ТЕХНОЛОГИЧЕСКАЯ КАРТА ЗАНЯТИЯ

Тема занятия: Построение и обучение нейросети

Аннотация к занятию: на данном занятии обучающиеся научатся создавать и обучать свёрточную нейросеть для задачи классификации картинок. В первой части занятия обучающиеся загрузят и подготовят датасет, определят класс свёрточной нейросети. Во второй части обучат свёрточную нейросеть, получат графики функции потерь и метрики качества.

Цель занятия: сформировать у обучающихся представление о создании и обучении свёрточной нейросети для задачи классификации картинок.

Задачи занятия:

- познакомить обучающихся с созданием и обучением свёрточной нейросети;
- получить графики функции потерь и метрики качества;
- применить полученные знания на практике.

Ход занятия

Этап занятия	Время	Деятельность педагога	Комментарии, рекомендации для педагогов
Организационный этап	2 мин.	Добрый день, ребята! Как настроение?	Приветствие. Создание в классе атмосферы психологического комфорта
Постановка цели и задач занятия. Мотивация учебной деятельности обучающихся	10 мин.	**Вопрос для обсуждения** Давайте повторим, как свёрточные нейросети понимают, что изображено на картинке. **Ответы обучающихся** На этом занятии мы научимся создавать и обучать сверточную нейросеть для задачи классификации картинок. Работать мы будем, как обычно, используя фреймворк PyTorch.	Способствовать обсуждению мотивационных вопросов
Изучение нового материала	50 мин.	На этом практическом занятии мы будем использовать датасет CIFAR. Это датасет цветных картинок размером 32 на 32, поделённых на 10 классов. Среди классов — самолеты, автомобили, птицы и другие.	Для справки: https://habr.com/ru/company/intel/blog/415811/ Перед уроком рекомендуется ознакомиться с

		Прежде чем мы перейдем к коду и запуску ячеек, давайте подключим к нашему Jupyter Notebook GPU. GPU — это graphical processing unit, проще говоря, видеокарта. На видеокарте нейросети будут обучаться быстрее, чем на обычном процессоре. Google Colab предоставляет возможность пользоваться видеокартой бесплатно, но для этого её нужно подключить в настройках. Заходим в edit, notebook settings и выбираем GPU. Если GPU уже выбрана, ничего делать не нужно, просто закройте эту вкладку. Если не выбрана, выберите GPU и нажмите save. Рекомендую подключать GPU на Google Colab всегда, когда вы собираетесь работать с нейросетями. Иначе ваши нейросети будут обучаться намного медленнее. Начинаем, как обычно, с импорта библиотек. Импортируем знакомые нам NumPy и Matplotlib, а также некоторые модули библиотеки PyTorch. Во-первых, сам Torch. А также два модуля из библиотеки Torchvision. Это библиотека, в которой находится много полезных функций для обучения нейросетей для компьютерного зрения. То есть для работы с картинками. Сегодня нам понадобятся два модуля этой библиотеки: datasets и transforms. Теперь давайте загрузим датасет. Модуль datasets библиотеки Torchvision, который мы импортировали, позволяет напрямую в коде скачивать некоторые стандартные датасеты. CIFAR, с которым мы сегодня будем работать, — один из стандартных датасетов, которые часто используются в обучающих ноутбуках,	материалами, представленными на сайте.

