Лабораторная работа №4
по дисциплине «Системы поддержки принятия решений»
«Исследование методов классификации изображений рукописных цифр с помощью сверточной нейронной сети»

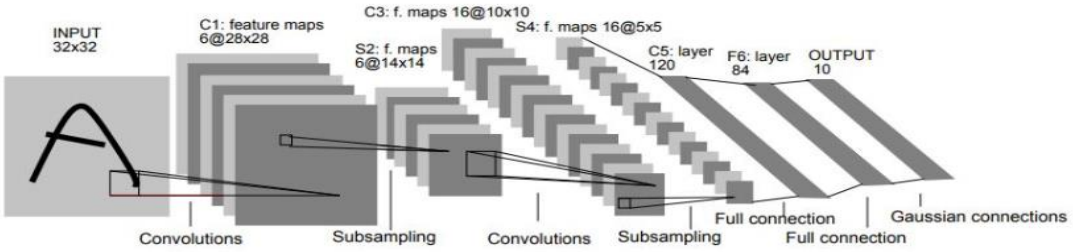
Цель работы – исследование принципов разработки сверточной нейронной сети на примере задачи классификации изображений.

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 сверточной нейронной сети LeNet (1998) представлена на рисунке.



Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	32x32	-	-	-
1	Convolution	6	28x28	5x5	1	tanh
2	Average Pooling	6	14x14	2x2	2	tanh
3	Convolution	16	10x10	5x5	1	tanh
4	Average Pooling	16	5x5	2x2	2	tanh
5	Convolution	120	1x1	5x5	1	tanh
6	FC	-	84	-	-	tanh
Output	FC	-	10	-	-	softmax

В лабораторной работе №3 мы классифицировали датасет MNIST (рукописные цифры) с помощью полносвязной нейронной сети. Улучшим наш результат, использовав сверточную нейронную сеть. Рассмотрим архитектуру LeNet (так называлась сеть у ее автора – **ЛеКуна, 1998**) – самой первой свёрточной нейронной сети. Она уже не самая лучшая, но довольно логичная.

На вход (см. рисунок) подаётся изображение размером 32 × 32, но в MNIST изображения 28 × 28, поэтому изображение от MNIST прогоним через свёртку 5 × 5 с шестью выходными каналами, свёртка 5 × 5 проходит по всему изображению. У неё нулевые **паддинги**, то есть она не выходит за границы изображения, по этой причине она обрезает

два пикселя с каждой стороны изображения, и выходное изображение от 32 × 32 преобразуется к изображению 28 × 28.

Но теперь это уже не изображение, а тензор глубиной 6, потому что свёртка имеет шесть выходных каналов. После этого тензор (28 × 28 × 6) передается в **average pooling** (сейчас уже везде используют **max pooling**, но здесь используется average pooling, как в классической сети LeNet), который вычисляет среднее значение по квадрату 2 × 2.

Он проходит по всему изображению со **страйдом** 2 (то есть берёт непересекающиеся участки 2 × 2), и вычисляет среднее значение пикселей или чисел, которые оказываются в этом тензоре. На выходе у него один пиксель, таким образом, всё изображение ужимается в два раза: было 28 × 28 × 6, стало 14 × 14 × 6.

Далее повторяется точно такая же свёртка, как в первый раз, только теперь у неё количество входных каналов – 6, а выходных каналов – 16. При этом, из 14 × 14 изображения получается 10 × 10, потому что по два пикселя «съелось» с каждой стороны, так как применялась свёртка без паддингов (без выхода за границы изображения).

Далее снова 2 × 2 average pooling, и затем есть два варианта (в принципе они аналогичные). В LeNet берут свёртку 5×5, и получается тензор 5 × 5 × 16 с количеством входных каналов 16, а выходных – 120. Эта свёртка превращает глубокий тензор в один вектор размера 1 × 1 × 120.

Сделаем немного по-другому: сначала тензор 5 × 5 × 16 растянем в один вектор. Теперь есть готовый вектор, к которому применим полносвязный слой, который на выходе будет иметь также 120 нейронов. При этом количество весов в обоих алгоритмов, очевидно, будет одинаковое.

После этого применим к нему два полносвязных слоя. Первый полносвязный из 120 нейронов получает 84 нейрона, а второй полносвязный слой дает ответ в виде 10 нейронов. Если требуются вероятности распознавания, то следует использовать **софтмакс**.

