on batch size = {}'.format(images.size()))
     5  print('labels shape on batch size = {}'.format(labels.size()))
```

```
images shape on batch size = torch.Size([8, 28, 28, 1])
labels shape on batch size = torch.Size([8])
```

Помимо этого в аугментациях (transform при инициализации конструктора класса) мы можем указать уже знакомый нам compose из набора преобразований или непосредственно функцию, которая нам нужна, например, конвертации в PyTorch Tensor.

```python
# конвертируем в тензор
train_dataset_with_transform = DatasetMNIST(
    transform=torchvision.transforms.ToTensor()
)

# например обрезать по центру
# transforms.Compose([
#     transforms.CenterCrop(10),
#     # https://github.com/albumentations-team/albumentations
#     transforms.ToTensor(),
# ])
```

```
Parsing...
found broken img: ./notMNIST_small/F/Q3Jvc3NvdmVyIEJvbGRPYmxpcXVlLr
found broken img: ./notMNIST_small/A/RGVtb2NyYXRpY2FCb2xkT2xkc3R5bG
Done
```

```python
img, lab = train_dataset_with_transform.__getitem__(0)

print('image shape at the first row : {}'.format(img.size()))
```

```
image shape at the first row : torch.Size([1, 28, 28])
```

После инициализации такого загрузчика тип данных станет PyTorch Tensor а не NumPy.
Подадим его в загрузчик и визуализируем пачку данных.

```
[ ]  1  train_loader_tr = DataLoader(train_dataset_with_transform, batch_size=8, shuffle=True)
     2
     3  train_iter_tr = iter(train_loader_tr)
     4  print(type(train_iter_tr))
     5
     6  images, labels = train_iter_tr.next()
     7
     8  print('images shape on batch size = {}'.format(images.size()))
     9  print('labels shape on batch size = {}'.format(labels.size()))
```

```
<class 'torch.utils.data.dataloader._SingleProcessDataLoaderIter'>
images shape on batch size = torch.Size([8, 1, 28, 28])
labels shape on batch size = torch.Size([8])
```

```
1  grid = torchvision.utils.make_grid(images)
2
3  plt.imshow(grid.numpy().transpose((1, 2, 0)))
4  plt.axis('off')
5  plt.title(labels.numpy());
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([6
/usr/local/lib/python3.7/dist-packages/matplotlib/text.py:1165: FutureWarning: elementwise c
  if s != self._text:
```

[1 5 0 6 3 9 9 7]

Помимо конвертации в PyTorch Tensor можно указать и compose — композицию преобразований, включающую «выпрямление» изображения в вектор.

```python
class Flatten():
    def __call__(self, pic):
        return pic.flatten()

    def __repr__(self):
        return self.__class__.__name__ + '()'
```

```python
a = Flatten()
```

```python
a(img).shape
```

```
torch.Size([784])
```

```python
new_transform = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    Flatten()
])
```

Поскольку данные в батчи набираются по индексам, мы можем указать множество индексов для train и test. Для это мы реализуем функцию subset_ind, которая отберёт 20 процентов индексов датасета для test, все остальные индексы — на train.

```
[ ]  1  # взять случайные 20 процентов индексов из датасета
     2  def subset_ind(dataset, ratio: float):
     3      return np.random.choice(len(dataset), size=int(ratio*len(dataset)), replace=False)
```

```
[ ]  1  dataset = DatasetMNIST(
     2      './notMNIST_small',
     3      transform=new_transform
     4  )
     5
     6  shrink_inds = subset_ind(dataset, 0.2)
     7  dataset = Subset(dataset, shrink_inds)
     8
     9  print(f'\n\n dataset size: {len(dataset)}, labels: {np.unique(dataset.dataset.labels)}')
```

