**Лабораторная работа №9**
**по дисциплине «Теория информации, данные, знания»**
**«Исследование методов классификации изображенийрукописных цифр с помощью полносвязнойнейронной сети»**

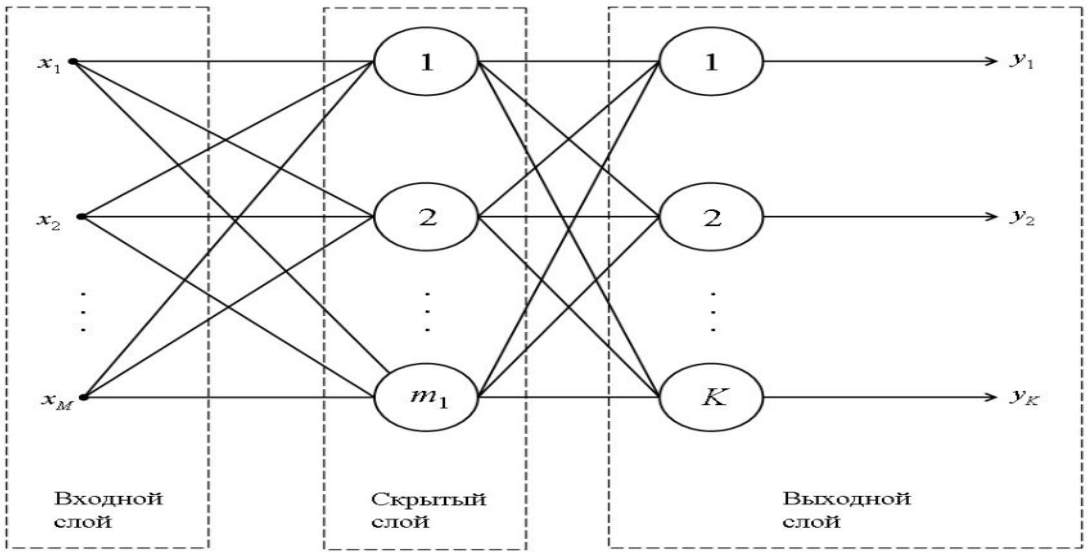
**Цель работы –** исследование принципов разработки нейронной сети на примере задачи классификации изображений помощью GPU.

**Классификация—** понятие в науке, обозначающее разновидность деления объёма понятия по определённому основанию (признаку, критерию), при котором объём родового понятия (класс, множество) делится на виды (подклассы, подмножества), а виды, в свою очередь делятся на подвиды и т.д.

**Общие положения**

Текст программы представлен ниже.

Общая структура полносвязной нейронной сети представлена на рисунке.




В данной работе рассмотрены методы классификации изображений с помощью уже известной полносвязной нейронной сети. Кроме того,представлен способ ускорять вычисления нейронной сети, перекладывая их на GPU.

Сначала проведем инициализацию randomseed – то, что мы делали и в прошлый раз, чтобы эксперименты были воспроизводимы. Подчеркнем, что обучение нейронной сети не будет воспроизводимо внутри одного компьютера. Если вы перезапустите компьютер и проделаете все ячейки сверху вниз, вот тогда результаты уже воспроизведутся. В принципе, можно объединить все инициализации сидов, которые здесь написаны, в функцию, и вызывать функцию целиком.

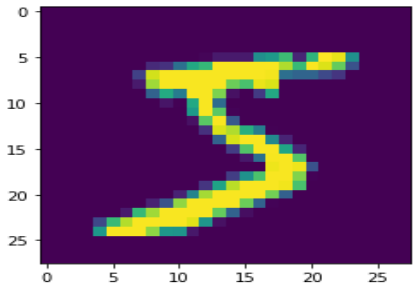
Кроме того, нам понадобится датасет. Самый простой датасетдля классификации изображений – это датасет MNIST, он содержит в себе рукописные цифры от 0 до 9, и его можно скачать с помощью библиотеки torchvision.datasets, мы скачаем и train, и test, и получим соответственно mnist_train, mnist_test.

