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 №3 Применение переноса обучения для решения практической задачи 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 техники переноса обучения нейронных сетей с использованием средств Intel® neon™ Framework [4]. Применение переноса обучения демонстрируется на примере решения задачи классификации пола человека по фотографии. Исходная задача – классификация изображений с большим числом категорий (1000 категорий), целевая задача – классификация пола человека по фотографии (2 категории). Модель решения исходной задачи, которая применяется к целевой задаче, при реализации переноса обучения в данной практической работе – сверточная нейронная сеть VGG-16 [5], обученная на данных конкурса ImageNet [6]. В качестве набора данных для решения целевой задачи используется IMDB-WIKI [7]. Приведенная последовательность действий по переносу обучения нейронных 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 общую схему переноса обучения нейронных сетей и применить данный подход для решения задачи классификации пола человека по фотографии с использованием средств инструмента глубокого обучения Intel® neon™ Framework.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Изучить общую схему переноса обучения нейронных сетей.
2. Изучить средства Intel® neon™ Framework, позволяющие реализовать перенос обучения.
3. Разработать основной скрипт для обучения и тестирования глубоких нейросетевых моделей, обеспечивающих решение задачи классификации пола человека по фотографии.
4. Разработать модель сверточной нейронной сети VGG-16 и описать ее с использованием средств Intel® neon™ Framework.
5. Выполнить три типа экспериментов по переносу обучения:
   - использование структуры переносимой модели и ее обучение для решения целевой задачи;
   - использование сверточной части модели, построенной для решения исходной задачи, в качестве метода извлечения признаков;
   - дообучение исходной модели для использования в условиях целевой задачи.
6. Применить обученные модели для решения задачи и оценить качество классификации.

2.2 Структура работы

В работе демонстрируется общая схема реализации переноса обучения в Intel® neon™ Framework. Вначале разрабатывается структура сверточной сети на основе VGG-16. Далее выполняется загрузка тестовых и тренировочных данных. Разрабатывается скрипт, обеспечивающий обучение и тестирование сети в соответствии с возможными типами экспериментов по переносу обучения, также осуществляется сохранение результатов решения задачи. Работа выполняется на примере задачи классификации пола человека по фотографии. В качестве набора данных используется IMDB-WIKI [4].

2.3 Рекомендации по проведению занятий

При выполнении данной практической работы рекомендуется следующая последовательность действий:

1. Изучить возможности работы со сверточными нейронными сетями в neon. Для этого можно использовать лекционные материалы и документацию инструмента neon.
2. Изучить технику переноса обучения в нейронных сетях. Теоретические основы представлены в лекционном материале, а также в предложенных в лекции источниках.

3. Разработать модель сверточной нейронной сети на основе VGG-16.
4. Загрузить тренировочные и тестовые данные.
5. Разработать скрипт для обучения и тестирования построенной модели, который позволяет реализовать три типа экспериментов по переносу обучения.
6. Провести эксперименты по переносу обучения.

3 Инструкция по выполнению работы

3.1 Разработка модели сверточной сети

Создание модели включает в себя описание структуры модели нейронной сети и задание целевой функции, используемой во время обучения. Перенос обучения предполагает наличие заранее обученной модели, из которой необходимо перенести часть обученных слоев в новую модель. Как следствие, новая модель похожа по структуре на исходную модель. Перенос имеет смысл только для параметрических слоев сети. Для выполнения переноса слоя из одной модели в другую необходимым условием является совпадение количества параметров в исходном слое и в целевом. Как правило, соответствие между слоями устанавливается посредством идентификаторов слоев.

Для решения задачи классификации пола человека по фотографии с использованием переноса обучения используем широко известную модель VGG-16 [5], предназначенную для классификации изображений размера 224x224 на 1000 категорий объектов. Описание структуры модели и обученные параметры модели доступны в репозитории neon [8]. Загрузку модели (~500 Мб) можно выполнить с использованием команды, приведенной ниже.

```
wget https://s3-us-west-1.amazonaws.com/nervana-modelzoo/VGG/VGG_D.p
```

После выполнения команды в текущей директории появится файл **`VGG_D.p`**, содержащий обученную модель.