```
Parsing...
found broken img: ./notMNIST_small/F/Q3Jvc3NvdmVyIEJvbGRPYmxpcXVlLnR0Zg==.png [it's ok if <10 i
found broken img: ./notMNIST_small/A/RGVtb2NyYXRpY2FCb2xkT2xkc3R5bGUgQm9sZC50dGY=.png [it's ok
Done


 dataset size: 3744, labels: [0 1 2 3 4 5 6 7 8 9]
```

```
     1  val_size = 0.2
     2  val_inds = subset_ind(dataset, val_size)
     3
     4  train_dataset = Subset(dataset, [i for i in range(len(dataset)) if i not in val_inds])
     5  val_dataset = Subset(dataset, val_inds)
     6
     7  print(f'  training size: {len(train_dataset)}\nvalidation size: {len(val_dataset)}')
```

```
  training size: 2996
validation size: 748
```

Встроенная в PyTorch функция subset позволяет создать датасет на основе подвыборки. Мы сделаем два загрузчика данных на основе индексов train и test. Далее, на их основе, мы создаём загрузчики, которые будут разбивать данные по батчам для train и test по аналогии с тем, как мы это делали ранее и на прошлом занятии.

```
[ ]  batch_size = 32

     train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, pin_memory=True, num_workers=2)
     test_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True, pin_memory=True, num_workers=2)
```

```
[ ]  train_iter = iter(train_loader)
     print(type(train_iter))

     images, labels = train_iter.next()

     print('images shape on batch size = {}'.format(images.size()))
     print('labels shape on batch size = {}'.format(labels.size()))
```

```
<class 'torch.utils.data.dataloader._MultiProcessingDataLoaderIter'>
images shape on batch size = torch.Size([32, 784])
labels shape on batch size = torch.Size([32])
```

Для примера загрузим одну пачку и убедимся в корректности формы (32 картинки, вытянутые в вектор из 784 значений яркости пикселей чёрно-белого изображения).

Инициализируем модель и оптимизаторы:

```
device=torch.device('cuda')

model = nn.Sequential(
    nn.Linear(784, 10),
    nn.Sigmoid(),
)
model = model.to(device, torch.float32)

opt = torch.optim.Adam(model.parameters(), lr=1e-3)
```

Функция train_model по функционалу совпадает с функцией train, которую мы обсуждали ранее.

```python
def train_model(model, train_loader, test_loader, loss_fn, opt, n_epochs: int):
    train_loss = []
    val_loss = []
    val_accuracy = []

    for epoch in range(n_epochs):
        ep_train_loss = []
        ep_val_loss = []
        ep_val_accuracy = []
        start_time = time.time()

        model.train(True) # enable dropout / batch_norm training behavior
        for X_batch, y_batch in train_loader:
            X_batch = X_batch.to(device)
            y_batch = y_batch.to(device)

            predictions = model(X_batch)
            loss = loss_fn(predictions, y_batch)
            loss.backward()
            opt.step()
            ep_train_loss.append(loss.item())

        model.train(False) # disable dropout / use averages for batch_norm
        with torch.no_grad():
            for X_batch, y_batch in test_loader:
                X_batch = X_batch.to(device)
                y_batch = y_batch.to(device)
                ep_val_loss.append(loss.item())
                y_pred = model(X_batch)
                y_pred = y_pred.max(1)[1].data
                ep_val_accuracy.append(np.mean( (y_batch.cpu() == y_pred.cpu()).numpy() ))

        print(f'Epoch {epoch + 1} of {n_epochs} took {time.time() - start_time:.3f}s')

        train_loss.append(np.mean(ep_train_loss))
        val_loss.append(np.mean(ep_val_loss))
        val_accuracy.append(np.mean(ep_val_accuracy))

        print(f"\t  training loss: {train_loss[-1]:.6f}")
        print(f"\tvalidation loss: {val_loss[-1]:.6f}")
        print(f"\tvalidation accuracy: {val_accuracy[-1]:.3f}")

