Нижегородский государственный университет им. Н.И. Лобачевского
Институт информационных технологий, математики и механики
Кафедра Математического обеспечения и суперкомпьютерных технологий

Образовательный курс «Введение в глубокое обучение с использованием Intel® neon™ Framework»

Практическая работа №4 Начальная настройка весов наиболее перспективных архитектур полностью связанных сетей с использованием средств Intel® neon™ Framework

При поддержке компании Intel

Жильцов М.С.

Нижний Новгород
2018

Содержание

1 Введение

Данная практическая работа направлена на изучение подходов к начальной настройке параметров глубоких моделей с использованием средств Intel® neon™ Framework. Процедура начальной настройки выполняется посредством построения стека автокодировщиков. Применение данного подхода демонстрируется на примере решения задачи классификации пола человека по фотографии. В качестве набора данных используется IMDB-WIKI [4]. Приведенная последовательность разработки сетей может быть использована для решения других задач классификации при условии, что предварительно разработаны функции для загрузки и чтения данных в формате, принимаемом инструментом Intel® neon™ Framework.

2 Методические указания

2.1 Цели и задачи работы

Цель настоящей работы состоит в том, чтобы изучить основные подходы к начальной настройке параметров глубоких моделей и выполнить начальную настройку некоторых разработанных ранее полносвязных сетей посредством построения стека автокодировщиков с использованием Intel® neon™ Framework.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Изучить подходы к начальной настройке параметров глубоких нейросетевых моделей.
2. Изучить общую схему построения стека автокодировщиков.
3. Разработать основной скрипт для обучения и тестирования глубоких нейросетевых моделей, обеспечивающих решение задачи классификации пола человека по фотографии.
4. Разработать модели нейронных сетей, необходимых для обучения стека автокодировщиков, и описать их с использованием средств Intel® neon™ Framework.
5. Выполнить обучение стека автокодировщиков и результирующей модели, оценить качество решения задачи классификации пола человека по фотографии.

2.2 Структура работы

В работе демонстрируется общая схема разработки и применения стека автокодировщиков в Intel® neon™ Framework. Вначале приводится пример разработки нейронных сетей, необходимых для обучения стека автокодировщиков. Описывается модель полносвязной сети для решения задачи классификации пола человека по фотографии. Далее выполняется загрузка тестовых и тренировочных данных. Разрабатывается скрипт, обеспечивающий обучение и тестирование автокодировщиков и результирующей сети, а также сохранение результатов решения задачи. В качестве набора данных используется IMDB-WIKI [4].

2.3 Рекомендации по проведению занятий

При выполнении данной лабораторной работы рекомендуется следующая последовательность действий:

1. Изучить подходы к начальной настройке параметров глубоких нейросетевых моделей. Для этого можно использовать лекционные материалы и рекомендованную по курсу литературу.
2. Изучить общую схему построения стека автокодировщиков.
3. Разработать модели нейронных сетей, необходимых для обучения стека автокодировщиков и результирующей модели.
4. Загрузить тренировочные и тестовые данные.
5. Разработать скрипт для обучения и тестирования построенных моделей.
6. Добавить сохранение результатов решения задачи в разработанный на предыдущем шаге скрипт.

3 Инструкция по выполнению работы

3.1 Разработка модели сети

Использование стека автокодировщиков для построения сети подразумевает выполнение следующих шагов:

- Разработка архитектуры целевой модели, для предварительной настройки которой применяется стек автокодировщиков.
- Выделение этапов обучения целевой модели и формирование стека автокодировщиков. Для каждого этапа предполагается использование отдельной модели.
- Обучение моделей стека автокодировщиков.
- Формирование результирующей модели, перенос обученных весов из стека автокодировщиков и обучение модели на целевом наборе данных.

В качестве целевой модели используем полносвязную сеть с тремя обучаемыми слоями. Структура сети описывается следующим образом:

```
layers = [
    DataTransform(transform=Normalizer(divisor=128.0)),

    Affine(nout=128, init=Xavier(), bias=Constant(0), activation=Tanh(),
        name='fc_1'),
    Affine(nout=64, init=Xavier(), bias=Constant(0), activation=Tanh(),
        name='fc_2'),
    Affine(nout=2, init=Xavier(), bias=Constant(0),
        activation=Logistic(shortcut=True), name='cls')
]
model = Model(layers)
cost = GeneralizedCost(costfunc=CrossEntropyBinary())
```

Отметим, что для слоев сети определены идентификаторы посредством указания параметра **`name`**. В качестве целевой функции используется бинарная кросс-энтропия.