Для решения задачи классификации с двумя категориями создадим описание новой модели, основанной на VGG-16. Для этого потребуется описание исходной модели [9]. Модель, представленная в репозитории, сохранена в устаревшем формате neon. Описание модели, подготовленное к переносу обучения и адаптированное для версии neon 2.6.0, представлено ниже. Фрагменты кода, выделенные полужирным начертанием, соответствуют изменениям, которые внесены в VGG-16 для реализации переноса обучения. По существу, выполнена замена слоев, соответствующих классификатору.

```
init1 = Xavier(local=True)
initfc = GlorotUniform()
relu = Rectlin()
conv_params = {'init': init1, 'strides': 1, 'padding': 1}

layers = []
for i, nofm in enumerate([64, 128, 256, 512, 512]):
    i = i + 1
    layers.append(Conv((3, 3, nofm), **conv_params, name='conv%d_1' % i))
    layers.append(Bias(init=Constant(0), name='conv%d_1_bias' % i))
    layers.append(Activation(Rectlin()))
    layers.append(Conv((3, 3, nofm), **conv_params, name='conv%d_2' % i))
    layers.append(Bias(init=Constant(0), name='conv%d_2_bias' % i))
    layers.append(Activation(Rectlin()))
    if (nofm > 128):
        layers.append(Conv((3, 3, nofm), **conv_params, name='conv%d_3' % i))
        layers.append(Bias(init=Constant(0), name='conv%d_3_bias' % i))
        layers.append(Activation(Rectlin()))
    layers.append(Pooling(2, strides=2))

layers.append(Affine(nout=4096, init=initfc,
    bias=Constant(0), activation=Rectlin(),
    name='fc6_new'))
```

```
layers.append(Dropout(keep=0.5))
layers.append(Affine(nout=4096, init=initfc,
    bias=Constant(0), activation=Rectlin(),
    name='fc7_new'))
layers.append(Dropout(keep=0.5))
layers.append(Affine(nout=2, init=initfc,
    bias=Constant(0), activation=Softmax(),
    name='cls_imdb_wiki_face'))
```

Исходная сеть VGG-16 имеет 1000 выходов, поэтому последний слой сети изменен таким образом, чтобы число выходов стало равно двум. Обратим внимание на параметр слоя **name**. Этот параметр позволяет задать идентификатор слоя. Можно видеть, что в описании сети для всех слоев указано значение этого параметра. При использовании техники переноса обучения идентификатор слоя является удобным и гибким способом обозначения слоев, которые необходимо скопировать из исходной модели в новую. В процессе копирования весов рассматриваются только слои с одинаковыми идентификаторами. Если слой не предполагается копировать, для него следует определить уникальный идентификатор. Исходная сеть обучена для изображений, размер которых не превышает 224x224. Предположим, что входные изображения набора данных имеют другой размер, например, 256x256. Исходная сеть содержит обучаемые (параметрические) слои двух видов: сверточные и полносвязные. Одним из полезных свойств сверточных слоев является инвариантность количества параметров относительно размера входного изображения, поэтому сверточные слои могут быть скопированы без изменений. В случае полносвязных слоев количество связей каждого нейрона совпадает с размером входного изображения или выходом предыдущего слоя. При изменении размера входа меняется и количество параметров. Таким образом, полносвязные слои не могут быть скопированы. Чтобы избежать копирования параметров полносвязных слоев определим для них новые идентификаторы.

Далее необходимо создать список слоев сети и модель.

```
from neon.models import Model

layers = []
# формирование списка слоев сети, представленного выше
# ...
model = Model(layers)
```

В результате приведенных действий формируется новая модель, позволяющая решать задачу классификации с двумя категориями. Разработанная модель может быть обучена с использованием весов из исходной модели.

Определим целевую функцию для оптимизации параметров сети. В данном случае используется функция бинарной кросс-энтропии.

```
from neon.transforms import CrossEntropyBinary

cost = GeneralizedCost(costfunc=CrossEntropyBinary())
```

Для удобства разместим код генерации модели в отдельный файл **models.py**. Создадим функцию генерации модели следующего вида:

```
def generate_vgg_16_transfer_model():
    init1 = Xavier(local=True)
    initfc = GlorotUniform()
    relu = Rectlin()
    conv_params = {'init': init1, 'strides': 1, 'padding': 1}

    layers = []
    for i, nofm in enumerate([64, 128, 256, 512, 512]):
        i = i + 1
        layers.append(Conv((3, 3, nofm), **conv_params, name='conv%d_1' % i))
        layers.append(Bias(init=Constant(0), name='conv%d_1_bias' % i))
```

```
        layers.append(Activation(Rectlin()))
        layers.append(Conv((3, 3, nofm), **conv_params, name='conv%d_2' % i))
        layers.append(Bias(init=Constant(0), name='conv%d_2_bias' % i))
        layers.append(Activation(Rectlin()))
        if nofm > 128:
            layers.append(Conv((3, 3, nofm), **conv_params,
                name='conv%d_3' % i))
            layers.append(Bias(init=Constant(0), name='conv%d_3_bias' % i))
            layers.append(Activation(Rectlin()))
        layers.append(Pooling(2, strides=2))

    layers.append(Affine(nout=4096, init=initfc,
        bias=Constant(0), activation=Rectlin(), name='fc6_new'))
    layers.append(Dropout(keep=0.5))
    layers.append(Affine(nout=4096, init=initfc,
        bias=Constant(0), activation=Rectlin(), name='fc7_new'))
    layers.append(Dropout(keep=0.5))
    layers.append(Affine(nout=2, init=initfc,
        bias=Constant(0), activation=Softmax(), name='cls_imdb_wiki_face'))
    ]
    model = Model(layers)
    cost = GeneralizedCost(costfunc=CrossEntropyBinary())
    return model, cost
```