		как этот. Поэтому его можно очень легко загрузить с помощью модуля datasets. Внутри модуля datasets CIFAR уже поделён на тренировочную и тестовую части, их нужно загружать отдельно. Загрузим обе части и положим тренировочную часть в переменную train_data, а тестовую — в переменную test_data. Нужно вызвать datasets.CIFAR и передать в эту функцию загрузки несколько параметров. Во-первых, параметр root: это название папки, в которую будет скачан датасет. Давайте напишем cifar10_data для обеих частей: тренировочной и тестовой. Мы увидим, что при запуске этой ячейки на диске создастся папка cifar10_data, внутри которой появятся ещё две папки: train и test. В train эта функция скачает тренировочную часть датасета, в test — тестовую часть датасета. Если вы работаете не в Google Colab, а на компьютере, то после запуска этой ячейки у вас на диске рядом с файлом этого Jupyter Notebook появится папка cifar10_data. Теперь к параметру train. Он определяет, какую часть датасета CIFAR мы скачиваем: тренировочную или тестовую. Если train равно True, то будет скачана тренировочная часть. Есть train равно False, то тестовая. Download True означает, что эта функция должна скачать датасет CIFAR из интернета в указанную нами папку. Download False можно ставить тогда, когда у вас на диске в указанной вот тут папке уже лежит датасет	

		CIFAR, и вы только хотите загрузить его в переменные train_data и test_data. У нас датасета еще нет, поэтому обязательно нужно его скачать. И последнее: аргумент transform. Этот аргумент задаёт преобразования, которые будут применены к каждой из картинок датасета при формировании батчей из них. Здесь мы задаём только одно преобразование: transforms.ToTensor(). Это значит, что при формировании батча картинок каждая картинка будет переведена в формат тензора PyTorch. Это логично, ведь когда мы строим нейросети в PyTorch, всё, что мы подаём им на вход, должно быть в формате тензора, включая картинки. Теперь всё. Запустим ячейку. Получим две переменные: в них находятся тренировочный датасет и тестовый датасет CIFAR. Давайте выведем переменную train_data и убедимся, что она представляет собой PyTorch-класс Dataset. Прежде чем мы создадим для датасетов даталоадеры, давайте разобьём тренировочный датасет ещё на две части: тренировочную и валидационную. В итоге получим три части датасета: тренировочную, валидационную и тестовую. Разбить train_data на две части легко: воспользуемся функцией random_split библиотеки PyTorch. Она работает примерно как train_test_split из Sklearn.	

		Для начала нужно понять, сколько картинок из train_data мы отнесём в тренировочную, а сколько — в валидационную выборку. Давайте отнесём в тренировочную выборку восемьдесят процентов картинок. Заведём переменную train_size, которая будет равна количеству картинок в train_data, умноженному на ноль целых восемь десятых. Так как это может быть нецелым числом, сделаем из него целое число, отбросив дробную часть. Train_size — это то, сколько картинок из train_data пойдут в тренировочную выборку. Val_size — это то, сколько картинок из train_data пойдут в валидационную выборку. Val_size — это просто размер train_data минус train_size. Теперь в функцию random_split подаём train_data и размеры будущих выборок: train_size и val_size. На выходе получим два датасета, поделённых в соответствии с заданными размерами. Назовём их train_data и val_data. Теперь у нас есть train_data, val_data и test_data. Заведём для всех даталоадеры. Положим для всех размер батча шестьдесят четыре. Для train_data ставим shuffle равно True, потому что на этих данных мы будем обучать нашу нейросеть. Для остальных частей ставим shuffle равно False, потому что на этих частях мы будем нейросеть только тестировать, перемешивать каждый раз эти части данных нам не обязательно. Теперь мы готовы описывать и обучать нашу нейросеть. Но перед этим давайте посмотрим на пару картинок из нашего датасета, чтобы понимать, с чем мы имеем дело.	