Выделим этапы обучения сети, на основе которых формируется стек автокодировщиков. Сеть имеет три параметрических слоя. Настроим параметры первых двух слоев, для этого построим отдельный автокодировщик для каждого из них. Необходимо подготовить описание сети для каждого из автокодировщиков. Формирование итоговой модели на основе автокодировщиков выполняется с использованием переноса обучения.

Автокодировщики состоят из двух основных компонентов, один (кодировщик) из которых формирует сжатое представление входных данных, другой (декодировщик) – восстанавливает исходное представление. Входные и выходные данные сети совпадают. Учтем эти особенности при построении дополнительных сетей.

Сеть, которой описывается первый автокодировщик, предназначена для настройки параметров первого слоя целевой сети. Входными и выходными данными на этом этапе являются изображения. Для определенности, примем размер изображения равным 128x128 пикселей. Изображение имеет 3 цветовых канала. Структура сети для первого автокодировщика описывается функцией следующего вида:

```
import numpy as np
from neon.models import Model
from neon.transforms import SumSquared, Normalizer, Tanh
from neon.layers import Affine, DataTransform, GeneralizedCost
from neon.initialiers import Gaussian, Constant

def generate_mlp_ae_stacked_step1_model(input_shape):
    output_size = int(np.prod(input_shape))

    layers = [
```

```
        DataTransform(transform=Normalizer(divisor=128.0)),

        Affine(nout=128, init=Gaussian(scale=0.1), bias=Constant(0),
            activation=Tanh(), name='fc_1'),
        Affine(nout=output_size, init=Gaussian(scale=0.1), bias=Constant(0),
            activation=Tanh(), name='fc_-1')
    ]
    model = Model(layers)
    cost = GeneralizedCost(costfunc=SumSquared())
    return (model, cost)
```

Функция принимает в качестве параметра размер входного изображения. Параметр необходим для указания количества выходов сети. Сеть содержит два слоя: кодировщик и декодировщик. Кодировщик отвечает первому слою целевой сети. Декодировщик служит для восстановления исходной размерности данных из сжатого представления, полученного в результате кодирования. Можно видеть, что идентификатор первого слоя совпадает с идентификатором первого слоя в описании целевой сети. Это используется далее во время переноса обучения. В качестве целевой функции при обучении указана сумма квадратов отклонений, так как решается задача линейной регрессии для каждого пикселя входного изображения.

Создание модели, соответствующей первому автокодировщику, выполняется посредством следующего вызова:

```
input_shape = (3, 128, 128)
model, cost = generate_mlp_ae_stacked_step1_model(input_shape)
```

Определим сеть, отвечающую второму автокодировщику. На данном этапе выполняется настройка параметров второго слоя целевой сети. Входными и выходными данными являются выходные данные первого кодировщика. Для получения целевой сети, необходимо выполнить перенос обучения для обученных на разных этапах слоев. Это можно сделать несколькими способами.

- На каждом этапе обучается новая модель автокодировщика, не зависящая от модели предыдущего автокодировщика. Обучение выполняется на выходных данных кодирующей части предыдущего автокодировщика. Модели, обученные на каждом из этапов, сохраняются. Для формирования целевой модели выполняется перенос параметров слоев из всех моделей автокодировщиков в стеке.
- На каждом этапе обучается новая модель автокодировщика, дополняющая модель предыдущего автокодировщика. Для обучения новой модели выполняется перенос параметров кодирующих слоев предыдущих автокодировщиков и фиксируются параметры слоев, обученных на предыдущих этапах. Входными данными на каждом этапе являются исходные изображения, выходными данными являются выходы кодирующей части сети, полученной от предыдущего автокодировщика. Целевая модель с предобученными параметрами формируется посредством переноса весов из модели последнего автокодировщика.

