Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа искусственного интеллекта
Дисциплина «Машинное обучение»

Методические указания к практическим работам на тему:

«Vision Transformer»

СОДЕРЖАНИЕ

1 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Transformer

Трансформер (англ. Transformer) — архитектура глубоких нейронных сетей, представленная в 2017 году исследователями из Google Brain.

Архитектура трансформера состоит из кодировщика и декодировщика. Кодировщик получает на вход векторизованую последовательность с позиционной информацией. Декодировщик получает на вход часть этой последовательности и выход кодировщика.

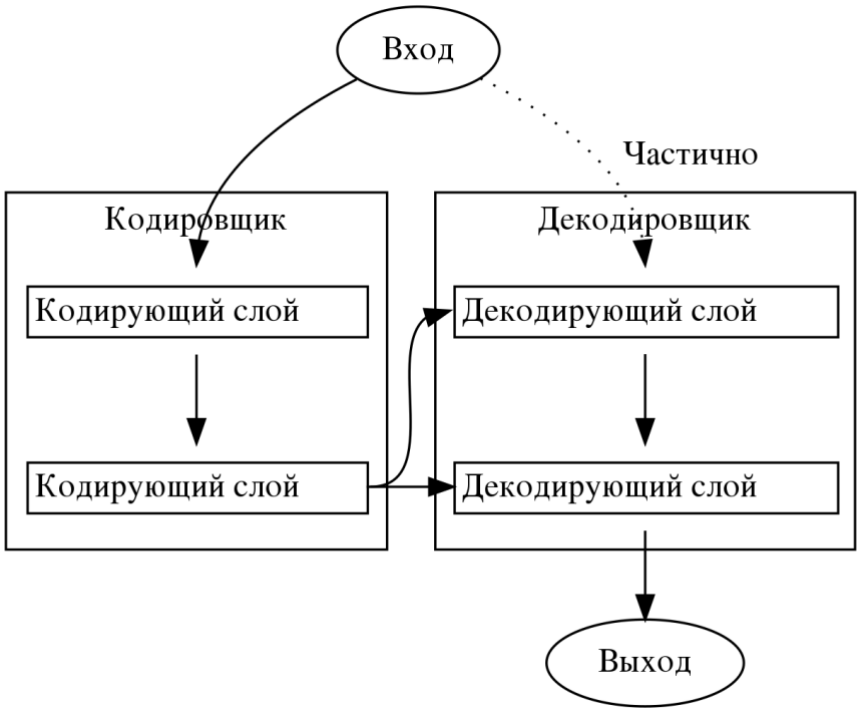

Рисунок 1 – Архитектура трансформера

Каждый кодировщик состоит из механизма самовнимания (вход из предыдущего слоя) и нейронной сети с прямой связью (вход из механизма самовнимания). Каждый декодировщик состоит из механизма самовнимания (вход из предыдущего слоя), механизма внимания к результатам кодирования (вход из механизма самовнимания и кодировщика) и нейронной сети с прямой связью (вход из механизма внимания).

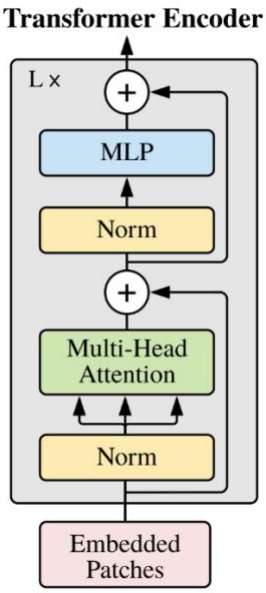

Рисунок 2 – Кодировщик

1.2 Vision Transformer

Transformer, применяемый непосредственно к последовательностям. фрагментов изображений, может хорошо выполнять задачи классификации изображений.

Vision Transformer (ViT) — это преобразователь, предназначенный для решения задач машинного зрения, таких как распознавание изображений.

Каждое изображение разбивается на фрагменты и представляет собой последовательность линейных эмбедингов этих фрагментов. Данные эмбэдинги являются входными данными для трансформера. Патчи изображений обрабатываются так же, как токены (слова) в приложении NLP.