Посмотрим, что там за данные и что там за «лейблы» – сначала посмотрим какие там типы данных. Мы видим, что *X_data* (сами картинки) имеют тип dtype "unsigned int8", а вот лейблы имеют тип "int64".

```
In [27]: import torchvision.datasets
         MNIST_train = torchvision.datasets.MNIST('./', download=True, train=True)
         MNIST_test = torchvision.datasets.MNIST('./', download=True, train=False)

In [28]: X_train = MNIST_train.train_data
         y_train = MNIST_train.train_labels
         X_test = MNIST_test.test_data
         y_test = MNIST_test.test_labels

In [29]: X_train.dtype, y_train.dtype

Out[29]: (torch.uint8, torch.int64)
```

Хочется, чтобы данные были в дробных числах. Соответственно, преобразуем *X_train* и *X_test* во ***float*** (в дробные числа). Посмотрим на размерности датасетов, которые мы скачали. Видим, что *X_train* и *X_test*имеют размерности 60 тысяч изображений и 10 тысяч соответственно, и сами изображения (они размером 28 на 28) – это очень маленькие картинки, поэтому мы и можем применять полносвязную нейронную сеть для такой задачи.

```
In [31]: X_train.shape, X_test.shape
Out[31]: (torch.Size([60000, 28, 28]), torch.Size([10000, 28, 28]))

In [32]: y_train.shape, y_test.shape
Out[32]: (torch.Size([60000]), torch.Size([10000]))

In [33]: import matplotlib.pyplot as plt
         plt.imshow(X_train[0, :, :])
         plt.show()
         print(y_train[0])
```




```
tensor(5)
```

«Лейблы» *y_train* и *y_test*имеют соответствующий размер и у них нет дополнительной размерности. Это одномерный тензор, и поэтому мы обозначили *X_train* с большой буквы а *y_train* – с маленькой, предполагая, что *X_train* – это многомерный тензор, *y* – одномерный. Кроме того, интересно – а как эти картинки выглядят: если мы нарисуем простой график (можно импортировать matplotlib – библиотеку для рисования графиков, там есть чудесная функция ***imshow*** и она может нарисовать картинку), мы видим что у нас есть какая-то цифра 5, и ей соответствует лейбл – тоже "5". Всё совпадает.

В данномслучае каждая картинка описывается двумерным тензором. Растянем эту картинку в один длинный вектор. Конечно, потеряется некоторая информация о том, какие пиксели были рядом, какие находились далеко, но в принципе для решения задачи этого хватит. Растянуть картинки нам поможет функция reshape.

Тензор *X_train* – это трёхмерный тензор, а мы хотим получить двухмерный тензор, где первая размерность сохраниться – будет 60 000 изображений, а вторая размерность растянется – было 28x28, а получится одна размерность в 784 пикселя. Применим эту опреациюи к train, и к test, получится 2-мерные тензоры X_train и X_test, и их можно передавать в полносвязную нейронную сеть.

Создадим такую нейронную сеть. Она будет очень похожей на ту которую мы создавали для вин, будет состоять из нескольких полносвязных слоёв. Создадим два слоя.

Fc1 – это (fullyconnected 1) слой, на вход которому приходит 784 пикселя. Далее они передаются в *n_hiddenneurons*, который можно задавать в зависимости от того, сколько нам нужно информации в скрытом слое. После этого будет сигмоидная активация, чтобы добавить нелинейность, и после этого, после активации, результат будет передаваться в ещё один полносвязный слой.

На входе у него *n hiddenneurons* (то же самое значение), а на выходе 10 (потому, что у нас 10 классов): это цифры от 0 до 9 (классификация на 10 классов).

Кроме того, нужно написать функцию forward, которая пропускает тензор *X* через все эти слои последовательно: fullyconnected, активация, fullyconnected второй, и выдаёт результирующий тензор, который являет собой выходы из второго полносвязного слоя размером 10.

Назовём эту сеть MNISTnet.Пусть у неё внутри будет 100 скрытых нейронов. Пока пропустим закомментированные части, они понадобятся позже.

Функциейloss точно так же, как и вдатасете с винами, будет кросс-энтропия (обычно используется в классификации). Нужно подчеркнуть, что функция CrossEntropyLoss на вход принимает не вероятности, а те выходы, которые до **софтмакса.** Функция forward не содержала софтмакс, чтобы ускорить вычисления, избежав софтмакса. Для ускорения вычислений можно софтмакс и кроссэнтропию соединить в одну функцию.

Метод градиентного спуска – Adam (практически, золотой стандарт) с learningrate 1e-3, вы можете попробовать какой-то другой, может быть сойдётся лучше. На его вход передаются, как и в прошлый раз, все параметры нейронной сети.