В данной работе рассматривается второй вариант реализации. Таким образом, модель второго автокодировщика включает в себя кодирующую часть первого автокодировщика (первый слой модели) и второй слой целевой модели. Декодирующая часть состоит из одного слоя, количество выходов которого совпадает с количеством выходов первого слоя целевой сети. Функция формирования модели второго автокодировщика приведена ниже.

```
def generate_mlp_ae_stacked_step2_model(input_shape):
    layers = [
        DataTransform(transform=Normalizer(divisor=128.0)),

        Affine(nout=128, init=Xavier(), bias=Constant(0),
            activation=Tanh(), name='fc_1'),
        Affine(nout=64, init=Xavier(), bias=Constant(0),
            activation=Tanh(), name='fc_2'),
        Affine(nout=128, init=Xavier(), bias=Constant(0),
```

```
            activation=Tanh(), name='fc_-2')
    ]
    model = Model(layers=layers)
    cost = GeneralizedCost(costfunc=SumSquared())
    return (model, cost)
```

Структура модели аналогична структуре модели автокодировщика с предыдущего этапа. Параметр функции **`input_shape`** введен для единообразия интерфейсов функций, создающих модели стека, и здесь не используется.

Для каждого из автокодировщиков необходимо создать дополнительную модель, состоящую только из кодирующей части. Это требуется для формирования выходных данных, используемых для обучения следующей сети в стеке автокодировщиков. Функция формирования такой сети для первого автокодировщика имеет вид:

```
def generate_mlp_ae_stacked_step1_encoder_model():
    layers = [
        DataTransform(transform=Normalizer(divisor=128.0)),

        Affine(nout=128, init=Gaussian(scale=0.1), bias=Constant(0),
            activation=Tanh(), name='fc_1'),
    ]
    model = Model(layers)
    cost = GeneralizedCost(costfunc=SumSquared())
    return (model, cost)
```

Функция формирования такой сети для второго автокодировщика аналогична соответствующей функции, описывающей структуру целевой сети без последнего слоя классификации.

```
def generate_mlp_ae_stacked_step2_encoder_model():
    layers = [
        DataTransform(transform=Normalizer(divisor=128.0)),

        Affine(nout=128, init=Xavier(), bias=Constant(0),
            activation=Tanh(), name='fc_1'),
        Affine(nout=64, init=Xavier(), bias=Constant(0),
            activation=Tanh(), name='fc_2'),
    ]
    model = Model(layers)
    cost = GeneralizedCost(costfunc=SumSquared())
    return (model, cost)
```

Все приведенные функции создания моделей размещаются в файле **`ae_models.py`**.

3.2 Разработка скрипта для обучения и тестирования модели

3.2.1 Общая структура скрипта

Скрипт для обучения и тестирования моделей состоит из нескольких логических частей:

1. Инициализация. Предусматривает, в частности, указание устройства, на котором выполняется обучение и тестирование модели.
2. Подготовка и загрузка данных. Предполагает вызов функций, разработанных в начальной практический работе.
3. Создание модели. Предусматривает вызов функций, разработанных ранее в настоящей практической работе.
4. Обучение модели. Предполагает выбор метода оптимизации, настройку его параметров и вызов метода обратного распространения ошибки.
5. Повтор шагов 3 и 4 для всех моделей в стеке автокодировщиков и целевой модели.
6. Тестирование целевой модели. Подразумевает прямой проход нейросетевой модели для тестового набора данных и определение качества работы построенной модели.

7. Сохранение вывода целевой модели. Предусматривает сохранение значений выхода сети для тестового набора данных.

Для обучения и тестирования целевой модели используется скрипт, разработанный в предыдущей практической работе. Здесь рассмотрим шаги, необходимые для обучения остальных моделей стека.

3.2.2 Инициализация

Перед использованием neon необходимо выполнить инициализацию библиотеки. На этом шаге определяется устройство, на котором будут выполняться вычисления, количество изображений в пачке во время обучения, используемый тип данных и другие параметры. Инициализация выполняется посредством вызова функции **gen_backend** [5]:

```
from neon.backends import gen_backend

be = gen_backend('gpu', batch_size=10)
```

3.2.3 Подготовка и загрузка данных

Действия по подготовке данных детально описаны в начальной практической работе. После выполнения этих действий должны быть сформированы файлы с данными обучающей и тестовой выборок набора данных IMDB-WIKI. Будем предполагать, что файлы размещены в директории **data_wiki** и называются **train.h5** и **test.h5**. Для загрузки данных в формате HDF5 используется тип **HDF5Iterator** [6]. Он позволяет загружать данные из внешней памяти пачками, размер которых указывается при инициализации neon. При обучении автокодировщиков используются итераторы типа **HDF5IteratorAutoencoder** [7]. Они отличаются тем, что возвращают в качестве выходных значений входные. Для загрузки данных потребуются следующие действия:

```
from neon.data import HDF5IteratorAutoencoder

train_set = HDF5IteratorAutoencoder('data_wiki/train.h5')
test_set = HDF5IteratorAutoencoder('data_wiki/test.h5')
```

В результате будут созданы объекты, обеспечивающие доступ к элементам набора данных.