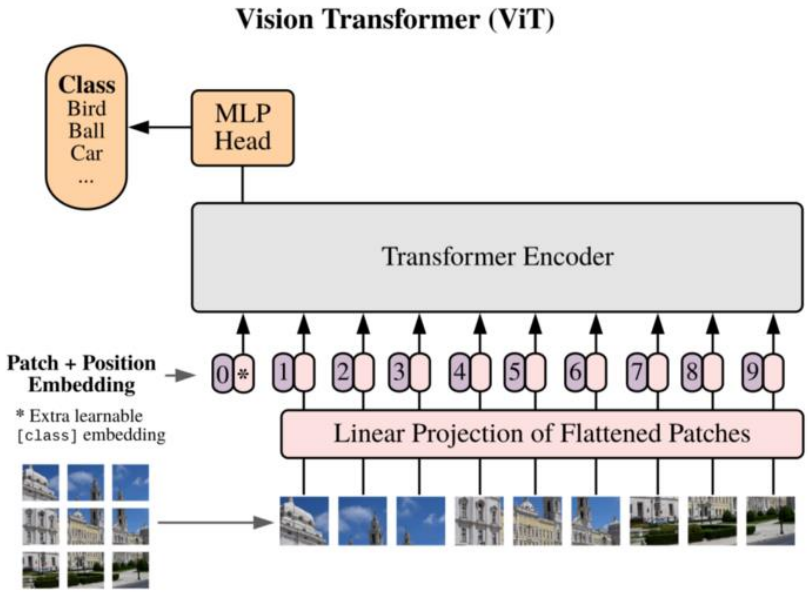

Рисунок 3 – Vision Transformer

Можно видеть, как входное изображение разбивается на патчи 16x16, которые затем преобразуются в эмбединги с использованием обычного полносвязного слоя, а перед ними – совокупность специального токена cls и позиционного встраивания. Результирующий тензор передается в стандартный кодировщик Transformer и, наконец, в MLP Head для классификации.

Трансформеры изначально лишены присущих CNN индуктивных смещений, таких как локальность, и плохо обобщают, когда обучаются на недостаточном количестве данных. Тем не менее, они достигают или превосходят уровень CNN в нескольких тестах распознавания изображений при обучении на больших наборах данных.

2 РЕАЛИЗАЦИЯ VISION TRANSFORMER

2.1 Инициализация

В качестве языка программирования для реализации был выбран Python. Разработка велась в Pytorch.

Сначала, установим необходимые пакеты:

```
!pip install einops
!pip install torch torchvision
!pip install torchsummary
!pip install torchvision --user
```

Далее, импортируем библиотеки:

```
import torch
import torch.nn.functional as F
from torch import Tensor, nn
from torchsummary import summary
from torchvision.transforms import Compose, Resize, ToTensor

from einops import rearrange, reduce, repeat
from einops.layers.torch import Rearrange, Reduce
from PIL import Image
import matplotlib.pyplot as plt
```

Будем использовать некоторое изображение для проверки:

```
import matplotlib.pyplot as plt
from PIL import Image
#изображение для проверки
img = Image.open('my_penguin.jpg')

fig = plt.figure()
plt.imshow(img)
plt.show()
```