		Чтобы на картинки посмотреть, давайте получим один батч картинок из тренировочного даталоадера. Просто напишем стандартный цикл генерации батчей и после получения первого же батча остановим его. В переменных images и labels после запуска этой ячейки будут содержаться шестьдесят четыре картинки и шестьдесят четыре значения классов этих картинок. Это будут пайторчевые тензоры. Выведем размерности этих тензоров. Размер тензора images — шестьдесят четыре на три на тридцать два на тридцать два. Первое — это размер батча. Далее, три на тридцать два на тридцать два — это размерность картинки. Тридцать два на тридцать два — это длина и высота картинки, три — количество её цветовых каналов. Обратите внимание, что PyTorch представляет картинку не привычным нам способом, когда сначала идут размерности длины и высоты, а затем размерность три — цветовые каналы, а наоборот: сначала идёт размерность канала, потом длина и высота. Такое представление картинки связано с удобством: при таком виде тензора картинки удобнее производить с неё операцию свёртки, свёрточная нейросеть работает быстрее. У тензора labels размер просто шестьдесят четыре: в этом тензоре просто шестьдесят четыре числа. У каждой картинки из images есть номер класса, к которому картинка принадлежит. Посмотрим на картинки. Я написала функцию show_images, в которую подаётся тензор — батч	

		картинок, и ответы к ним. То есть подаются images и labels. Внутри функция визуализирует первые десять картинок из images. Строим здесь поле для визуализации десяти картинок. Затем проходимся в цикле по десяти полям и делаем следующее: берем i-ю картинку из images. Это будет тензор размерностью три на тридцать два на тридцать два. Переводим её из тензора в NumPy, а затем в привычный вид, когда цветовой канал идёт последним, а не первым, то есть делаем так, чтобы размерность картинки стала тридцать два на тридцать два на три. Для этого транспонируем её каналы. И теперь мы можем визуализировать её с помощью plt.imshow. В качестве подписи ставим значение класса этой картинки, его берём из labels. Давайте запустим эту функцию и подадим в неё наши images и labels. Вот что видим: это примеры картинок из датасета CIFAR. Они выглядят довольно пиксельно, но это неудивительно: их размер всего тридцать два на тридцать два. Над каждой картинкой написан номер класса, к которому картинка относится. Чтобы было понятнее, какой класс что означает, ниже указана таблица соответствий названий классов и их номеров. У нас есть самолет, автомобиль, лягушка и так далее. Давайте также на всякий случай заведём словарь, где каждому номеру класса поставим в соответствие его название. Далее, когда мы будем тестировать нашу свёрточную нейросеть, мы сможем взять ответы	

		нейросети и по словарю понять, какому классу какую картинку она отнесла. Итак, пришло время определить класс нашей свёрточной нейросети. Для начала импортируем модули PyTorch, которые нам понадобятся. Это знакомые вам nn, где лежат слои, и F, откуда мы будем брать функции активации. Также импортируем accuracy_score из Sklearn. С её помощью мы будем при обучении и тесте сети измерять accuracy классификации. Переходим к построению нейросети. Как обычно, нейросеть — это класс, у которого нужно определить методы init и forward. В init мы объявим все слои, которые нейросеть будет содержать, а в forward опишем, как картинка будет идти он начала нейросети до её конца через эти слои. Назовём класс ConvNet. Пусть в нашей нейросети будет всего четыре слоя: два свёрточных и два полносвязных. Начинаем с init. Определяем первый свёрточный слой, назовем его conv1. Свёрточный слой — это nn.Conv2d. Внутрь нужно передать следующие аргументы: in_channels — это количество карт активации, которые этот свёрточный слой принимает на вход. Out_channels — количество карт активации, которые этот слой генерирует. То есть это количество фильтров в этом слое. Kernel_size — размер каждого фильтра в слое. Так как наши картинки в датасете CIFAR цветные, то у первого свёрточного слоя in_channels будет	