3.2.4 Создание модели

Используя ранее созданный файл с описаниями моделей, определим список функций генерации моделей для каждого из автокодировщиков. Дополнительно определим список функций, создающих кодирующие модели на каждом этапе.

```
import models

step_model_generators = [
    models.generate_mlp_ae_stacked_step1_model, # модель первого этапа
    models.generate_mlp_ae_stacked_step2_model  # модель второго этапа
]
step_encoder_generators = [
    models.generate_mlp_ae_stacked_step1_encoder_model,
    models.generate_mlp_ae_stacked_step2_encoder_model
]
```

Создание модели выполняется посредством вызова соответствующей функции из списка.

```
input_shape = (3, 128, 128)
step = 0
step_model, cost = step_model_generators[step](input_shape)
step_encoder, _ = step_encoder_generators[step]()
```

3.2.5 Обучение модели

Рассмотрим обучение первой модели стека. Оптимизацию параметров модели выполним с помощью алгоритма стохастического градиентного спуска (Stochastic Gradient Descend, SGD).

```
from neon.optimizers import GradientDescentMomentum

optimizer = GradientDescentMomentum(0.01, momentum_coef=0.9, wdecay=0.0005)
```

Для обучения сети необходимо выполнить следующие действия:

```
from neon.callbacks.callbacks import Callbacks

callbacks = Callbacks(model)
step_model.fit(train_set, optimizer=optimizer,
    num_epochs=10, cost=cost, callbacks=callbacks)
```

Далее необходимо сформировать данные для следующей модели в стеке и выполнить ее обучение. Для формирования данных создадим кодирующую сеть и выполним перенос весов из обученной на текущем этапе модели. Перенос обучения выполняется с помощью функции **`import_matching_layers`**, реализованной в предыдущей практической работе.

```
import_matching_layers(step_model.get_description(get_weights=True),
    step_encoder)
```

Функция создания и сохранения набора данных для следующего шага обучения может быть реализована так, как показано ниже.

```
def generate_next_dataset(encoder, train_set, val_set, save_path, step):
    lshape = next(iter(train_set))[0].get()[:, 0].shape

    save_path_base = save_path[:save_path.rfind(".")] + \
        "_outputs_step" + str(step)

    def make_subset(subset_name, subset):
        outputs = encoder.get_outputs(subset)
        subset_path = save_path_base + "_" + subset_name + ".h5"
        generate_dataset(subset_name, subset, outputs, subset_path)
        return HDF5Iterator(subset_path), outputs[0].shape
    next_train_set, lshape = make_subset('train', train_set)
    next_val_set, _ = make_subset('val', val_set)

    return next_train_set, next_val_set, lshape
```

Функция создания набора в формате HDF5:

```
import h5py as h5
from contextlib import closing

def generate_dataset(subset_name, subset, outputs, save_path):
    inputs = iter(subset)
    sample_input = next(iter(subset))[0].get().transpose()[0]
    lshape = sample_input.shape
    input_size = int(np.prod(lshape))
    output_size = int(np.prod(outputs[0].shape))
    with closing(h5.File(save_path, 'w')) as dataset_file:
        dataset_inputs = dataset_file.create_dataset('input',
            (len(outputs), input_size), dtype=np.int8)
        dataset_inputs.attrs['lshape'] = lshape
        dataset_outputs = dataset_file.create_dataset('output',
            (len(outputs), output_size), dtype=np.int8)
        i = 0
        for batch in inputs:
            for inp in batch[0].get().transpose():
                if (len(outputs) <= i):
                    break
                output = outputs[i]
                dataset_inputs[i] = inp.reshape((-1)).astype(np.int8)
```

```
                dataset_outputs[i] = output.reshape((-1)).astype(np.int8)
                i += 1
```

В этих функциях выполняется получение выходов кодирующей части сети с текущего этапа и их сохранение в файлах формата HDF5. Описание формата хранения данных в файле и способов получения данных представлено в первой практической работе. Функции возвращают пару итераторов на сформированные наборы данных (тренировочный и тестовый) и форму входных данных нового набора. Создание нового набора данных выполняется следующим образом:

```
save_path = '.'
train_set, val_set, input_shape = generate_next_dataset(
    step_encoder, train_set, val_set, save_path, step)
```