Таким образом, *сверточные нейронные сети занимаются тем, что постепенно сжимают информацию*. С каждой итерацией пулингов информации становится всё меньше, потому что в тензорах меньше данных, но эта информация становится всё ближе к итоговым результатам – очищается всякий шум, а всё нужное остаётся.

Примечание

В LeNet функции активации – тангенсы. Сейчас применяют другие активации, например, ReLU или ELU тут было бы более актуально. Также вместо свёртки 5 × 5 применяют две свёртки 3 × 3.

В отличие от полносвязной сети, которая видела картинку как один длинный вектор, в конволюционную сеть передается картинка как трёхмерный тензор. Первый канал – это глубина картинки, в черно-белой картинке это 1 канал с яркостью серого пикселя, а в RGB картинке были бы RGB каналы. Таким образом, картинку 28 × 28 необходимо разжать до 1 × 28 × 28. Делаем X_train.unsqueeze и ставим нужный индекс.

```
In [8]: X_train = X_train.unsqueeze(1).float()
        X_test = X_test.unsqueeze(1).float()

In [9]:
        X_train.shape

Out[9]: torch.Size([60000, 1, 28, 28])
```

Теперь ***X_train*** – тензор 60 000 × 1 × 28 × 28, то же самое делаем с тестом.

Конволюционный (сверточный) слой, называется Conv2d, потому что он двухмерный. Если бы у нас были трёхмерные изображения, например, мозга человека, можно было бы использовать Conv3d, который в PyTorch тоже есть. В этом Conv2d есть много параметров: количество входных каналов – 1, выходных каналов будет 6. Размер ядра свёртки равен 5, если свёртка не симметричная (бывают растянутые свёртки), то можно указать **tuple** из двух чисел, но у нас свёртка будет квадратная. Такой слой назовём conv1.

Далее мы должны применить активизацию. Активации в LeNet – тангенсы, соответственно применяем torch.nn.Tanh. После этого – первый пулинг, называется pool1, это average pooling. Напоминаю, что average pooling уже не так популярны, они используются обычно в конце архитектуры, чтобы сжать всё изображение, которое получилось, а в середине сети обычно используется max pooling, но в силу традиций здесь используем average pooling 2d. У него kernel_size равно 2, потому что это пулинг 2 × 2, и stride равно 2, потому что он применяется без пересечений. Этот пулинг сожмёт изображение от 28 × 28 до 14 × 14. И так далее, аналогично, в соответствии с указанной на рисунке архитектурой сети.

В функции **forward** мы повторяем эту логику, но теперь применяем эти слои к некоторому входному тензору *X*. Входной тензор *X* – это, на самом деле, батч из картинок. Мы применяем конволюцию, активацию, пулинг, конволюцию, активацию, пулинг, и далее растянем тензор, который на самом деле четырёхмерный, потому что первая размерность отвечает за размерность батча.

У PyTorch-тензоров есть функция **view**, которая тензор преобразует к нужной размерности. Первая размерность будет *x.size[0]* – это размер батча, а дальше тензор будет одномерный, соответственно эти три размерности нужно перемножить и получить 400. Далее первый полносвязный слой, активация, второй полносвязный, активация, полносвязный слой.

Перенесем сразу работу на GPU. Соответственно, создана переменная **device**, как и в работе №3. Далее инициализируем функцию **loss** в виде кросс-энтропии, так как решаем задачу классификации. Используем градиентный спуск ADAM, которому на вход передаются все параметры сети, с шагом градиентного спуска 0.001.

Далее описан процесс обучения по батчам размером 100, каждую эпоху печатаем **accuracy** и накапливаем **loss** на данной эпохе. Внутри каждого батча обнуляются градиенты, батч прогоняется через сеть с помощью функции **forward**, которую мы реализовали выше. Далее считаем градиенты и делаем шаг градиентного спуска.

*Важный момент, **утечка памяти** в данном примере. Сеть становится чуть-чуть побольше, и видеокарты не хватает – в какой-то момент обсчёт падает, если результат вычисления **лосса** на тестовых данных помещать непосредственно в **list**.*

На самом деле, в **loss** хранится весь граф, который помогает потом обсчитать градиенты. При этом он хранится на GPU (если его специально на CPU не перенести), и память не очищается (на GPU накапливаются эти графы).

Для этого используем ".data", и останется только одно число – скаляр. После этого отправим результат на CPU, чтобы он не занимал память.