		обязательно три: у цветной картинки три канала. Если бы наши картинки были чёрно-белыми, то in_channel было бы равно единице. Например, если бы мы решали задачу классификации картинок MNIST, то in_channel было бы равно единице. А вот out_channels мы можем выбрать сами. Пусть будет равно шести. И размер фильтров в этом слое сделаем равным три на три. Изначально картинки CIFAR были размером тридцать два на тридцать два. После этого свёрточного слоя карты активации будут размером тридцать на тридцать. Добавим далее слой пулинга. Назовем его pool. Слой пулинга — это MaxPool2d. Аргумент kernel_size — это размер ядра пулинга. Поставим его два на два: стандартный пулинг, который делит карты активации на квадраты размером два на два и в итоге уменьшает размеры карт активации в два раза. После пулинга карты активации будут иметь размер пятнадцать на пятнадцать. Далее определим второй слой свёрток. Тоже Conv2d. In_channels равно шести: это число совпадает с out_channels предыдущего слоя. Сколько карт активации выдал первый слой, столько получил на вход второй. Out_channels давайте поставим равным девяти. Ради разнообразия давайте поставим kernel_size во втором слое равным четыре на четыре. Ядра размерности три на три используют особенно часто, но	

		я хочу показать, что использовать другие размеры ядер в принципе возможно. После такого свёрточного слоя размер карт активации станет равным двенадцать на двенадцать. Всё, закончим со сверточными слоями. Прежде чем мы перейдём к полносвязным слоям, нужно ещё завести слой flatten. Вообще flatten — это не слой, а операция: она принимает на вход карты активации, растягивает их в векторы и конкатенирует. Но в PyTorch нужно задать эту операцию как слой nn.Flatten. Идём дальше. Остались два полносвязных слоя, fc1 и fc2. В fc1 на входе будет двенадцать на двенадцать на шесть нейронов, потому что карты активации последнего свёрточного слоя были размером двенадцать на двенадцать, и их было шесть. На выходе давайте поставим сто двадцать восемь нейронов. И во втором слое тогда будет сто двадцать восемь на десять нейронов. Десять на выходе, потому что мы решаем задачу классификации на десять классов. Отлично, объявили все нужные слои. Переходим к forward, где распишем, как входящая картинка x будет проходить через эти слои. Сначала прогоняем x через первый сверточный слой. И не забываем добавить функцию активации. В нашей нейросети давайте все функции активации всех промежуточных слоёв сети сделаем ReLU. Прогоняем x	

		через пулинг, далее через второй свёрточный слой и снова функцию активации. Теперь flatten. Прогоняем x через flatten. Полученное значение подаём в первый полносвязный слой и функцию активации, затем во второй полносвязный слой. Второй полносвязный слой — это последний слой сети, поэтому активация здесь не нужна. При обучении сети мы будем использовать функцию потерь кросс-энтропию. В PyTorch в этой функции потерь уже встроена функция активации softmax. Поэтому в самой сети никакой функции активации в последнем слое не нужно. Мы объявили сверточную нейросеть, осталось её обучить. Давайте заведём нейросеть, назовём переменную conv_net. Мы подключали к ноутбуку GPU, чтобы нейросети обучались быстрее. Однако просто подключить GPU для этого недостаточно. Нужно перенести саму нейросеть на GPU. Сказать PyTorch, что нашу нейросеть нужно обучать на GPU, а не на обычном процессоре. Сделать это очень просто. Заведём переменную device. Она будет равна torch.device.cuda, если на сервере доступна GPU, и будет равна torch.device.cpu, если GPU не доступна. Здесь cuda — это GPU, CPU — это обычный процессор. Затем мы просто положим нашу нейросеть на этот девайс: conv_net.to(device).	

