Лабораторная работа №5
по дисциплине «Системы поддержки принятия решений»
«Исследование архитектур сверточных нейронных сетей
на датасетах MNIST и CIFAR»

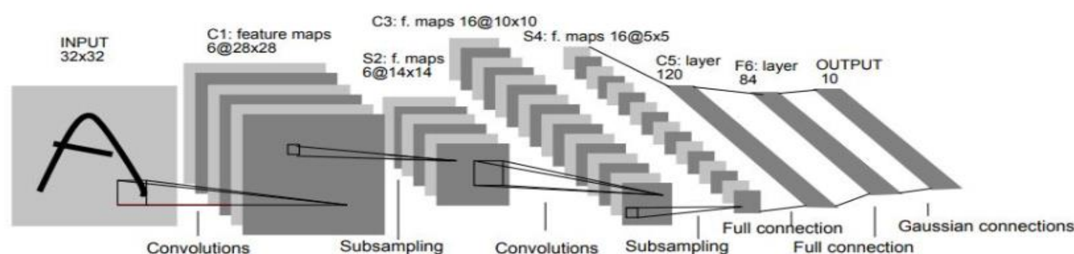
Цель работы – исследование различных архитектур сверточных нейронных сетей на примере задачи классификации изображений.

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 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 №4 мы классифицировали датасет MNIST (рукописные цифры) с помощью сверточной нейронной сети. Исследуем результаты классификации, использовав различные архитектуры сверточной нейронной сети на базе LeNet.

Изменим LeNet, чтобы повысить качество на валидации. Сейчас уже никто не использует активации **тангенсами**, потому что они приводят к затуханию градиента. Нельзя построить действительно глубокую сеть так, чтобы ошибки, посчитанные в конце сети хорошо возвращались к началу сети. Тангенсы или сигмоиды приводят к тому, что сигнал очень быстро затухает.

Во-вторых, сейчас мало кто использует свёртки 5×5. Свёртки 5×5 заменяют на подряд идущие две свёртки 3×3. В свертке 5×5 – 25 весов, а в двух свёртках 3×3 – 18 весов. Весов меньше – меньше переобучения.

В-третьих, сейчас вместо **average** используется везде **max pooling**.

И последний момент – это **батч-нормализация**, которая призвана ускорять обучение. Рассмотрим переменные, которые конфигурируют класс LeNet5 и функцию forward: **activation**, **pooling**, **conv_size** и **use_batch_norm**. Они определяют вид активации (тангенсом или ReLU), вид pooling (average" или "max pooling). Если "conv_size = 5", то идет одна свёртка 5 на 5, либо "3" (две последовательных 3 на 3). Переменная **use_batch_norm** определяет, будет использована батч-нормализация или нет.

Теперь функция forward выглядит следующим образом. Если "conv_size = 5", то используем одну конволюцию 5 на 5, если "conv_size=3", то будем использовать сначала одну конволюцию 3 на 3, и результат её передадим в следующую конволюцию 3 на 3. Далее идет активация (ReLU или тангенс).

Если используем батч-нормализацию, то дополнительно после свёртки вставим слой батч-нормализации **batchnorm1** или **batchnorm2**.

Отметим, что слои батч-нормализации вызываются с помощью **torch.nn.BatchNorm2d**, потому что мы имеем дело с картинками. Если нормализовать некоторый вектор после, например, fully-connected слоя, то следует использовать **torch.nn.BatchNorm1d**. На вход нужно передать **num_features**, то есть то количество каналов, которое имеет картинка или тензор перед батч-нормализацией.

Важный момент, который нужно добавить в процесс обучения, связанный тоже с батч-нормализацией – это некоторый флажок: либо сеть находится в состоянии тренировки (флаг "net.train"), либо сеть находится в состоянии "evaluation", (флаг "net.eval").