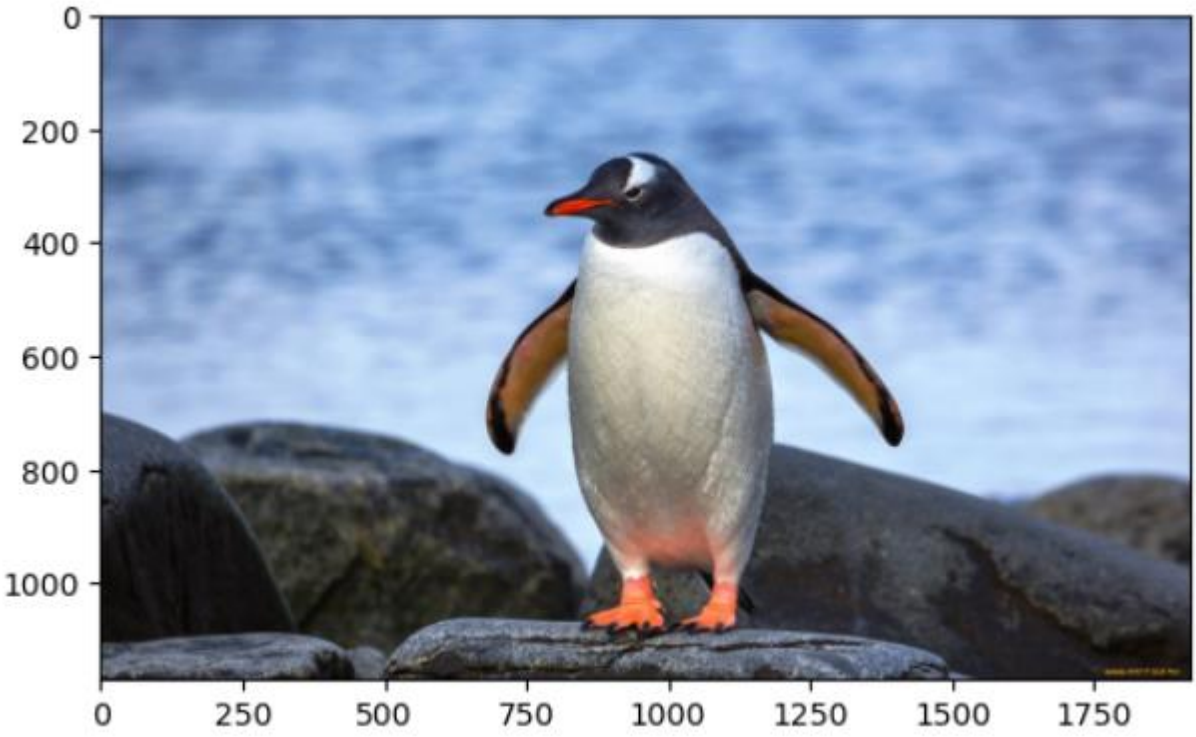

Рисунок 4 – Изображение для проверки

Данное изображение имеет размер 1920x1120 пикселей. Необходимо привести изображение к нужному размеру 224x224 пикселя. **Препроцессинг:**

```
#препроцессинг
trans = Compose([
    Resize((224, 224)),
    ToTensor(),
])

x = trans(img)
x = x.unsqueeze(0)
print(x.shape)
torch.Size([1,3,224,224])
```

```
torch.Size([1, 3, 224, 224])
```

2.2 Компоненты архитектуры

Далее, описываем класс PatchEmbedding. Здесь, представлена функция __init__, которая реализует разбиение изображения на патчи длиной patch_size и высотой patch_size, а затем реализует их сглаживание. Функция forward, реализует добавление позиций данных эмбедингов.

#создается последовательность эмбедингов, которые являются входом кодировщика

```
#создается последовательность эмбедингов, которые являются входом
кодировщика
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 16,
emb_size: int = 768,img_size: int=224):
        self.patch_size = patch_size
        super().__init__()
        self.projection = nn.Sequential(
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size,
stride=patch_size),
            Rearrange('b e (h) (w) -> b (h w) e'),
        ) # разбивает изображение на патчи s1xs2, а затем сглаживает их
        self.cls_token = nn.Parameter(torch.randn(1,1, emb_size))
        #self.positions = nn.Parameter(torch.randn(1, x.shape[1],
self.embedding_dim))
        self.positions = nn.Parameter(torch.randn((img_size // patch_size)
**2 + 1, emb_size))


    def forward(self, x: Tensor) -> Tensor:
        b, _, _, _ = x.shape
        x = self.projection(x)
        cls_tokens = repeat(self.cls_token, '() n e -> b n e', b=b)
        x = torch.cat([cls_tokens, x], dim=1) #добавляет эмбединги позиций
        x += self.positions
        return x
```

Механизм внимания принимает три входа: запросы (queries), ключи (keys) и значения (values). Затем он вычисляет матрицу внимания с помощью запросов и ключей.

