МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра ВТ

ОТЧЕТ

по лабораторной работе № 1

по дисциплине «Нейронные сети»

CNN

Студент гр. 9308 _____ Хамитов А.К.

Преподаватель _____ Неверов Е.А.

Санкт-Петербург

2023

Цель работы

Задание 1: Подготовка данных

В задании необходимо продемонстрировать методы подготовки и предварительной обработки данных, таких как увеличение и нормализация данных.

Задание 2: Архитектура CNN и трансферное обучение

В этом задании необходимо написать и уметь объяснить как работает разработанная архитектура.

Данное задание делится на 2 части:

1. Разработка собственной архитектуры (и последующее обучение с 0) – 3-5 различных архитектур с разной комбинацией слоев/функций активации;
2. Трансферное обучение (*transfer learning*).

Задание 3: Обучение CNN

В данном задании необходимо продемонстрировать знания, как обучать CNN, используя функции потерь, оптимизаторы и методы регуляризации.

Задание 4: Настройка гиперпараметров и выбор модели

В этом задании студенты должны объяснить, как настраивать гиперпараметры в CNN и как выполнять выбор модели.

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:
```

```
get_ipython().system(' pip install -q kaggle')
from google.colab import files
files.upload()
get_ipython().system(' mkdir ~/.kaggle')
get_ipython().system(' cp kaggle.json ~/.kaggle/')
get_ipython().system(' kaggle datasets list')
```

```
# In[2]:
```

```
get_ipython().system(' kaggle datasets download -d vencerlanz09/pharmaceutical-drugs-and-vitamins-synthetic-images')
```

```
# In[3]:
```

```
get_ipython().system(' unzip pharmaceutical-drugs-and-vitamins-synthetic-images.zip')
```

```
# # CNN
```

```
# In[4]:
```

```
get_ipython().system(" rmdir /content/'Drug Vision'/'Data Combined'/.ipynb_checkpoints")
```

```
# ! rmdir /content/'Drug Vision'/'Data Combined'/'Neozep'
# ! rmdir /content/'Drug Vision'/'Data Combined'/'Medicol'
# ! rmdir /content/'Drug Vision'/'Data Combined'/'Kremil S'
# ! rmdir /content/'Drug Vision'/'Data Combined'/'Fish Oil'
# ! rmdir /content/'Drug Vision'/'Data Combined'/'Decolgen'
# ! rmdir /content/'Drug Vision'/'Data Combined'/'DayZinc'
```

```
# In[8]:
```

```
import torch
import torchvision
import torchvision.transforms as transforms
import os
```

```
# In[9]:
```

```
classes = os.listdir('./Drug Vision/Data Combined')
num_classes = len(classes)
classes
```

```
# In[10]:
```

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

```
batch_size = 16
```

```
dataset = torchvision.datasets.ImageFolder(root='./Drug Vision/Data Combined', transform=transform)
```

```
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size, test_size])
```

```
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)
```

```
classes = os.listdir('./Drug Vision/Data Combined')
```

```
# In[11]:
```

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

```
# функция для показа изображения
def imshow(img):
    img = img  / 2  + 0.5
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

```
# получаем несколько случайных обучающих изображений
dataiter = iter(trainloader)
images, labels = next(dataiter)
```

```
# показать изображения
imshow(torchvision.utils.make_grid(images))
# показать лейблы изображений
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```

```
# In[12]:
```

```
import torch.nn as nn
import torch.nn.functional as F
```

```
# Инициализация модели
class Net(nn.Module):
    def __init__(self):
        super().__init__()
```

```
        self.conv1 = nn.Conv2d(3, 6, 5) # (in_channels, out_channels, kernel_size) Применяет 2D-свертку к входному сигналу, состоящему из нескольких входных плоскостей.
        self.pool = nn.MaxPool2d(2, 2) # (kernel_size, stride) Применяет MaxPool2D-объединение к входному сигналу, состоящему из нескольких входных плоскостей.
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 72 * 72, 120 )# (in_features , out_features) Применяет линейное преобразование к входящим данным
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 4)
```

```
    # Это forward функция, которая определяет структуру сети.
    # Здесь мы принимаем только один вход, но можно использовать больше.
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) #(input, start_dim) Сглаживает input путем преобразования его в одномерный тензор.
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return torch.log_softmax(x, dim=1)
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net = Net().to(device)
```

```
# In[13]:
```

```
class Regulator(nn.Module):
    def __init__(self,in_channels,out_channels,stride=1,kernel_size=3,padding=1,bias=False):
        super(Regulator,self).__init__()
        self.cnn1 = nn.Sequential(
            nn.Conv2d(in_channels,out_channels,kernel_size,stride,padding,bias=False),
            nn.ReLU(True)
        )
        self.cnn2 = nn.Sequential(
            nn.Conv2d(out_channels,out_channels,kernel_size,1,padding,bias=False),
        )

    def forward(self,x):
        x = self.cnn1(x)
        x = self.cnn2(x)
        x = nn.ReLU(True)(x)
        return x
```