Как и в работе №2, будем обучаться батчевым или стохастическим градиентным спуском. Будем делить наш датасет на маленькие части (так называемые батчи), передавать эти батчи в нейронную сеть с помощью функции forward. После этого будем вызывать loss-функцию, которая скажет нам размер ошибки. После этого можно посчитать градиент с помощью функции backward, и дальше сделаем градиентный шаг, вызвав optimizer.step.

Batch_sizeимеет размер 100 (это число, которое выбрано наугад), поставим 10 000 эпох, учитывая что мы можем остановить обучение в любой момент, а 10000 точно хватит.

Каждый раз будем перемешивать датасет на каждой эпохе и выделять оттуда последовательные участки, таким образом, чтобы внутри одной эпохи каждый элемент, каждая картинка была показана всего один раз.

После этого будем вызывать функцию forward на батче, дальше будем считать loss-функцию по prediction, то есть выходам нейронной сети, и по Y_batch, то есть по лейблам – по тем классам, которые мы бы хотели, чтобы нейросеть предсказала. Далее можнопосчитать backward, то есть посчитать градиенты, а потом сделать градиентный шаг у оптимайзера.

Кроме того, хотелось бы видеть каждую эпоху, как растет качество распознавания. Видеть рост этого качества лучше всего на тестовом датасете – на тех данных, которые нейросеть не видела. Поэтому будем вызывать mnistnet.forward ещё и на тестовом датасете, X_test. Будем делать forward по всему тестовомудатасету, предполагая, что он не очень большой.

Батчевый градиентный спуск нужен всего по двум причинам. Во-первых, потому что таким образом сам процесс оптимизации происходит быстрее – лучше сделать 10 градиентных шагов на эпоху, чем сделать один. Во-вторых, потому что у нас могут не помещаться батчи в GPU (на видеокарту), если мы будем делать их довольно большими.

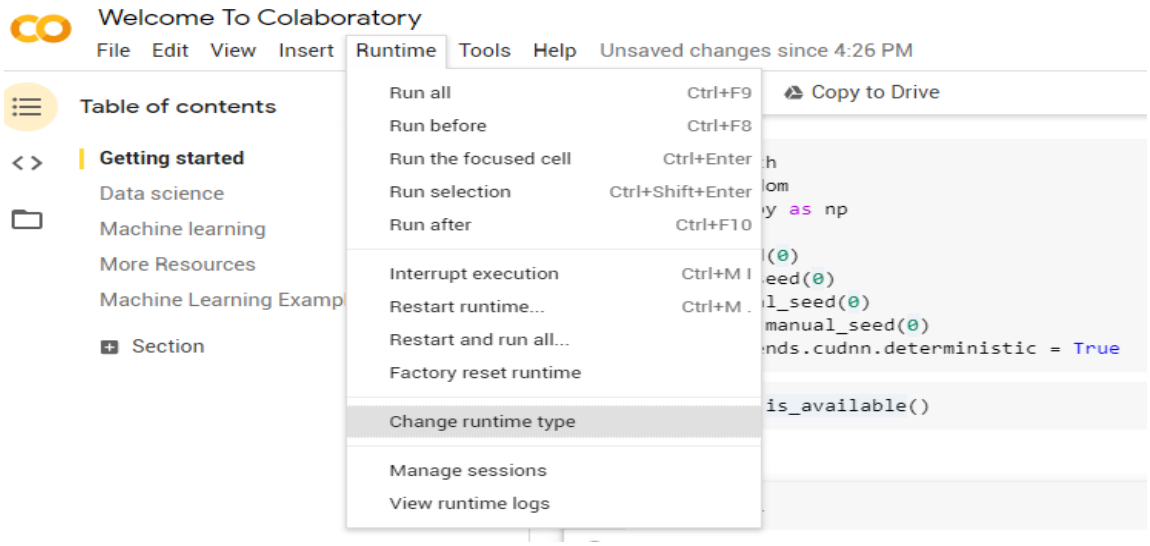
Посчитаем долю правильных ответов –accuracy. Нейронная сеть предсказывает класс, для которого она выдает максимальное значение. Это значение отправляется в софтмакс, если нужно узнать его вероятность, хотя уже перед софтмаксомясно, какой наиболее вероятный класс предсказала нейронная сеть. Это будет тот нейрон, у которого максимальный выход.