    return train_loss, val_loss, val_accuracy
```

Далее обучим модель в течение 30 эпох.

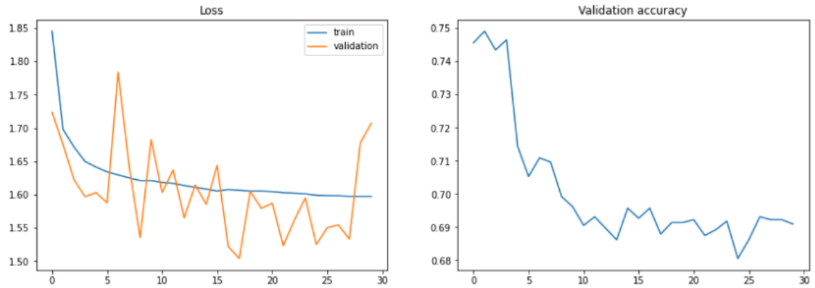
```
[ ]  1  n_epochs = 30
     2
     3  loss_func = torch.nn.CrossEntropyLoss()
     4  train_loss, val_loss, val_accuracy = train_model(model, train_loader, test_loader, loss_func, opt, n_epochs)
```

Визуализируем падение loss и изменение accuracy на тестовой выборке.

```
1  def plot_train_process(train_loss, val_loss, val_accuracy):
2      fig, axes = plt.subplots(1, 2, figsize=(15, 5))
3
4      axes[0].set_title('Loss')
5      axes[0].plot(train_loss, label='train')
6      axes[0].plot(val_loss, label='validation')
7      axes[0].legend()
8
9      axes[1].set_title('Validation accuracy')
10     axes[1].plot(val_accuracy)
```

+ Код + Текст

```
[ ]  1  plot_train_process(train_loss, val_loss, val_accuracy)
```



Попробуем сделать более сложную модель и обучить её:

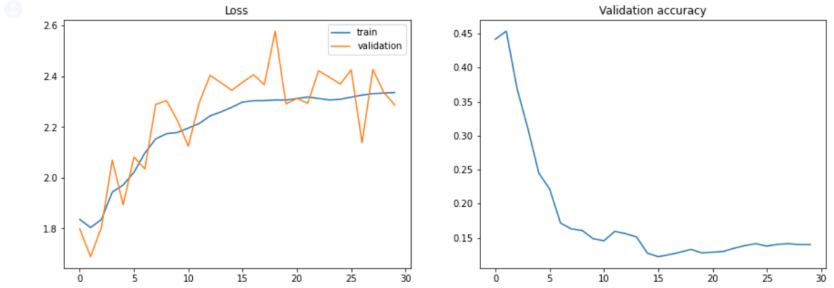
```
model = nn.Sequential(
    nn.Linear(784, 500),
    nn.ReLU(),
    nn.Linear(500, 200),
    nn.ReLU(),
    nn.Linear(200, 10),
    nn.Sigmoid(),
)
model.to(device, torch.float32)

opt = torch.optim.Adam(model.parameters(), lr=1e-3)
```

+ Код + Текст

```
n_epochs = 30

train_loss, val_loss, val_accuracy = train_model(model, train_loader, test_loader, loss_func, opt, n_epochs)
```

Графики показывают, что модель почти сразу переобучилась и тестовое качество сильно упало:

```
plot_train_process(train_loss, val_loss, val_accuracy)
```



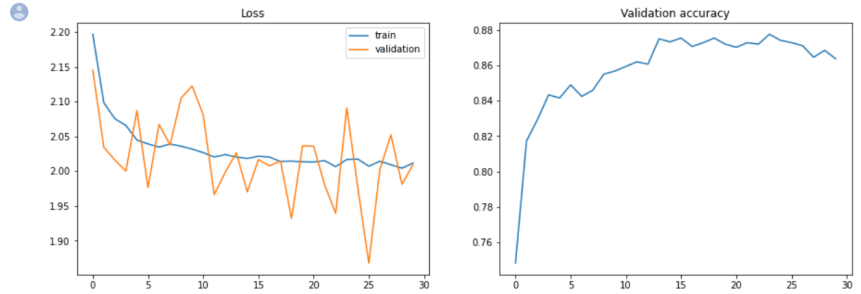
Добавим в модель регуляризацию (стабилизируем обучение) — dropout.