		Получается, с помощью этих двух строк кода наша нейросеть автоматически будет перенесена на GPU, если GPU доступна, и останется на обычном процессоре CPU, если видеокарта не доступна. Доступность GPU проверяется с помощью определённой команды: она обращается к серверу и понимает, подключена ли GPU. Мы знаем, что у нас GPU подключена, и проверять это ещё раз, казалось бы, смысла нет. Но на всякий случай стоит это делать: наш GPU может быть, например, сломана. Тогда нам нужно оставить нашу нейросеть на обычном процессоре. Плюс, используя эти две строчки, вы сможете запускать один и тот же код на разных серверах: на тех, где есть GPU, и где его нет. Нейросеть автоматически будет обучаться на GPU там, где она есть, и на CPU там, где видеокарты нет. Это очень удобно. Перейдём к обучению нейросети. Для начала, как обычно, заведем лосс-функцию и оптимайзер. Лосс-функция — знакомая нам кросс-энтропия, оптимайзер — Adam. Передаем сюда conv_net.parameters, задаём learning_rate одна тысячная. Перед тем, как написать функцию для обучения сети, заведём Tensorboard. Tensorboard — это очень удобная утилита для автоматического отслеживания прогресса в процессе обучения нейросети. Давайте научимся Tensorboard заводить и визуализировать. Во-первых, нам нужно создать папку, в которую во время обучения нейросети будут складываться файлы с	

		информацией о текущем значении лосс-функции и accuracy. Назовем эту папку logs. Вот этот код создает эту папку. Эта строчка с помощью этой функции проверяет, есть ли уже папка с названием logs на диске, и если её нет, то функция makedirs такую папку создаёт. Запустим и посмотрим на файлы. Появилась папка logs. Пока что она пустая. Теперь загрузим в наш Jupyter Notebook расширение Tensorboard. Помимо этого, нам нужно завести сущность summarywriter. С помощью summarywriter мы будем во время обучения сети записывать в папку logs файлы с информацией о том, какие loss и accuracy у нашей сети в данный момент обучения. Ниже в коде обучения сети мы увидим, как это делать. Визуализируем Tensorboard. Запустим эту команду. Подождём и увидим вот такую табличку. Пока что в ней ничего нет, потому что мы ещё не запустили обучение нейросети и никакую информацию в папку logs не сохранили. В настройках нужно поставить галочку напротив reload data, чтобы информация здесь обновлялась прямо во время обучения нейросети. Мы будем наблюдать графики в режиме реально времени. Графики будут обновляться раз в тридцать секунд. Переходим теперь непосредственно к функции обучения сети. Определим функцию train. Код для	

		обучения свёрточной нейросети точно такой же, как и для обучения полносвязной. Сейчас мы в этом убедимся. Давайте пройдёмся по функции. Она принимает на вход нейросеть, которую нужно обучать, лосс-функцию и оптимайзер. Также параметр количества эпох для обучения, по умолчанию стоит три. Обучение сети происходит в этом цикле. Итерируемся по эпохам. Внутри эпохи итерируемся по батчам картинок из даталоадера. Получаем батч картинок и ответов к ним, далее прогоняем картинки через нейросеть, получаем выходы сети — логиты, которые вместе с верными ответами подаём в лосс-функцию. Далее вычисляем градиенты (loss.backward), делаем шаг оптимайзера и обязательно optimizer.zero_grad. Как видите, всё ровно то же самое, что и для обучения полносвязных сетей: ничего дополнительно изобретать не нужно. Остальной код в этой функции нужен для получения метрики качества accuracy в разные моменты обучения сети, валидации нейросети на валидационной части данных и записи информации в папку logs для Tensorboard. Будем записывать в папку logs информацию о текущих значениях loss и accuracy нейросети на текущем батче. Заведём ещё до начала обучения сети переменную num_iter. Она будет содержать число итераций, которое	

		прошло с начала обучения сети. Одна итерация — это одна итерация цикла, один шаг обучения сети на одном батче. Будем каждый раз эту переменную увеличивать на один. Далее. С помощью writer запоминаем текущее значение лосса, которое мы получили. Запишем название информации, которую мы сейчас передаём сюда. Давайте напишем loss/train: это означае, что мы сохраняем текущее значение loss-функции на текущем батче тренировочных данных. Далее передаём значение loss и переменную num_iter. То есть сохраняем значение loss-функции на итерации номер num_iter обучения сети. Так writer будет записывать на каждом шаге в папку logs информацию о том, какое значение loss на тренировочном батче было на этом шаге. Выше будет строиться график изменения значения loss с течением итераций обучения. Мы это увидим, когда запустим обучение. Ещё на каждом шаге будем вычислять accuracy на текущем батче. Для этого берём argmax от логитов: для каждой картинки батча получаем номер класса, к которому нейросеть отнесла эту картинку. Далее вычисляем количество совпадающих значений у y_batch и логитов: то есть количество правильных ответов сети на это батче. Делим их на количество элементов батча, получаем accuracy на батче.	