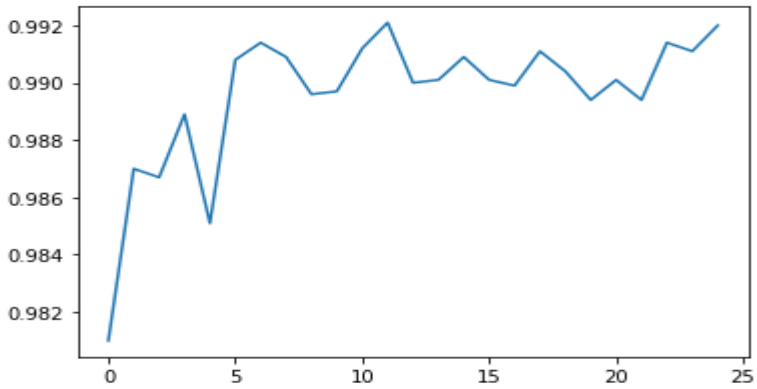
Так же, как в прошлый раз, берём тот нейрон, у которого *выход наибольший*. Сравниваем номер этого нейрона с y_test, там находится номер той цифры, которую мы хотим предсказать. Далее преобразовываем ответ к float, потому что именно от float можно взять среднее (mean).

Далее складируем accuracy в test_accuracy_history.

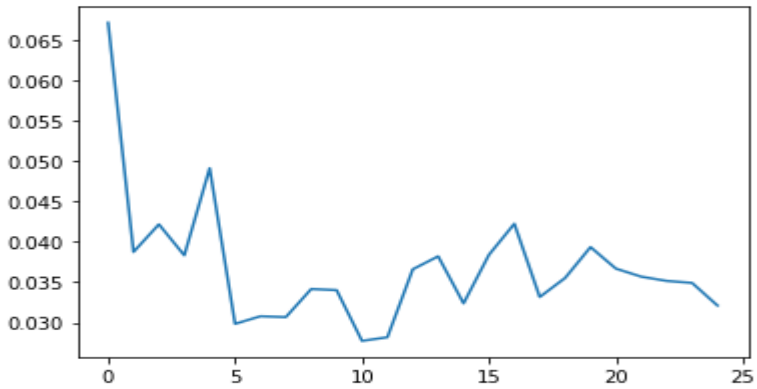
Запустим нейронную сеть и посмотрим, как она обучается. Видим, что на первой же эпохе она обучилась на 97-98%. Это, в принципе, хорошо, но не предел возможностей: самые лучшие результаты на MNIST составляют около **30 сотых одного процента**.

Видим, что сеть довольно долго проходит эпохи. На самом деле это хорошая скорость, хотя в нашей нейронной сети несколько десятков тысяч параметров, это не большая, а даже маленькая сеть. И, скорее всего, всё время тратится на то, чтобы передать датасет из оперативной памяти на видеокарту.

```
In [79]: plt.plot(test_accuracy_history);
```




```
In [80]: plt.plot(test_loss_history);
```



```
import torch
import random
import numpy as np

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

import torchvision.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
##

len(y_train), len(y_test)
##

X_train = X_train.float()
X_test = X_test.float()
##

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

X_train = X_train.unsqueeze(1).float()
X_test = X_test.unsqueeze(1).float()
##

X_train.shape
##
```

```
class LeNet5(torch.nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()

        self.conv1 = torch.nn.Conv2d(
            in_channels=1, out_channels=6, kernel_size=5, padding=2)
        self.act1  = torch.nn.Tanh()
        self.pool1 = torch.nn.AvgPool2d(kernel_size=2, stride=2)

        self.conv2 = torch.nn.Conv2d(
            in_channels=6, out_channels=16, kernel_size=5, padding=0)
        self.act2  = torch.nn.Tanh()
        self.pool2 = torch.nn.AvgPool2d(kernel_size=2, stride=2)

        self.fc1   = torch.nn.Linear(5 * 5 * 16, 120)
        self.act3  = torch.nn.Tanh()

        self.fc2   = torch.nn.Linear(120, 84)
        self.act4  = torch.nn.Tanh()

        self.fc3   = torch.nn.Linear(84, 10)

    def forward(self, x):

        x = self.conv1(x)
        x = self.act1(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.act2(x)
        x = self.pool2(x)

        x = x.view(x.size(0), x.size(1) * x.size(2) * x.size(3))

        x = self.fc1(x)
        x = self.act3(x)
        x = self.fc2(x)
        x = self.act4(x)
        x = self.fc3(x)

        return x

lenet5 = LeNet5()
##
```