Мы будем реализовывать механизм внимания с несколькими heads, поэтому вычисления будут разделены на несколько heads с меньшим размером входных данных.

Основная концепция состоит в использовании произведения между запросами и ключами, чтобы понять, насколько каждый элемент в последовательности важен для остальных. Такая информация позже используется для масштабирования значений.

Реализуем механизм внимания:

```
#механизм внимания
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_size: int = 768, num_heads: int = 8, dropout:
float = 0):
```

```
        super().__init__()
        self.emb_size = emb_size
        self.num_heads = num_heads
        self.qkv = nn.Linear(emb_size, emb_size * 3) # queries, keys and
values matrix
        self.att_drop = nn.Dropout(dropout)
        self.projection = nn.Linear(emb_size, emb_size)

    def forward(self, x : Tensor, mask: Tensor = None) -> Tensor:
        # split keys, queries and values in num_heads
        qkv = rearrange(self.qkv(x), "b n (h d qkv) -> (qkv) b h n d",
h=self.num_heads, qkv=3)
        queries, keys, values = qkv[0], qkv[1], qkv[2]
        # sum up over the last axis
        energy = torch.einsum('bhqd, bhkd -> bhqk', queries, keys) #
batch, num_heads, query_len, key_len

        if mask is not None:
            fill_value = torch.finfo(torch.float32).min
            energy.mask_fill(~mask, fill_value)

        scaling = self.emb_size ** (1/2)

        att = F.softmax(energy, dim=-1) / scaling
        att = self.att_drop(att)
        out = torch.einsum('bhal, bhlv -> bhav ', att, values) # sum over
the third axis
        out = rearrange(out, "b h n d -> b n (h d)")
        out = self.projection(out)

        return out
```

Реализуем shortcut connection:

```
#shortcut connection
class ResidualAdd(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

    def forward(self, x, **kwargs):
        res = x
        x = self.fn(x, **kwargs)
        x += res
        return x
```

Из блока внимания выходные данные передаются на полносвязный слой. **Реализуем полносвязный слой:**

```
#полносвязный слой
class FeedForwardBlock(nn.Sequential):
    def __init__(self, emb_size: int, L: int = 4, drop_p: float = 0.):
        super().__init__(
            nn.Linear(emb_size, L * emb_size),
            nn.GELU(),
            nn.Dropout(drop_p),
            nn.Linear(L * emb_size, emb_size),
        )
```

Реализуем кодировщик (Encoder) трансформера:

```
#в целом весь transformer encoder блок
class TransformerEncoderBlock(nn.Sequential):
    def __init__(self, emb_size: int = 768, drop_p: float = 0.,
forward_expansion: int = 4,
                 forward_drop_p: float = 0.,
                 **kwargs):

        super().__init__(
            ResidualAdd(nn.Sequential(
                nn.LayerNorm(emb_size),
                MultiHeadAttention(emb_size, **kwargs),
                nn.Dropout(drop_p)
            )),
            ResidualAdd(nn.Sequential(
                nn.LayerNorm(emb_size),
                FeedForwardBlock(
                    emb_size, L=forward_expansion, drop_p=forward_drop_p),
                nn.Dropout(drop_p)
            )
            ))
```

Далее, используем несколько Encoder блоков:

```
#несколько encoder блоков
class TransformerEncoder(nn.Sequential):
    def __init__(self, depth: int = 12, **kwargs):
        super().__init__(*[TransformerEncoderBlock(**kwargs) for _ in
range(depth)])
```

Добавим классификатор. Он является стандартным полносвязным блоком, который возвращает вероятность класса. На вход принимает ранее вычисленное среднее по всей последовательности. **Реализуем классификатор:**

```python
#классификатор
class ClassificationHead(nn.Sequential):
    def __init__(self, emb_size: int = 768, n_classes: int = 1000):
        super().__init__(
            Reduce('b n e -> b e', reduction='mean'),
            nn.LayerNorm(emb_size),
            nn.Linear(emb_size, n_classes))
```