Дело в том, что батч-нормализация происходит не в момент вычисления градиентов (не в момент backward), а в момент forward. Каждый раз, когда сеть проходит в forward-направлении, параметры математического ожидания и стандартного отклонения в batch-norm слое заново обучаются/подгоняются. Для того, чтобы этого не происходило во время тестирования, нужно явно указать batch-norm слою, что сеть не обучается, а мы её тестируем, чтобы он не изменял эти параметры математического ожидания и стандартного отклонения.

Были такие случаи, когда готовую, работающую сеть на PyTorch, отдавали заказчику, и через какое-то время она переставала работать, потому что параметры внутри batchnorm-слоя изменялись, а сеть не была проставлена в evaluation-режим.

Для уменьшения потребления памяти GPU используется конструкция (отменяет хранение истории градиентов):

```
with torch.no_grad():
     test_preds = net.forward(X_test)
```

Анализ результатов показывает, что для исследования стоит взять более сложный датасет, где качество будет похуже, но можно увидеть влияние различие добавок на точность классификации и ошибку. Дело в том, что даже классические методы на этом датасете показывают довольно хорошие результаты (в районе 98-99%).

Возьмем действительно сложный датасет, с которым классические методы не справляются. Есть такой датасет – CIFAR-10, он состоит из 32×32 RGB-картинок, разбитых на 10 классов, всего картинок 60 тысяч – 50 тысяч для обучения и 10 тысяч для валидации.

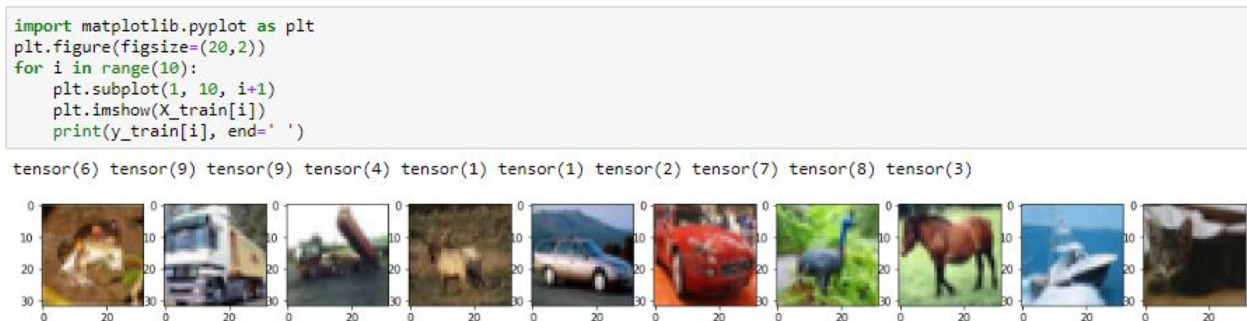
Также существует его увеличенная версия – CIFAR-100 на 100 классов.

Обучим на нём наши сети из предыдущего шага. Код практически не изменится, но необходимо немного изменить загрузку датасета, потому что теперь есть три канала в изображении. Во-вторых, изменится сеть, так как раньше приходили изображения 28×28, а теперь 32×32.

Загрузим этот датасет с помощью библиотеки **torchvision**. Сформируем **CIFAR_train** и **CIFAR_test**. Нужно преобразовать эти датасеты в FloatTensor для картинок и в LongTensor для классов. Видим, что разбивка совпадает с заявленной: 50 000 идёт в train, 10 000 идёт в test.

Если мы посмотрим на максимальное и минимальное значения в картинках, то окажется, что минимальное значение равно 0, а максимальное равно 255. С такими изображениями можно работать, но для удобства отнормируем эти данные – разделим значение каждого пикселя на 255.

Можно визуализировать классы. Например, метка 9 отвечает за "truck", то есть за грузовики.



Ещё одна особенность этого датасета состоит в том, что, как и у обычных картинок, канал "цвет" кодируется в последней размерности. Сначала идёт высота картинки, ширина, а после этого – цвет. Но pytorch требует, чтобы этот канал шёл на первом месте. Нужно реорганизовать размерность тензора таким образом, чтобы цвет шёл на втором месте – после количества картинок в датасете. Это делается с помощью метода "**permute**".