		Текущее значение accuracy вместе с текущим значением num_iter также подаём в writer. В названии запишем accuracy/train. Так во время обучения сети будет визуализироваться график изменения accuracy на тренировочных данных. Здесь остался код. Здесь мы после каждой эпохи обучения сети будем вычислять значение loss-функции и метрики accuracy на валидационных данных. Будем записывать это в writer, чтобы графики изменений accuracy и loss на валидации тоже визуализировались. Чтобы протестировать модель на валидационных данных, задаём здесь model train False. Model train False нужно делать всегда перед тем, как вы собираетесь тестировать вашу модель на каких-либо данных. А выше перед каждой эпохой обучения всегда возвращаем обратно model train True. Считаем accuracy и лосс на валидации с помощью функции evaluate. Она реализована выше. Она принимает на вход модель, даталоадер, на котором функция будет тестировать модель, и loss-функцию. Далее она проходится по батчам даталоадера, прогоняет картинки батча через нейросеть, вычисляет loss-функцию на ответах модели и верных ответах. Запоминает полученное значение loss на батче в массив losses. Дальше вычисляем, к какому номеру класса нейросеть отнесла картинки батча. Берём argmax. Вычисляем количество правильных ответов нейросети на батче.	

		Добавляем к переменной num_correct, в которой по завершении цикла будет записано итоговое количество правильных ответов нейросети на все картинки даталоадера. Чтобы получить итоговый accuracy на валидации, делим num_correct на количество картинок в валидации. Количество картинок — это просто длина даталоадера. Полученные accuracy и среднее значение loss по всем батчам выдаём в качестве ответа функции evaluate. Вернёмся к нашей функции train. Здесь после каждой эпохи запускаем evaluate, получаем loss и accuracy на валидационной выборке. Подаём их на запись в writer, также вместе с текущим значением количества итераций обучения. Называем их Loss/val и Accuracy/val. Вот, собственно, и весь код обучения сети. После завершения всех эпох обучения функция возвращает обученную модель. Перед запуском обучения остался один нюанс. Помните, мы перенесли нашу сеть на GPU? Чтобы нейросеть можно было обучать на GPU, нужно перенести на GPU данные для её обучения. Давайте это сделаем. Посмотрим вот сюда, где X_batch подается на вход сети. Тут нужно X_batch тоже положить на GPU. Делается это ровно так же, как с сетью. Давайте	

		скажем, что X_batch теперь равен X_batch.to(device). Мы перенесли X_batch на GPU. Далее. Так как нейросеть работает на GPU, то её ответы — логиты — тоже будут лежать на GPU. Вычислим loss-функцию между логитами и верными ответами y_batch. Чтобы между ними можно было вычислить loss-функцию, обе переменные должны находиться на одном процессоре: либо обе на GPU, либо обе на CPU. Поэтому нужно либо логиты перенести обратно на CPU, либо y_batch положить на GPU. Давайте положим y_batch на GPU, так loss будет вычисляться быстрее. Запишем y_batch = y_batch.to(device). Теперь всё будет работать. Наконец, запустим обучение нашей нейросети с помощью этой функции. Передаём в функцию нашу conv_net, loss-функцию и оптимизатор, которые мы объявили выше, и ставим количество эпох десять. Обучение началось. Ид`м выше к нашему Tensorboard. Смотрите, тут начали появляться четыре графика: их названия те же, что мы писали в функции train, когда с помощью writer сохраняли значения loss и accuracy на train и валидации. Вот этот график, к примеру, — график accuracy на train. По оси X — итерации обучения, по Y — accuracy нашей сети на одном батче на этих итерациях обучения. Видно, что по мере обучения accuracy в среднем растёт. То, что его так волнует, неудивительно: мы считаем accuracy по одному батчу, а один батч — шестьдесят четыре картинки — это очень мало.	