```
class Net2(nn.Module):
    def __init__(self):
        super(Net2,self).__init__()

        self.block1 = nn.Sequential(
            nn.Conv2d(3,64,kernel_size=2,stride=2,padding=3,bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True)
        )

        self.block2 = nn.Sequential(
            nn.MaxPool2d(1,1),
```

```
            Regulator(64,64),
        )

        self.block3 = nn.Sequential(
            Regulator(64,128,2),
        )

        self.block4 = nn.Sequential(
            Regulator(128,256,2),
        )
        self.block5 = nn.Sequential(
            Regulator(256,512,2),
        )
        self.maxpool = nn.MaxPool2d(2, 2)
        # vowel_diacritic
        self.fc1 = nn.Linear(512*10*10, num_classes)

    def forward(self,x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)
        x = self.block5(x)
        x = self.maxpool(x)
        # print(x.shape)
        x = x.view(x.size(0),-1)
        x = self.fc1(x)
        return torch.log_softmax(x, dim=1)
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net2 = Net2().to(device)
```

```
# In[30]:
```

```
class Regulator(nn.Module):
    def __init__(self,in_channels,out_channels,stride=1,kernel_size=3,padding=1,bias=False):
        super(Regulator,self).__init__()
        self.cnn1 = nn.Sequential(
            nn.Conv2d(in_channels,out_channels,kernel_size,stride,padding,bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(True)
        )
        self.cnn2 = nn.Sequential(
            nn.Conv2d(out_channels,out_channels,kernel_size,1,padding,bias=False),
            nn.BatchNorm2d(out_channels)
        )
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels,out_channels,kernel_size=1,stride=stride,bias=False),
                nn.BatchNorm2d(out_channels)
            )
        else:
            self.shortcut = nn.Sequential()

    def forward(self,x):
        residual = x
```

```
        x = self.cnn1(x)
        x = self.cnn2(x)
        x += self.shortcut(residual)
        x = nn.ReLU(True)(x)
        return x
```

```
class Net3(nn.Module):
    def __init__(self):
        super(Net3,self).__init__()

        self.block1 = nn.Sequential(
            nn.Conv2d(3,64,kernel_size=2,stride=2,padding=3,bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True)
        )

        self.block2 = nn.Sequential(
            nn.MaxPool2d(1,1),
            Regulator(64,64),
        )

        self.block3 = nn.Sequential(
            Regulator(64,128,2),
        )

        self.block4 = nn.Sequential(
            Regulator(128,256,2),
        )
        self.block5 = nn.Sequential(
            Regulator(256,512,2),
        )
        self.maxpool = nn.MaxPool2d(2, 2)
        # vowel_diacritic
        self.fc1 = nn.Linear(512*10*10, num_classes)

    def forward(self,x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)
        x = self.block5(x)
        x = self.maxpool(x)
        # print(x.shape)
        x = x.view(x.size(0),-1)
        x = self.fc1(x)
        return torch.log_softmax(x, dim=1)
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net3 = Net3().to(device)
```

```
# In[31]:
```

```
device
```

```
# In[15]:
```

```
class Regulator(nn.Module):
    def __init__(self,in_channels,out_channels,stride=1,kernel_size=3,padding=1,bias=False):
        super(Regulator,self).__init__()
        self.cnn1 = nn.Sequential(
            nn.Conv2d(in_channels,out_channels,kernel_size,stride,padding,bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(True)
        )
        self.cnn2 = nn.Sequential(
            nn.Conv2d(out_channels,out_channels,kernel_size,1,padding,bias=False),
            nn.BatchNorm2d(out_channels)
        )
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels,out_channels,kernel_size=1,stride=stride,bias=False),
                nn.BatchNorm2d(out_channels)
            )
        else:
            self.shortcut = nn.Sequential()

    def forward(self,x):
        residual = x
        x = self.cnn1(x)
        x = self.cnn2(x)
        x += self.shortcut(residual)
        x = nn.ReLU(True)(x)
        return x
```

```
class Net4(nn.Module):
    def __init__(self):
        super(Net4,self).__init__()

        self.block1 = nn.Sequential(
            nn.Conv2d(3,64,kernel_size=2,stride=2,padding=3,bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True)
        )
        self.droput = nn.Dropout(p=0.5)
        self.block2 = nn.Sequential(
            nn.MaxPool2d(1,1),
            Regulator(64,64),
        )
        self.droput = nn.Dropout(p=0.5)
        self.block3 = nn.Sequential(
            Regulator(64,128,2),
        )
        self.droput = nn.Dropout(p=0.5)
        self.block4 = nn.Sequential(
            Regulator(128,256,2),
        )
        self.block5 = nn.Sequential(
            Regulator(256,512,2),
        )
        self.maxpool = nn.MaxPool2d(2, 2)
        # vowel_diacritic
        self.fc1 = nn.Linear(512*10*10, num_classes)
```