Теперь у датасета будет shape 50000×3×32×32, то есть каналы будут идти перед размерностью изображения.

Далее изменим сеть LeNet таким образом, чтобы она принимала изображения 32×32 и три канала на входе, в **in_chanels** первой конволюции поставим "3" (раньше было "1"). Размерность 32×32 получается за счет паддинга. Больше ничего не меняется – процесс обучения не изменяется, он остался таким же, как и раньше.

Теперь явно есть отличия в результатах. LeNet. Начальный – с тангенсами, без свёрток 3×3, без max pooling и без батч-нормализации получает самые плохие показатели качества. Он обучается примерно на 55% accuracy, то есть он угадывает в 55% процентах случаев. Так как классов 10, это не самое плохое качество. Если бы он угадывал случайно, какое было бы качество?

Мы видим интересную особенность батч-нормализации – сеть с батч-нормализацией учится очень быстро. Уже на пятой эпохе сеть достигает качества, которое без батч-нормализации достигается на 20-й эпохе. Получается ускорение в четыре раза. Однако батч-нормализация и переобучается быстрее. Логично предположить, что если весь процесс обучения ускоряется, то и момент переобучения наступает быстрее.

Более явно это можно увидеть на графиках лосс-функции, в которых очевидно, что переобучение при батч-нормализации наступает уже на пятой эпохе и качество начинает ухудшаться. Почему качество ухудшается так сильно (на графиках сеть всё ещё лучше предсказывает, чем, например, стандартный LeNet с тангенсом)? Дело в том, что переобученные сетки очень уверены в своих ответах. Если не переобученная сеть предсказывает *неправильно* с вероятностью 0.6, то переобученная сеть будет предсказывать *неправильно* с вероятностью 1, и, соответственно, её кросс-энтропия будет огромной.

На этом графике можно увидеть ещё одну контринтуитивную вещь (какую?).

Можно сделать вывод, что все рекомендации являются некоторыми эмпирическими фактами, которые могут не воспроизводиться на конкретной задаче.

Исследование на датасете MNIST

```
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
##
```

```python
class LeNet5(torch.nn.Module):
    def __init__(self,
                 activation='tanh',
                 pooling='avg',
                 conv_size=5,
                 use_batch_norm=False):
        super(LeNet5, self).__init__()

        self.conv_size = conv_size
        self.use_batch_norm = use_batch_norm

        if activation == 'tanh':
            activation_function = torch.nn.Tanh()
        elif activation == 'relu':
            activation_function  = torch.nn.ReLU()
        else:
            raise NotImplementedError

        if pooling == 'avg':
            pooling_layer = torch.nn.AvgPool2d(kernel_size=2, stride=2)
        elif pooling == 'max':
            pooling_layer  = torch.nn.MaxPool2d(kernel_size=2, stride=2)
        else:
            raise NotImplementedError

        if conv_size == 5:
            self.conv1 = torch.nn.Conv2d(
                in_channels=1, out_channels=6, kernel_size=5, padding=2)
        elif conv_size == 3:
            self.conv1_1 = torch.nn.Conv2d(
                in_channels=1, out_channels=6, kernel_size=3, padding=1)
            self.conv1_2 = torch.nn.Conv2d(
                in_channels=6, out_channels=6, kernel_size=3, padding=1)
        else:
            raise NotImplementedError

        self.act1 = activation_function
        self.bn1 = torch.nn.BatchNorm2d(num_features=6)
        self.pool1 = pooling_layer

        if conv_size == 5:
            self.conv2 = self.conv2 = torch.nn.Conv2d(
                in_channels=6, out_channels=16, kernel_size=5, padding=0)
        elif conv_size == 3:
            self.conv2_1 = torch.nn.Conv2d(
                in_channels=6, out_channels=16, kernel_size=3, padding=0)
            self.conv2_2 = torch.nn.Conv2d(
                in_channels=16, out_channels=16, kernel_size=3, padding=0)
        else:
            raise NotImplementedError

        self.act2 = activation_function
        self.bn2 = torch.nn.BatchNorm2d(num_features=16)
        self.pool2 = pooling_layer

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

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

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

    def forward(self, x):
        if self.conv_size == 5:
            x = self.conv1(x)
        elif self.conv_size == 3:
            x = self.conv1_2(self.conv1_1(x))
        x = self.act1(x)
        if self.use_batch_norm:
            x = self.bn1(x)
        x = self.pool1(x)

        if self.conv_size == 5:
            x = self.conv2(x)
        elif self.conv_size == 3:
            x = self.conv2_2(self.conv2_1(x))
        x = self.act2(x)
        if self.use_batch_norm:
            x = self.bn2(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
```