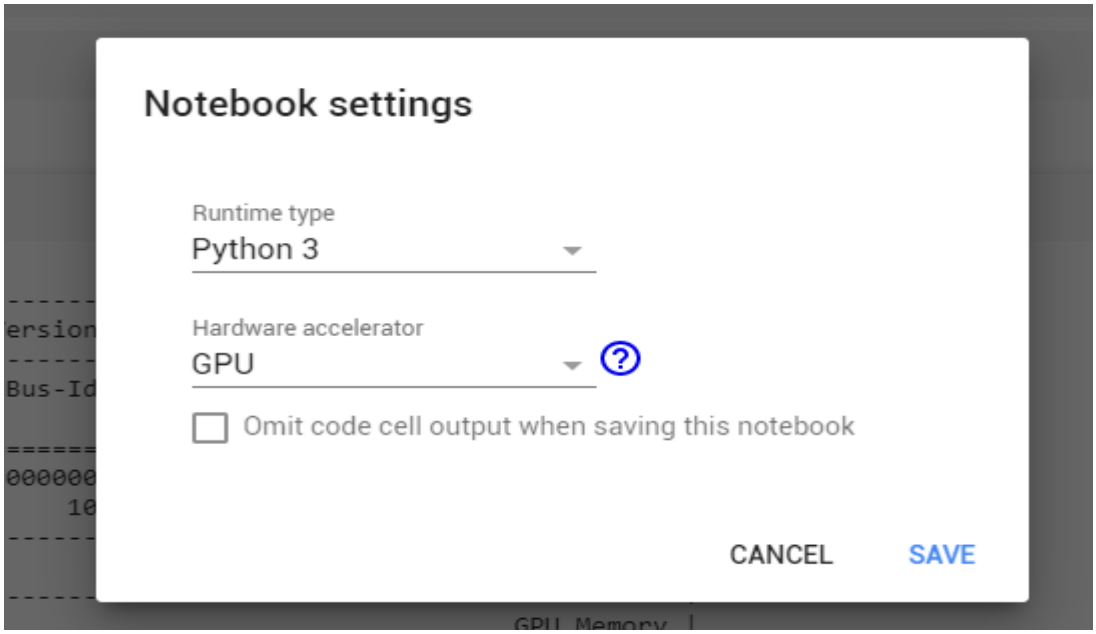
Для его выявленияпыполнимargmax (argmax нам отдаст номер нейрона, одного из десяти, у которого максимальный выход), и сравним это значение с *y_test*.

Если номер нейрона с максимальным выходом совпал с номером класса (от нуля до девяти), значит сеть работает верно. Посчитав среднее количество угадываний, получим accuracy.

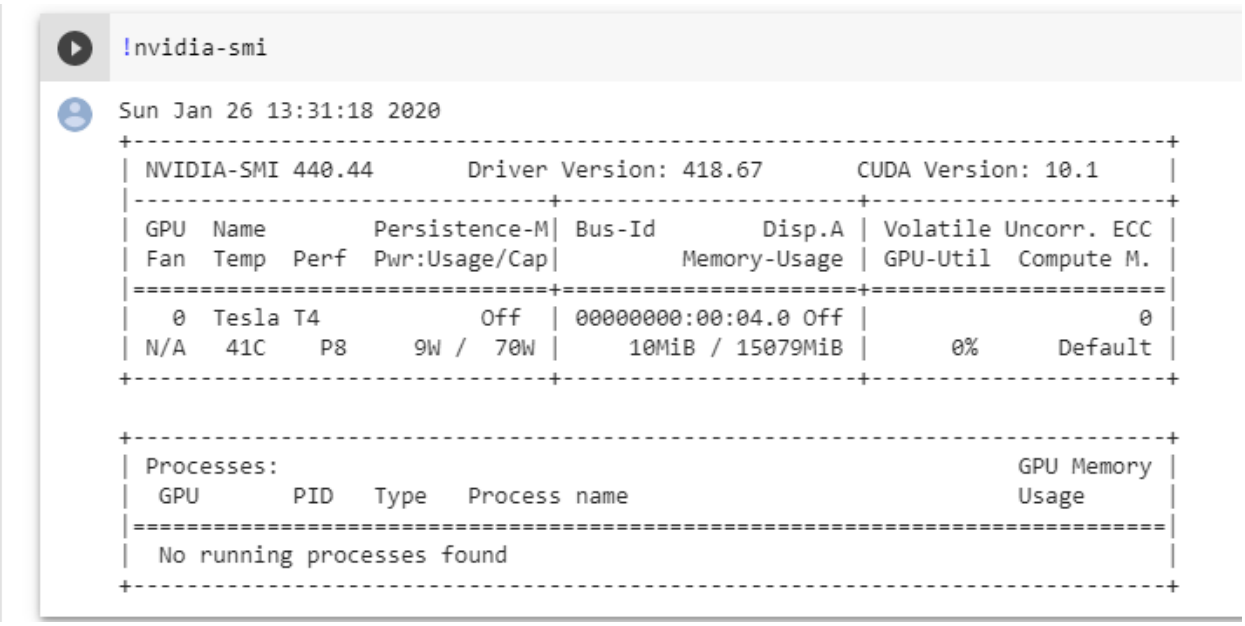
Запустим процесс обучения и увидим, что сеть работает довольно медленно, эпоха происходит за несколько секунд, но уже сразу после обучения одной эпохи получается accuracy 90%. Понятно, что если нейронная сеть будет предсказывать случайные цифры, то она будет иметь в среднем accuracy 10%, то есть нейронная сеть уже за одну эпоху довольно неплохо обучилась.