```
    def forward(self,x):
        x = self.block1(x)
        x = self.droput(x)
        x = self.block2(x)
        x = self.droput(x)
        x = self.block3(x)
        x = self.droput(x)
        x = self.block4(x)
        x = self.droput(x)
        x = self.block5(x)
        x = self.maxpool(x)
        # print(x.shape)
        x = x.view(x.size(0),-1)
        x = self.fc1(x)
        return torch.log_softmax(x, dim=1)
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net4 = Net4().to(device)
```

```
# In[16]:
```

```
batch = next(iter(trainloader))
```

```
Net4().forward(torch.FloatTensor(batch[0]))
```

```
# Коэффициент скорости обучения - это гиперпараметр, определяющий порядок того, как мы будем корректировать
наши весы с учётом функции потерь в градиентном спуске. Чем ниже величина, тем медленнее мы движемся по
наклонной. Хотя при использовании низкого коэффициента скорости обучения мы можем получить положительный
эффект в том смысле, чтобы не пропустить ни одного локального минимума, — это также может означать, что нам
придётся затратить много времени на сходимость, особенно если мы попали в область плато.
#
# Импульс (momentum) в нейронных сетях — это вариант стохастического градиентного спуска . Он заменяет
градиент импульсом , который представляет собой совокупность градиентов, как очень хорошо объяснено
[здесь](https://towardsdatascience.com/10-gradient-descent-optimisation-algorithms-86989510b5e9).
#
#
#
```

```
# In[17]:
```

```
losses = {
    0: [],
    1: [],
    2: [],
    3: []
}
```

```
acc = {
    0: [],
    1: [],
    2: [],
    3: []
}
```

```
# In[18]:
```

```
from torch.optim import Adam
```

```
#Критерии полезны для обучения нейронной сети. Учитывая входные данные и цель, они вычисляют градиент в
соответствии с заданной функцией потерь
criterion = nn.CrossEntropyLoss()
```

```
# In[36]:
```

```
#torch.optim - это пакет, реализующий различные алгоритмы оптимизации. Наиболее часто используемые методы уже
поддерживаются, а интерфейс достаточно общий, так что более сложные методы могут быть также легко
интегрированы в будущем.
optimizer = Adam(net.parameters(), lr=0.001, weight_decay=0.0001)
```

```
for epoch in range(40):  # многократное прохождение по набору данных
```

```
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # получаем входные данные; данные - это список [inputs, labels].
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
```

```
        # обнуляем градиенты параметров
        optimizer.zero_grad()
```

```
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
        # вывести статистику обучения
        running_loss += loss.item()
        if i % 100 == 99:
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss}')
            losses[0].append(running_loss)
            running_loss = 0.0

        correct = 0
        total = 0
        if i % 200 == 199:
          with torch.no_grad():
            for data in testloader:
                images, labels = data
                # рассчитываем выходные данные, пропуская изображения через сеть
                outputs = net(images.to(device))
                # класс с наибольшей мощностью - это то, что мы выбираем в качестве предсказания
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted.to(device) == labels.to(device)).sum().item()
            print(f'Net 1: {100 * correct // total} %')
```

```
            acc[0].append(correct // total)
```

```
print('Finished Training')
```

```
# In[37]:
```

```
#torch.optim - это пакет, реализующий различные алгоритмы оптимизации. Наиболее часто используемые методы уже поддерживаются, а интерфейс достаточно общий, так что более сложные методы могут быть также легко интегрированы в будущем.
optimizer = Adam(net2.parameters(), lr=0.001, weight_decay=0.0001)
```

```
for epoch in range(40):  # многократное прохождение по набору данных
```

```
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # получаем входные данные; данные - это список [inputs, labels].
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
```

```
        # обнуляем градиенты параметров
        optimizer.zero_grad()
```

```
        # forward + backward + optimize
        outputs = net2(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
        # вывести статистику обучения
        running_loss += loss.item()
        if i % 100 == 99:    # вывести каждые 500 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss}')
            losses[1].append(running_loss)
            running_loss = 0.0

        correct = 0
        total = 0
        if i % 200 == 199:
          with torch.no_grad():
            for data in testloader:
                images, labels = data
                # рассчитываем выходные данные, пропуская изображения через сеть
                outputs = net2(images.to(device))
                # класс с наибольшей мощностью - это то, что мы выбираем в качестве предсказания
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted.to(device) == labels.to(device)).sum().item()
            print(f'Net 2: {100 * correct // total} %')
            acc[1].append(correct // total)
```