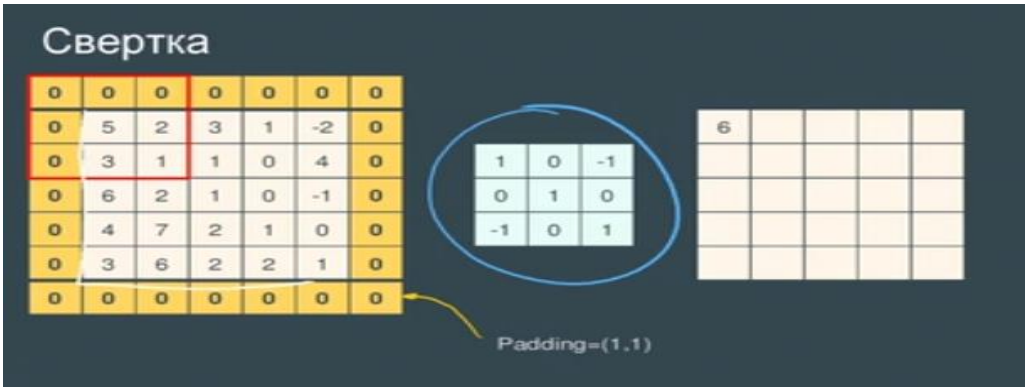
```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
lenet5 = lenet5.to(device)
##
```

```
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(lenet5.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 in range(10000):
    order = np.random.permutation(len(X_train))
    for start_index in range(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 = lenet5.forward(X_batch)

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

        optimizer.step()

    test_preds = lenet5.forward(X_test)
    test_loss_history.append(loss(test_preds, y_test).data.cpu())

    accuracy = (test_preds.argmax(dim=1) == y_test).float().mean().data.cpu
()
    test_accuracy_history.append(accuracy)

    print(accuracy)
```
##
```
lenet5.forward(X_test)
```
##
```
# plt.plot(test_accuracy_history);
plt.plot(test_loss_history);
```

Задание на лабораторную работу

1. Изучить понятия: **свертка, паддинг, страйд, average_pooling, max_pooling.**



2. Исследовать нейронную сеть при заданных начальных параметрах (см. таблицу).
3. Исследовать зависимость точности распознавания от количества слоев, метода активации (например, ReLU), шага ***lr*** и типа пулинга.
4. Замерьте время вычисления 100 эпох на CPU и на GPU. Какое ускорение вы наблюдаете?
5. Попробуйте добиться качества 0.992 на данном датасете.

   Обратите внимание на следующие моменты:
   Появляется ли у вас переобучение при увеличении количества эпох?
   Как добавление различных слоев влияет на скорость обучения (какие слои быстрее: сверточные или полносвязные)?
6. Проверьте верность распознавания на нескольких примерах входных данных.

```
In [48]: print(y_test[199])
         tensor(2, device='cuda:0')

In [57]: X_test1=X_test[199, :, :, :]

In [58]: X_test1 = X_test1.unsqueeze(0).float()

In [59]: X_test1.shape
Out[59]: torch.Size([1, 1, 28, 28])

In [60]: lenet5.forward(X_test1)
Out[60]: tensor([[-2.6600,  0.8833, 12.3160, -0.6526, -0.1819, -6.2044, -2.5759,  1.7355,
                   1.8877, -3.6546]], device='cuda:0', grad_fn=<AddmmBackward>)

In [63]: X_test1=X_test[199, 0, :, :].to('cpu')

In [64]: plt.imshow(X_test1)
         plt.show()
```

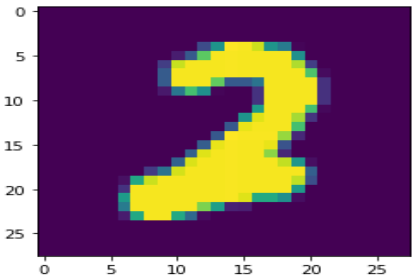


Таблица. Начальные значения гиперпараметров нейронной сети

Вариант	**Метод оптимизации**	**Метод активации**	**Шаг градиентного спуска *lr***
0	ADAM	Tanh	0.01
1	ADAM	Tanh	0.001
2	ADAM	ReLU	0.01
3	ADAM	ReLU	0.001
4	SGD	Tanh	0.01
5	SGD	Tanh	0.001
6	SGD	ReLU	0.01
7	SGD	ReLU	0.001
8	SGD	Tanh	0.0001
9	ADAM	Tanh	0.0001

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.

Полезная ссылка

https://neurohive.io/ru/tutorial/cnn-na-pytorch/