##

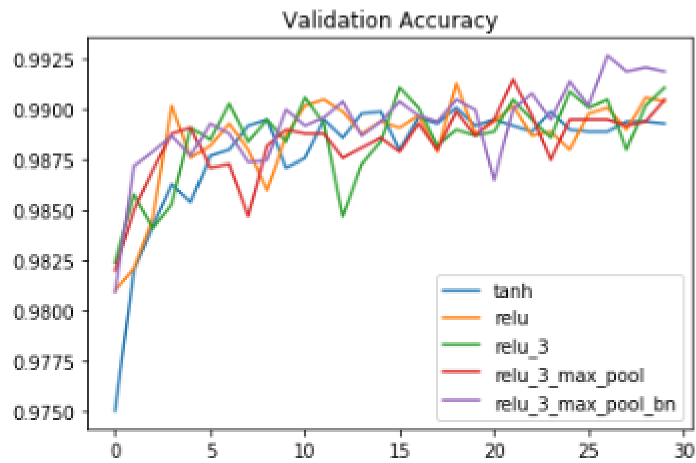
```
def train(net, X_train, y_train, X_test, y_test):
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    net = net.to(device)
    loss = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=1.0e-3)

    batch_size = 100

    test_accuracy_history = []
    test_loss_history = []

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

    for epoch in range(30):
        order = np.random.permutation(len(X_train))
        for start_index in range(0, len(X_train), batch_size):
            optimizer.zero_grad()
            net.train()

            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 = net.forward(X_batch)
```

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

            optimizer.step()

        net.eval()
        with torch.no_grad():
             test_preds = net.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)
    print('---------------')
    return test_accuracy_history, test_loss_history

accuracies = {}
losses = {}


accuracies['tanh'], losses['tanh'] = \
    train(LeNet5(activation='tanh', conv_size=5),
          X_train, y_train, X_test, y_test)

accuracies['relu'], losses['relu'] = \
    train(LeNet5(activation='relu', conv_size=5),
          X_train, y_train, X_test, y_test)

accuracies['relu_3'], losses['relu_3'] = \
    train(LeNet5(activation='relu', conv_size=3),
          X_train, y_train, X_test, y_test)

accuracies['relu_3_max_pool'], losses['relu_3_max_pool'] = \
    train(LeNet5(activation='relu', conv_size=3, pooling='max'),
          X_train, y_train, X_test, y_test)

accuracies['relu_3_max_pool_bn'], losses['relu_3_max_pool_bn'] = \
    train(LeNet5(activation='relu', conv_size=3, pooling='max', use_batch_n
orm=True),
          X_train, y_train, X_test, y_test)
```
##

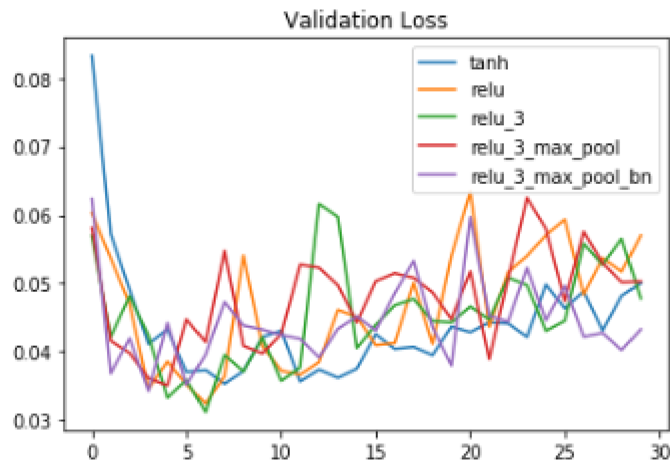
```
for experiment_id in accuracies.keys():
    plt.plot(accuracies[experiment_id], label=experiment_id)
plt.legend()
plt.title('Validation Accuracy');
```
##