Для ускорения этого процесса переместим вычисления на GPU. При использовании GoogleColab также можно воспользоваться видеокартой. Убедитесь, что у вас Runtime стоит в "GPU", а не в "CPU", иначе ничего не получится.




Далеенужно остановить текущие вычисления (Runtime->Interruptexecution). Затемс помощью строчки **torch.cuda.is_available()**проверим, видит ли Torch GPU? Если результат будет True, то видео-карта видна. Кроме того, можно посмотреть, а занята ли видеокарта какими-то вычислениями. Это делается с помощью консольной команды "!nvidia-smi". Можно выполнять консольные команды прямо из интерфейса Jupyter или из интерфейса Collab, если ставить восклицательный знак перед командой.




Видно, что видеокарта не простая, а TeslaT4 (Colabзадействовал ресурсы Google)и сейчас на ней занято всего 10MB из 15 000. Кроме того, видно, что она не занята никакими вычислениями и её утилизация (gpuutil) –0%. Также "noprocessesfound" означает, что на видеокарте сейчас нет исполняемых процессов.

Примечание.

NVIDIA представила на GTC 2018 в Японии новое семейство GPU для машинного обучения и анализа информации в центрах обработки данных. Новые решения Tesla T4 («T» в названии означает новую архитектуру Turing) являются преемниками текущего семейства GPU P4, которые сейчас предлагают практически все крупные поставщики облачных вычислений. По словам NVIDIA, Google будет в числе первых компаний, которые задействуют в своей облачной платформе ускорители T4.

Теперь переложим вычисления на видеокарту (программный код, который был закомментирован). Перекладываем тензоры на видеокарту и, автоматически, те вычисления, которые происходили с этими тензорами, начинают происходить на видеокарте. Во-первых, необходимо переложить на видеокарту веса нейронной сети.

Также нужно переложить на видеокарту входы – те картинки, которые мы передаём в нейросеть. Создадим переменную "device", которая будет либо строчкой "cuda:0", что соответствует нулевой видеокарте (если их много), либо "cpu", если видеокарты нет.

Таким образом, создается кросс-платформенный код, который будет работать и на CPU, и на GPU, в зависимости от того, что выдаст строчка. torch.cuda_is_available. Если будет True, то, соответственно, device превратится в "cuda:0", а если "cuda_is_available" равно False, значит будет CPU.

Теперь нужно на этот device переложить нейронную сеть. Это делается командой "**mnist_net.to(device)",**так как mnist_netотнаследован от torch.nn.module. Именно torch.nn.module имеет встроенную функцию **to**, которая может перекладывать веса нейронной сети, её параметры, на видеокарту или на CPU.

Чтобы убедиться, что нейронная сеть переложилась на GPU, можно вывестипараметры на экран. Увидим некоторый список, в котором элементы – это параметры нейронной сети. Возьмём первый элемент в списке – тензор, который

содержит в себе все веса первого слоя. Второй элемент – это все bias первого слоя (все смещения).