2.3 Реализация архитектуры

Теперь из всех этих блоков составляем Vision Transformer. **Реализуем архитектуру Vision Transformer:**

```python
#vision transformer
class ViT(nn.Sequential):
    def __init__(self,
                 in_channels: int = 3,
                 patch_size: int = 16,
                 emb_size: int = 768,
                 img_size: int = 224,
                 depth: int = 12,
                 n_classes: int = 1000,
                 **kwargs):
        super().__init__(
            PatchEmbedding(in_channels, patch_size, emb_size, img_size),
            TransformerEncoder(depth, emb_size=emb_size, **kwargs),
            ClassificationHead(emb_size, n_classes)
        )
```

Проверим корректность реализации:

```python
print(summary(ViT(), (3,224,224), device='cpu'))
```

```
==========================================
Total params: 86,415,592
Trainable params: 86,415,592
Non-trainable params: 0
------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 364.33
Params size (MB): 329.65
Estimated Total Size (MB): 694.56
------------------------------------------
```

3 ОБУЧЕНИЕ VISION TRANSFORMER

3.1 Исходные данные

В качестве исходных данных используем CIFAR-10.

Набор данных CIFAR-10 состоит из 60000 цветных изображений размером 32x32 пикселя, каждое изображение относится к одному из 10 классов, по 6000 изображений в каждом классе. Есть 50000 обучающих изображений и 10000 тестовых изображений.

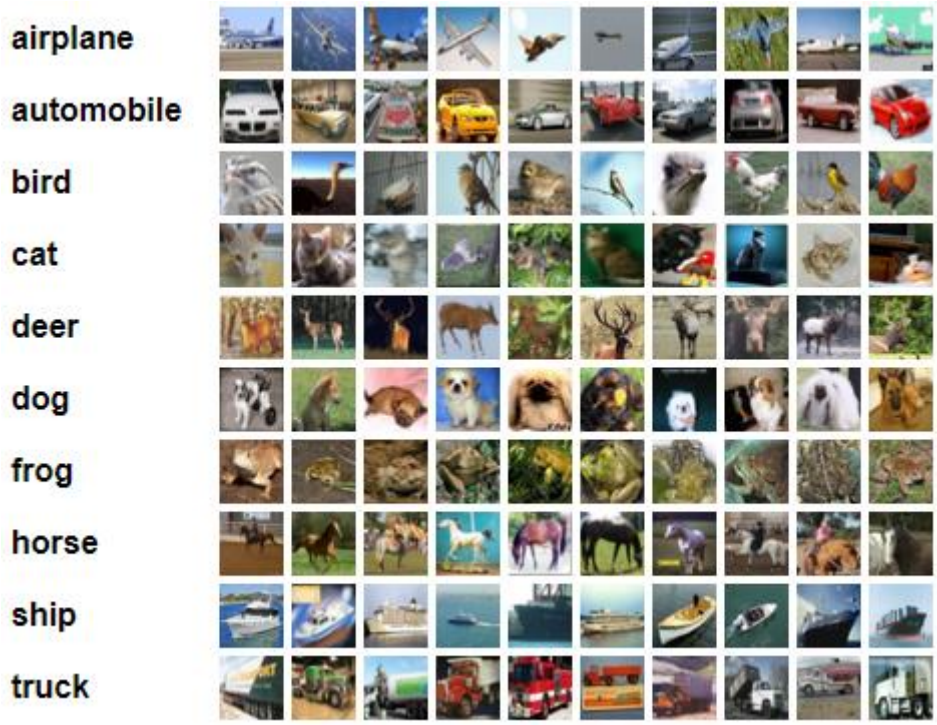

Рисунок 5 – Набор данных CIFAR-10

3.2 Подготовка к обучению

Для того, чтобы обучение модели проходило быстрее, рекомендуется использовать GPU. В Google Collab можно сменить среду выполнения через контекстное меню: «Среда выполнения» → «Сменить среду выполнения» → «GPU» → «T4»