		Поэтому от батча к батчу accuracy может сильно отличаться, в зависимости от того, какие картинки в него попали. Но в среднем accuracy растёт, и это хорошо. Ниже график loss на train-датасете. В среднем он падает, так и должно быть. Правые графики — графики accuracy и loss на валидации. Их мы вычисляем раз в эпоху и сразу на всём валидационном датасете, поэтому точки на этих графиках не такие частые, и их не так шатает. Но тут тоже всё хорошо: на accuracy растёт, loss падает. Вот так просто с помощью Tensorboard можно выводить графики при обучении нейросетей и следить за прогрессом. При этом код нужен минимальный. Можно, конечно, такие графики в процессе обучения строить и самому, без Tensorboard, а с помощью библиотеки Matplotlib. Но это сложнее. Вам нужно будет завести несколько массивов для хранения значений loss и accuracy на train и валидации, каждые несколько итераций это отрисовывать, вручную подписывать графики. С Tensorboard, на мой взгляд, гораздо проще. Итак, наша сеть обучилась: эта ячейка отработала, прошло десять эпох. Давайте посчитаем итоговый accuracy нейросети на всём наборе тренировочных данных и на тестовых данных. Воспользуемся той же функцией evaluate, с помощью которой мы считали после каждой эпохи accuracy и loss для валидации.	

		Подадим в эту функцию здесь тренировочный даталоадер, тут — тестовый. Запускаем. Последнее. Графики, которые мы получили в Tensorboard для обучения нашей сети, можно легко сохранить онлайн. Так вы сможете возвращаться к ним через время и даже поделиться своими графиками обучения с другими людьми. Делается это с помощью такого кода. Пишем название папки, куда мы сохраняли нашу информацию для отрисовки графиков. У нас это была папка logs. Кстати, если мы просто посмотрим в папку logs, то увидим, что информация там сохранилась в виде файла непонятного расширения. Это не картинки графиков. Так просто вы этот файл на компьютере не откроете, на графики легко не посмотрите. Поэтому лучше сохранить их с помощью этой команды. Можно написать название эксперимента, которому соответствуют графики. Например, если вы обучали нейросеть для задачи классификации собачек, можете написать, что это классификация собачек. Мы тут для примера напишем просто «Мой последний эксперимент». Здесь можно добавить расширенное описание того, что за графики вы сохраняете. Название и описание помогут вам не путаться в ваших графиках, если вы сохраните много разных. Нужно пройти небольшой процесс авторизации.	

Закрепление изученного материала	15 мин.	**Вопросы для обсуждения** ● Как построить сверточную нейросеть из свёрточных и полносвязных слоёв? ● Как обучить свёрточную нейросеть?	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 домашнем задании вы потренируетесь в построении полносвязных и сверточных нейронных сетей. Мы сравним качество нейросетей на датасете MNIST с рукописными цифрами. Наша цель - построить нейронную сеть для решения задачи классификации цифр на 10 классов. **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). **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.	

Рекомендуемые ресурсы для дополнительного изучения:

1. Нейронные сети для начинающих. [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/312450/
2. Что такое свёрточная нейронная сеть. [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/309508/
3. Наглядно о том, как работает свёрточная нейронная сеть. [Электронный ресурс] – Режим доступа: https://habr.com/ru/company/skillfactory/blog/565232/
4. Обзор нейронных сетей для классификации изображений. [Электронный ресурс] – Режим доступа: https://habr.com/ru/company/intel/blog/415811/