После создания данных, можно выполнить обучение следующей сети в стеке. Для этого потребуется создать ее и перенести обученные веса из предыдущей модели. Для перенесенных слоев параметр скорости обучения в текущей реализации устанавливается равным нулю. Подробное описание этих действий приведено в предыдущей практической работе. Код, необходимый для обучения новой модели, показан ниже.

```
# Создание новой модели
step = 1
prev_model = step_model
step_model, cost = step_model_generators[step](input_shape)
imported_layers = import_matching_layers(
    prev_model.get_description(get_weights=True), step_model)

# Определение алгоритма оптимизации
default_optimizer = GradientDescentMomentum(
    0.1, momentum_coef=0.9, wdecay=0.0005)
optimizers_mapping = {'default': default_optimizer}

# Запрет обучения перенесенных слоев
imported_layers_optimizer_params = \
    default_optimizer.get_description().copy()['config']
imported_layers_optimizer_params['learning_rate'] = 0
imported_layers_optimizer = \
    GradientDescentMomentum(**imported_layers_optimizer_params)
optimizers_mapping.update(
    {l: imported_layers_optimizer for l in imported_layers})
optimizer = MultiOptimizer(optimizers_mapping)

# Обучение
callbacks = Callbacks(model, eval_set=val_set)
step_model.fit(train_set, optimizer=optimizer,
    num_epochs=10, cost=cost, callbacks=callbacks)
```

Указанная последовательность действий может использоваться для обучения всех последующих моделей в стеке, если они имеются. Для этого потребуется расширить списки функций, генерирующих модели.

Шаги по обучению моделей автокодировщика можно представить циклом следующего вида:

```
prev_model = None
for step, step_model_name in enumerate(model_steps):
    step_model, cost = step_model_generators[step](input_shape)

    optimizer = None
    if prev_model is not None:
        imported_layers = import_matching_layers(
            prev_model.get_description(get_weights=True), step_model)

        default_optimizer = GradientDescentMomentum(
```

```
            0.1, momentum_coef=0.9, wdecay=0.0005)
        optimizers_mapping = {'default': default_optimizer}
        imported_layers_optimizer_params = \
            default_optimizer.get_description().copy()['config']
        imported_layers_optimizer_params['learning_rate'] = 0
        imported_layers_optimizer = \
            GradientDescentMomentum(**imported_layers_optimizer_params)
        optimizers_mapping.update(
            {l: imported_layers_optimizer for l in imported_layers})
        optimizer = MultiOptimizer(optimizers_mapping)
    else:
        optimizer = GradientDescentMomentum(
            0.1, momentum_coef=0.9, wdecay=0.0005)

    callbacks = Callbacks(model, eval_set=val_set)
    step_model.fit(train_set, optimizer=optimizer,
        num_epochs=10, cost=cost, callbacks=callbacks)

    step_encoder, _ = step_encoder_generators[step]()
    train_set, val_set, input_shape = generate_next_dataset(
        step_encoder, train_set, val_set, save_path, step)

    prev_model = step_model
```

3.2.6 Сохранение моделей

Для сохранения модели в файле с указанным именем используется функция модели **`save_params`** [8]. Код сохранения модели на этапе обучения автокодировщика:

```
save_path = './model'
step_model.save_params(save_path + ".step_" + str(step) + ".prm")
```

3.3 Запуск обучения и тестирования модели и проведение экспериментов

3.3.1 Запуск скрипта для обучения и тестирования модели без указания параметров

К настоящем моменту предполагается, что разработаны скрипты для выполнения обучения с переносом (файл **`main_transfer.py`**) и обучения моделей стека автокодировщиков (файл **`main_train_stack.py`**). Запуск скрипта обучения стека осуществляется из командной строки посредством вызова команд, приведенных ниже. Вначале инициализируется виртуальное окружение neon, далее выполняется запуск скрипта.

```
. .venv/bin/activate
python main_train_stack.py
```

3.3.2 Параметризация запуска

Процесс запуска имеет множество параметров, относящихся к работе с neon, источнику данных, нейронной сети, параметрам обучения и тестирования. Все эти параметры можно зафиксировать в скрипте, изменяя их по необходимости. Другим способом является введение параметров запуска скрипта.