Проверим среду выполнения:

```
#проверка,что на GPU
import tensorflow as tf
tf.test.gpu_device_name()
```

```
'/device:GPU:0'
```

Теперь, импортируем библиотеки:

```
#импорт библиотек
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torch.utils.data.sampler import SubsetRandomSampler
from torch.cuda.amp import autocast, GradScaler
```

Зададим параметры обучения:

```
# Задаем параметры обучения
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
num_epochs = 20
batch_size = 32
learning_rate = 0.001
```

Загрузим набор данных CIFAR-10. Данные необходимо нормализировать:

```
# Загружаем CIFAR-10 датасет, преобразуем размер и нормализируем
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False,
download=True, transform=transform)
```

Для обучения необходимо **реализовать загрузчики данных**. Так, для каждой эпохи обучения будет выбираться пакет размером batch_size случайных изображений из обучающей выборки и передаваться в качестве входных данных.

```
# Создаем загрузчики данных
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True, num_workers=2)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False, num_workers=2)
```

Теперь, инициализируем саму модель Vision Transformer. Инициализируем метод обучения, функцию потерь и функцию, которая позволяет автоматически масштабировать градиенты:

```
# Инициализируем модель и оптимизатор
#torch.cuda.empty_cache()
model =
ViT(in_channels=3,patch_size=4,emb_size=64,img_size=32,depth=12,n_classes=
10).to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()
# Инициализируем GradScaler для автоматического масштабирования градиентов
scaler = GradScaler()
```

3.3 Обучение модели

Обучим модель на 20 эпохах, при размере пакета 32 и начальной скорости обучения 0.001. **Код обучения:**

```
# Обучение модели
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()

        with autocast():
            outputs = model(images)
            loss = criterion(outputs, labels)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        running_loss += loss.item()

    # Вывод промежуточных результатов
    print(f"Epoch [{epoch + 1}/{num_epochs}], Loss: {running_loss /
len(train_loader):.4f}")
```

Промежуточные результаты:

```
Epoch [1/20], Loss: 1.7284
Epoch [2/20], Loss: 1.3934
Epoch [3/20], Loss: 1.2522
Epoch [4/20], Loss: 1.1583
Epoch [5/20], Loss: 1.0960
Epoch [6/20], Loss: 1.0467
Epoch [7/20], Loss: 1.0011
Epoch [8/20], Loss: 0.9644
Epoch [9/20], Loss: 0.9269
Epoch [10/20], Loss: 0.8983
Epoch [11/20], Loss: 0.8712
Epoch [12/20], Loss: 0.8468
Epoch [13/20], Loss: 0.8174
Epoch [14/20], Loss: 0.7951
Epoch [15/20], Loss: 0.7761
Epoch [16/20], Loss: 0.7555
Epoch [17/20], Loss: 0.7361
Epoch [18/20], Loss: 0.7121
Epoch [19/20], Loss: 0.7024
Epoch [20/20], Loss: 0.6891
```

4 ТЕСТИРОВАНИЕ VISION TRANSFORMER

Протестируем обученную модель на данных из тестирующей выборки.

Код тестирования:

```
# Оценка модели
model.eval()
total_correct = 0
total_samples = 0

with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        _, predicted = torch.max(outputs, 1)

        total_samples += labels.size(0)
        total_correct += (predicted == labels).sum().item()

accuracy = total_correct / total_samples * 100
print(f"Accuracy on test set: {accuracy:.2f}%")
```

Результат тестирования:

```
Accuracy on test set: 67.09%
```

5 ЗАДАНИЕ

1. Реализовать структуру Vision Transformer.
2. Выполнить обучение модели.
3. Выполнить тестирование модели.
4. Изменить каждый из параметров обучения (количество эпох, размер пакета, начальная скорость обучения), повторить пункты 2 и 3.
5. Сравнить полученные результаты.