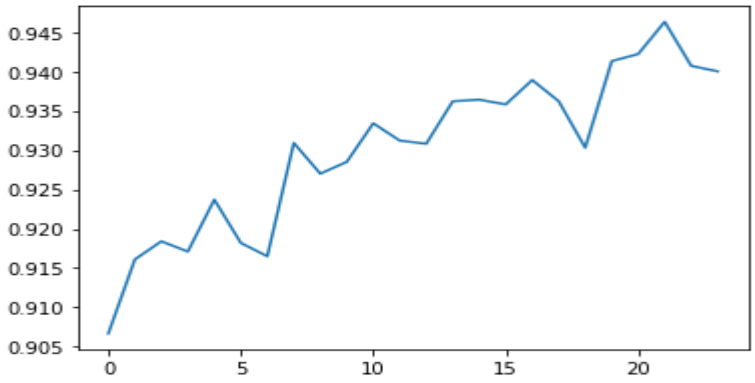
Видно, что везде прописан device cuda:0 – значит, все параметры теперь лежат на видеокарте. Кроме того, нужно переложить на видеокарту входные данные, тестовый датасет*X_test*, потому что мы тестируем по всему датасету сразу и считаем, что он помещается в видеопамяти.*Y_train* и *X_train* не будем сразу класть на видеокарту, будем это делать по батчам,

После перезапуска (для корректности), увидим, что обучение пошло гораздо быстрее (в несколько десятков раз). Обучение, которое проходит на видеокарте, как правило, в десятки раз быстрее, чем обучение, которое происходит на CPU.

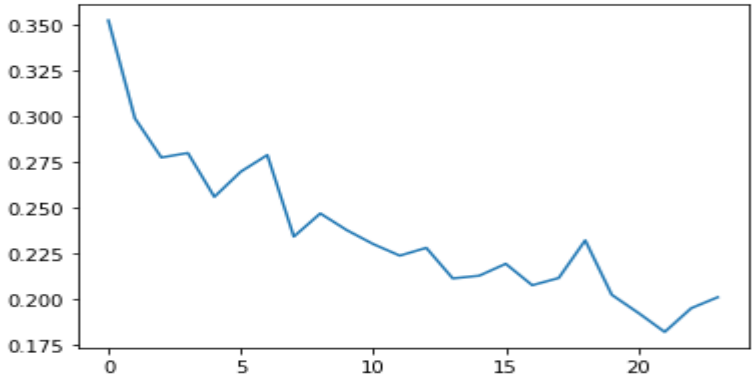
Можно усложнить нейронную сеть, добавить связанные слои, изменить активации, попробовать разные параметры для Адама, попробовать другой градиентный спуск,попробовать другой размер батча.

Дляпостроенияграфика**accuracy**и**loss**добавим"test_loss_history" и "test_accuracy_history".

```
Out[17]: [<matplotlib.lines.Line2D at 0x206696ae648>]
```

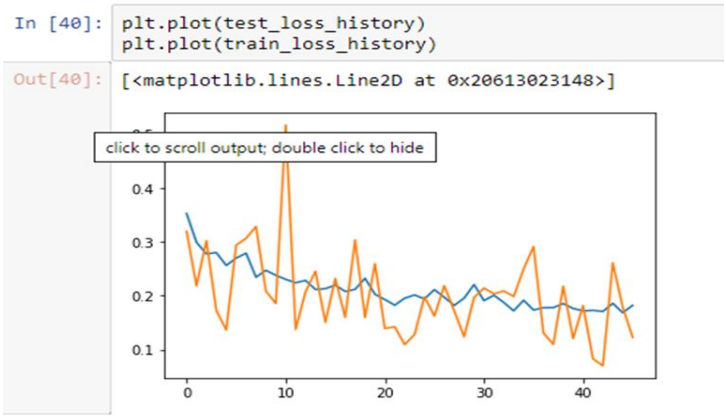



```
In [18]: plt.plot(test_loss_history);
```




Обычно колебания в **loss** меньше, чем в **accuracy**, потому что accuracy имеет округлённые значения, а в loss учитываются вероятности, которые не так дискретны и не дают таких колебаний на графике.