neon предоставляет средства обработки и использования параметров командной строки. Для этого предназначен тип **`NeonArgparser`** [9]. Для его использования потребуется следующий код:

```
from neon.util.argparser import NeonArgparser

parser = NeonArgparser()
args = parser.parse_args()
```

После выполнения этого кода переменная **args** содержит набор параметров командной строки. Инициализация neon выполняется автоматически с учетом переданных параметров. Задание пользовательских параметров можно выполнить следующим образом:

```
parser.add_argument('--data_root', default='./data_wiki')
```

После разбора параметров значение нового параметра извлекается следующим образом:

```
data_root = args.data_root
train_set = HDF5IteratorAutoencoder(data_root + '/train.h5')
test_set = HDF5IteratorAutoencoder(data_root + '/test.h5')
```

3.3.3 Запуск скрипта с указанием входных параметров

Для запуска скрипта потребуется передача некоторых параметров. Запуск можно выполнить с использованием команды вида:

```
python main_train_stack.py -b gpu -e 10 -z 32 --data_root data_wiki \
    --serizalize 5 --save_path model.prm
```

Рассмотрим более подробно перечень параметров.

- **b <cpu, mkl, gpu>** обеспечивает выбор устройства для запуска обучения и тестирования.
- **e <number>** указывает количество эпох обучения.
- **z <number>** устанавливает размер пачки.
- **data_root <dir>** указывает директорию, содержащую данные.
- **serialize <number>**, **save_path <name.prm>** обеспечивают сохранение модели во время обучения каждые **<number>** эпох в файл с именем **<name.prm>**.

Полный перечень доступных параметров запуска можно получить, передав опцию **--help**.

Для повышения удобства работы рекомендуется создать shell-скрипт, содержащий командную строку запуска. В таком скрипте дополнительно можно выполнить инициализацию переменных окружения и активацию виртуального окружения.

3.3.4 Обучение и тестирование целевой модели

На данный момент предполагается, что выполнено обучение стека автокодировщиков. Последним шагом является обучение целевой модели на данных задачи. Для определенности, будем считать, что результаты обучения стека, т.е. модель, обученная на последнем шаге, сохранены в файле **mlp_stack_final.prm**. Необходимо перенести обученные слои в целевую модель и обучить ее. Для этого используем скрипт, реализующий перенос обучения, разработанный в предыдущей работе. В нем необходимо указать новую целевую модель. Запуск выполняется следующим образом:

```
python main_transfer.py -b gpu -e 10 -z 32 --data_root data_wiki \
    --serizalize 5 --save_path model.prm --source_model mlp_stack_final.prm
```

Полные исходные коды примера можно найти в материалах данного курса:

Practice/Practice4_ae/main_train_stacked_autoencoder.py и **Practice/models/ae_models.py**.

4 Литература

4.1 Основная литература

1. Хайкин С. Нейронные сети. Полный курс. – М.: Издательский дом «Вильямс». – 2006. – 1104 с.
2. Осовский С. Нейронные сети для обработки информации. – М.: Финансы и статистика. – 2002. – 344 с.
3. Goodfellow I., Bengio Y., Courville A. Deep Learning. – MIT Press. – 2016. – [http://www.deeplearningbook.org].

4.2 Ресурсы сети Интернет

4. Домашняя страница набора данных IMDB-WIKI [https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki].
5. Документация Intel® neon™ Framework: инициализация [http://neon.nervanasys.com/docs/latest/generated/neon.backends.gen_backend.html#neon.backends.gen_backend].
6. Документация Intel® neon™ Framework: тип HDF5Iterator [http://neon.nervanasys.com/docs/latest/generated/neon.data.hdf5iterator.HDF5Iterator.html#neon.data.hdf5iterator.HDF5Iterator].
7. Документация Intel® neon™ Framework: тип HDF5IteratorAutoencoder [http://neon.nervanasys.com/docs/latest/generated/neon.data.hdf5iterator.HDF5IteratorAutoencoder.html#neon.data.hdf5iterator.HDF5IteratorAutoencoder].
8. Документация Intel® neon™ Framework: save_params [http://neon.nervanasys.com/docs/latest/generated/neon.models.model.Model.html#neon.models.model.Model.save_params].
9. Документация Intel® neon™ Framework: тип NeonArgparser [http://neon.nervanasys.com/docs/latest/generated/neon.util.argparser.NeonArgparser.html#neon.util.argparser.NeonArgparser].