```
for experiment_id in losses.keys():
    plt.plot(losses[experiment_id], label=experiment_id)
plt.legend()
plt.title('Validation Loss');
```
##

```python
for experiment_id in accuracies.keys():
    plt.plot(accuracies[experiment_id], label=experiment_id)
plt.legend()
plt.title('Validation Accuracy');
```




```python
for experiment_id in losses.keys():
    plt.plot(losses[experiment_id], label=experiment_id)
plt.legend()
plt.title('Validation Loss');
```



Исследование на датасете CIFAR

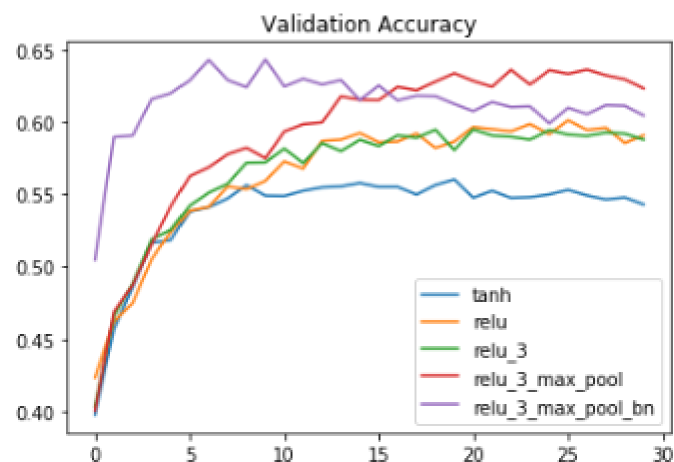
```
import torch
import random
import numpy as np

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

import torchvision.datasets
##

CIFAR_train = torchvision.datasets.CIFAR10('./', download=True, train=True)
CIFAR_test = torchvision.datasets.CIFAR10('./', download=True, train=False)
##

X_train = torch.FloatTensor(CIFAR_train.data) y_train =
torch.LongTensor(CIFAR_train.targets) X_test =
torch.FloatTensor(CIFAR_test.data) y_test =
torch.LongTensor(CIFAR_test.targets)
##

len(y_train), len(y_test)
##
X_train.min(), X_train.max()
##
X_train /= 255.
X_test /= 255.
##
CIFAR_train.classes
##

import matplotlib.pyplot as plt
plt.figure(figsize=(20,2))
for i in range(10):
    plt.subplot(1, 10, i+1)
    plt.imshow(X_train[i])
    print(y_train[i], end=' ')
##
X_train.shape, y_train.shape
##

X_train = X_train.permute(0, 3, 1, 2)
X_test = X_test.permute(0, 3, 1, 2)
##
X_train.shape

##
```

```python
class LeNet5(torch.nn.Module):
    def __init__(self,
                 activation='tanh',
                 pooling='avg',
                 conv_size=5,
                 use_batch_norm=False):
        super(LeNet5, self).__init__()

        self.conv_size = conv_size
        self.use_batch_norm = use_batch_norm

        if activation == 'tanh':
            activation_function = torch.nn.Tanh()
        elif activation == 'relu':
            activation_function  = torch.nn.ReLU()
        else:
            raise NotImplementedError

        if pooling == 'avg':
            pooling_layer = torch.nn.AvgPool2d(kernel_size=2, stride=2)
        elif pooling == 'max':
            pooling_layer  = torch.nn.MaxPool2d(kernel_size=2, stride=2)
        else:
            raise NotImplementedError