Кроме того, можно смотреть на такие графики раздельно по train и по test. Если вы увидите, что качество на train растёт гораздо лучше, это значит, что вы имеете дело с явлением **переобучения** и ваша нейронная сеть, вместо того, чтобы выучивать какие-то новые вещи, просто «запоминает» картинки. *Это можно побороть с помощью уменьшения параметров нейронной сети, уменьшения количества слоёв и уменьшения их размера.*



```
importtorch
importrandom
importnumpyasnp

random.seed(0)
np.random.seed(0)
torch.manual_seed(0)
torch.cuda.manual_seed(0)
torch.backends.cudnn.deterministic=True
##
```

```
importtorchvision.datasets
MNIST_train=torchvision.datasets.MNIST('./', download=True, train=True)
MNIST_test=torchvision.datasets.MNIST('./', download=True, train=False)
##
```

```
X_train=MNIST_train.train_data
y_train=MNIST_train.train_labels
X_test=MNIST_test.test_data
y test=MNIST_test.test_labels
##
```

```
X train.dtype, y_train.dtype
##
```

```
X_train=X_train.float()
X test=X_test.float()
##
```

```
X train.shape, X_test.shape
##
```

```
y_train.shape,y_test.shape
##
```

```
importmatplotlib.pyplotasplt
plt.imshow(X_train[0, :, :])
plt.show()
print(y_train[0])
##
```

```
X_train=X_train.reshape([-1, 28*28])
X test=X_test.reshape([-1, 28*28])
##
```

```
classMNISTNet(torch.nn.Module):
def__init__(self,n_hidden_neurons):
super(MNISTNet,self).__init__()
self.fc1=torch.nn.Linear(28*28,n_hidden_neurons)
self.ac1=torch.nn.Sigmoid()
self.fc2=torch.nn.Linear(n_hidden_neurons,10)
defforward(self,x):
x=self.fc1(x)
x=self.ac1(x)
x=self.fc2(x)returnx
mnist_net=MNISTNet(100)
##

# torch.cuda.is_available()

# !nvidia-smi

# device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
# mnist_net = mnist_net.to(device)
# list(mnist_net.parameters())

loss=torch.nn.CrossEntropyLoss()
optimizer=torch.optim.Adam(mnist_net.parameters(),lr=1.0e-3)

batch_size=100
# test_accuracy_history = []
# test_loss_history = []

# X_test = X_test.to(device)
# y_test = y_test.to(device)

for epoch inrange(10000):
order=np.random.permutation(len(X_train))

forstart_indexinrange(0, len(X_train), batch_size):
optimizer.zero_grad()

batch_indexes=order[start_index:start_index+batch_size]

X_batch=X_train[batch_indexes] #.to(device)
y_batch=y_train[batch_indexes] #.to(device)

preds=mnist_net.forward(X_batch)

loss_value=loss(preds, y_batch)
loss_value.backward()

optimizer.step()

test_preds=mnist_net.forward(X_test)
#     test_loss_history.append(loss(test_preds, y_test))

accuracy= (test_preds.argmax(dim=1) ==y_test).float().mean()
#     test_accuracy_history.append(accuracy)
print(accuracy)
```

**Задание на лабораторную работу**

1. Исследовать нейронную сеть при заданных начальных параметрах (см. таблицу).
2. Исследовать зависимость точности распознавания от количества нейронов в скрытом слое, количества слоев, метода активации.
3. Замерьте время вычисления 100 эпох на CPU и на GPU. Какое ускорение вы наблюдаете?
4. Постройте на одном графике loss для train и test. Имеется ли переобучение сети?

Таблица. Начальные значения гиперпараметров нейронной сети

| **Вариант** | **Метод оптимизации** | **Число нейронов в скрытом слое** ***n_hidden_neurons*** | **Шаг градиентного спуска** ***lr*** |
|---|---|---|---|
| 0 | ADAM | 10 | 0.01 |
| 1 | ADAM | 20 | 0.001 |
| 2 | ADAM | 30 | 0.01 |
| 3 | ADAM | 40 | 0.001 |
| 4 | ADAM | 5 | 0.01 |
| 5 | SGD | 10 | 0.001 |
| 6 | SGD | 20 | 0.01 |
| 7 | SGD | 30 | 0.001 |
| 8 | SGD | 40 | 0.01 |
| 9 | SGD | 50 | 0.001 |

**Содержание отчета**

1. Титульный лист
2. Цель работы, постановка задачи исследования.
3. Описание методики исследования.
4. Результаты исследования в соответствии с заданием.
5. Выводы по работе.