```
[ ]
model = nn.Sequential(
    nn.Linear(784, 500),
    nn.Dropout(p=0.2),
    nn.ReLU(),
    nn.Linear(500, 200),
    nn.Dropout(p=0.6),
    nn.BatchNorm1d(200),
    nn.ReLU(),
    nn.Linear(200, 10),
    nn.Dropout(p=0.6),
    nn.Sigmoid(),
)
model.to(device, torch.float32)

opt = torch.optim.Adam(model.parameters(), lr=1e-4)
```

```
[ ]
n_epochs = 30

train_loss, val_loss, val_accuracy = train_model(model, train_loader, test_loader, loss_func, opt, n_epochs)
```

Теперь видно, что модель стабилизировалось, а качество на тесте начинает расти:

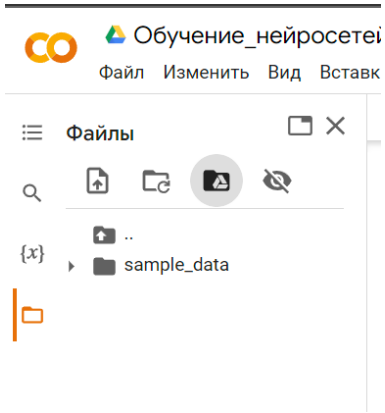
```
plot_train_process(train_loss, val_loss, val_accuracy)
```



Помимо этого важно отметить, что в реальных кейсах обучение может идти часами. Нужно сохранять веса модели (model.state_dict()) и состояния оптимизатора (optimizer.state_dict()), чтобы продолжить обучение (функция load_checkpoint) в случае сбоя.

```
[ ]  1  def save_checkpoint(checkpoint_path, model, optimizer):
     2      state = {
     3          'state_dict': model.state_dict(),
     4          'optimizer' : optimizer.state_dict()}
     5      torch.save(state, checkpoint_path)
     6      print('model saved to %s' % checkpoint_path)
     7
     8  def load_checkpoint(checkpoint_path, model, optimizer):
     9      state = torch.load(checkpoint_path)
    10      model.load_state_dict(state['state_dict'])
    11      optimizer.load_state_dict(state['optimizer'])
    12      print('model loaded from %s' % checkpoint_path)
```

Сбой может произойти, например, если Colab прекратит работу из-за бездействия. В этом случае рекомендуется сохранять модель непосредственно на Google Диске, который можно подключить на боковой панели (внутри будет папка MyDrive):



		Так можно восстановить обучение с момента последнего сбоя.	
Закрепление изученного материала	15 мин.	**Вопросы для обсуждения** • Для чего нужна функция ImageFolder? • Какую функцию нужно использовать для загрузки датасета? • С помощью чего можно реализовать асинхронную загрузку?	Педагог организует беседу по вопросам
Этап подведения итогов занятия (рефлексия)	8 мин.	**Вопросы для обсуждения** • Чему я научился? • С какими трудностями я столкнулся? • Какие вопросы остались? Что осталось непонятным?	Педагог способствует размышлению обучающихся над вопросами
Информация о домашнем задании, инструктаж по его применению	5 мин.	В этом занятии вам предстоит потренироваться в обучении моделей с помощью библиотеки Pytorch. Мы будем работать с игрушечным датасетом moons. Вам предстоит самостоятельно реализовать в Pytorch логистическую регрессию. Вам необходимо написать модуль на PyTorch, реализующий функцию f(X)=Xw, где w — параметр (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).	

		В самом модуле LogisticRegression мы не применяем к выходу сигмоиду, поскольку после этого неудобно будет вычислять функцию потерь. Сигмоиду нужно будет применять к выходу самостоятельно. Поэтому модуль LogisticRegression не будет отличаться от модуля LinearRegression, который мы реализовывали на семинаре Введение в Pytorch. Далее, потренируемся в вычислении функции потерь. Даны матрица объекты-признаки X, вектор весов w и вектор правильных ответов y. Вычислите функцию потерь по алгоритму выше.	

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/.