        if conv_size == 5:
            self.conv1 = torch.nn.Conv2d(
                in_channels=3, out_channels=6, kernel_size=5, padding=0)
        elif conv_size == 3:
            self.conv1_1 = torch.nn.Conv2d(
                in_channels=3, out_channels=6, kernel_size=3, padding=0)
            self.conv1_2 = torch.nn.Conv2d(
                in_channels=6, out_channels=6, kernel_size=3, padding=0)
        else:
            raise NotImplementedError

        self.act1 = activation_function
        self.bn1 = torch.nn.BatchNorm2d(num_features=6)
        self.pool1 = pooling_layer

        if conv_size == 5:
            self.conv2 = self.conv2 = torch.nn.Conv2d(
                in_channels=6, out_channels=16, kernel_size=5, padding=0)
        elif conv_size == 3:
            self.conv2_1 = torch.nn.Conv2d(
                in_channels=6, out_channels=16, kernel_size=3, padding=0)
            self.conv2_2 = torch.nn.Conv2d(
                in_channels=16, out_channels=16, kernel_size=3, padding=0)
        else:
            raise NotImplementedError

        self.act2 = activation_function
        self.bn2 = torch.nn.BatchNorm2d(num_features=16)
        self.pool2 = pooling_layer

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

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

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

    def forward(self, x):
        if self.conv_size == 5:
            x = self.conv1(x)
        elif self.conv_size == 3:
            x = self.conv1_2(self.conv1_1(x))
        x = self.act1(x)
        if self.use_batch_norm:
            x = self.bn1(x)
        x = self.pool1(x)

        if self.conv_size == 5:
            x = self.conv2(x)
        elif self.conv_size == 3:
            x = self.conv2_2(self.conv2_1(x))
        x = self.act2(x)
        if self.use_batch_norm:
            x = self.bn2(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
```

```python
##
def train(net, X_train, y_train, X_test, y_test):
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    net = net.to(device)
    loss = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(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 in range(30):
        order = np.random.permutation(len(X_train))
        for start_index in range(0, len(X_train), batch_size):
            optimizer.zero_grad()
            net.train()

            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 = net.forward(X_batch)

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

```
            optimizer.step()

            X_batch

        net.eval()
        with torch.no_grad():
             test_preds = net.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)
    del net
    return test_accuracy_history, test_loss_history

accuracies = {}
losses = {}

accuracies['tanh'], losses['tanh'] = \
    train(LeNet5(activation='tanh', conv_size=5),
          X_train, y_train, X_test, y_test)
accuracies['relu'], losses['relu'] = \
    train(LeNet5(activation='relu', conv_size=5),
          X_train, y_train, X_test, y_test)
accuracies['relu_3'], losses['relu_3'] = \
    train(LeNet5(activation='relu', conv_size=3),
          X_train, y_train, X_test, y_test)
accuracies['relu_3_max_pool'], losses['relu_3_max_pool'] = \
    train(LeNet5(activation='relu', conv_size=3, pooling='max'),
          X_train, y_train, X_test, y_test)
accuracies['relu_3_max_pool_bn'], losses['relu_3_max_pool_bn'] = \
    train(LeNet5(activation='relu', conv_size=3, pooling='max', use_batch_n
orm=True),
          X_train, y_train, X_test, y_test)
```



Задание на лабораторную работу

1. Изучить понятия: **батч-нормализация, переобучение нейронной сети.**

   Примечание. В машинном обучении довольно известным фактом является то, что нормализованные (или стандартизованные) данные приводят к более быстрому обучению модели.

2. Исследовать нейронную сеть LeNet при различных параметрах (5 вариантов) на датасете MNIST. Объяснить результаты экспериментов.
3. Исследовать нейронную сеть LeNet при различных параметрах (5 вариантов) на датасете CIFAR. Построить графики **validation loss**. Объяснить результаты экспериментов. Каким образом можно избежать переобучения сети?

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/