```
print('Finished Training')
```

```
# In[ ]:
```

```
#torch.optim - это пакет, реализующий различные алгоритмы оптимизации. Наиболее часто используемые методы уже
поддерживаются, а интерфейс достаточно общий, так что более сложные методы могут быть также легко
интегрированы в будущем.
optimizer = Adam(net3.parameters(), lr=0.001, weight_decay=0.0001)
```

```
for epoch in range(40):  # многократное прохождение по набору данных
```

```
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # получаем входные данные; данные - это список [inputs, labels].
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
```

```
        # обнуляем градиенты параметров
        optimizer.zero_grad()
```

```
        # forward + backward + optimize
        outputs = net3(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
        # вывести статистику обучения
        running_loss += loss.item()
        if i % 100 == 99:    # вывести каждые 500 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss}')
            losses[2].append(running_loss)
            running_loss = 0.0

        correct = 0
        total = 0
        if i % 200 == 199:
          with torch.no_grad():
            for data in testloader:
                images, labels = data
                # рассчитываем выходные данные, пропуская изображения через сеть
                outputs = net3(images.to(device))
                # класс с наибольшей мощностью - это то, что мы выбираем в качестве предсказания
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted.to(device) == labels.to(device)).sum().item()
            print(f'Net 3: {100 * correct // total} %')
            acc[2].append(correct // total)
```

```
print('Finished Training')
```

```
# In[ ]:
```

```
#torch.optim - это пакет, реализующий различные алгоритмы оптимизации. Наиболее часто используемые методы уже
поддерживаются, а интерфейс достаточно общий, так что более сложные методы могут быть также легко
интегрированы в будущем.
optimizer = Adam(net4.parameters(), lr=0.001, weight_decay=0.0001)
```

```
for epoch in range(40):  # многократное прохождение по набору данных
```

```
    running_loss = 0.0
```

```
    for i, data in enumerate(trainloader, 0):
        # получаем входные данные; данные - это список [inputs, labels].
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
```

```
        # обнуляем градиенты параметров
        optimizer.zero_grad()
```

```
        # forward + backward + optimize
        outputs = net4(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
        # вывести статистику обучения
        running_loss += loss.item()
        if i % 100 == 99:    # вывести каждые 500 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss}')
            losses[3].append(running_loss)
            running_loss = 0.0

        correct = 0
        total = 0
        if i % 200 == 199:
          with torch.no_grad():
            for data in testloader:
                images, labels = data
                # рассчитываем выходные данные, пропуская изображения через сеть
                outputs = net4(images.to(device))
                # класс с наибольшей мощностью - это то, что мы выбираем в качестве предсказания
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted.to(device) == labels.to(device)).sum().item()
            print(f'Net 4: {100 * correct // total} %')
            acc[3].append(correct // total)
```

```
print('Finished Training')
```

```
# In[77]:
```

```
#Сохранение нашей модели
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
torch.save(net2.state_dict(), PATH)
torch.save(net3.state_dict(), PATH)
torch.save(net4.state_dict(), PATH)
```

```
# In[78]:
```

```
dataiter = iter(testloader)
images, labels = next(dataiter)
```

```
# вывод изображений
imshow(torchvision.utils.make_grid(images))
```

```
print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```

```
# In[79]:
```

```
net = Net()
net2 = Net2()
net3 = Net3()
net4 = Net4()
net.load_state_dict(torch.load(PATH))
nets = [net, net2, net3, net4]
```

```
# In[80]:
```

```
outputs = net(images)
```

```
# In[81]:
```

```
correct = 0
total = 0
numof = -1
# поскольку мы не обучаемся, нам не нужно вычислять градиенты для наших выходов
with torch.no_grad():
    for net_elem in nets:
      for data in testloader:
          images, labels = data
          # рассчитываем выходные данные, пропуская изображения через сеть
          outputs = net_elem(images)
          # класс с наибольшей мощностью - это то, что мы выбираем в качестве предсказания
          _, predicted = torch.max(outputs.data, 1)
          total += labels.size(0)
          correct += (predicted == labels).sum().item()
      numof += 1
      print(f'Net {numof}: {100 * correct // total} %')
```

```
# In[83]:
```

```
# Подготовка
```

```
numof = -1
for net_elem in nets:
  numof += 1
  print(f'Net {numof}')
  correct_pred = {classname: 0 for classname in classes}
  total_pred = {classname: 0 for classname in classes}
  with torch.no_grad():
      for data in testloader:
          images, labels = data
          outputs = net_elem(images)
          _, predictions = torch.max(outputs, 1)
          # собираем правильные прогнозы для каждого класса
```

```
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1
```

```
# Выводим точность на каждом классе
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```