Загрузку модели из других скриптов можно выполнить с помощью кода:

```
import models

model, cost = models. generate_vgg_16_transfer_model()
```

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 практической работе.
3. Создание целевой модели. Предусматривает вызов функций, разработанных ранее в настоящей практической работе.
4. Загрузка исходной модели и выполнение переноса весов в целевую модель.
5. Обучение целевой модели. Предполагает выбор метода оптимизации, настройку его параметров и вызов метода обратного распространения ошибки.
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.

Рассмотрим более подробно разработку каждой логической части скрипта.

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`** [10]:

```
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`** [11]. Он позволяет загружать данные из внешней памяти пачками, размер которых указывается при инициализации neon. Для загрузки данных потребуются следующие действия:

```
from neon.data import HDF5Iterator

train_set = HDF5Iterator('data_wiki/train.h5')
test_set = HDF5Iterator('data_wiki/test.h5')
```

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

Данные, которые предоставляет объект типа **`HDF5Iterator`**, возвращаются в сохраненном формате. В наборе данных метки классов хранятся в числовом виде. В случае задачи классификации neon требует предоставления меток классов в one-hot-представлении, в котором целевой класс объектов описывается не числом, а вектором длины, равной количеству классов. Указанный вектор содержит нули во всех элементах, кроме одного, совпадающего с индексом целевого класса. Для автоматического преобразования данных из индексного представления в one-hot представление используется тип **`HDF5IteratorOneHot`**.

```
from neon.data import HDF5IteratorOneHot

train_set = HDF5IteratorOneHot('data_wiki/train.h5')
test_set = HDF5IteratorOneHot('data_wiki/test.h5')
```

3.2.4 Создание модели

Используя ранее созданный файл с описанием модели, создадим модель посредством вызова соответствующей функции.

```
import models

model, cost = models.generate_vgg_16_transfer_model()
```

3.2.5 Загрузка исходной модели и копирование обученных весов

neon предоставляет средства по сохранению и загрузке моделей. Для загрузки сохраненной модели VGG-16 из файла используется функция **`load_obj`** [12].

```
from neon.util.persist import load_obj

pretrained_model_filepath = 'VGG_D.p'
pretrained_model = load_obj(pretrained_model_filepath)
```

Далее необходимо выполнить копирование слоев из загруженной модели в целевую. Для этого потребуется получить из целевой модели набор слоев. Это можно сделать посредством вызова метода **`get_description`** [13] у объекта модели или прямым обращением к полю **`layers`**.

```
pdesc = model.get_description(get_weights=True)
# pdesc['model']['config']['layers'] - словарь с описаниями слоев и весами
layers = model.layers.layers
# layers - словарь с описаниями слоев и весами
```

Создадим функцию для копирования весов из одной модели в другую. Функция выполняет следующие действия:

- Формирует ассоциативные массивы слоев исходной модели и целевой. Ключом является название слоя.
- Формирует массив слоев, для которых выполнено копирование весов.

– Для каждого слоя целевой модели проверяет наличие слоя в исходной модели и, если он найден, копирует веса. Для копирования используется метод **load_weights** [14], вызываемый у объекта слоя.
– Возвращает массив слоев, для которых скопированы веса.

```
def import_matching_layers(source_model, target_model):
    source_model_layers = {l['config']['name']: l \
        for l in source_model['model']['config']['layers']}
    print("Source model has layers:", source_model_layers.keys())
    imported_layers = []
    target_model_layers = target_model.layers.layers
    for i, target_layer in enumerate(target_model_layers):
        target_layer_desc = target_layer.get_description()
        source_layer = \
            source_model_layers.get(target_layer_desc['config']['name'])
        if (source_layer is not None):
            target_layer.load_weights(source_layer)
            imported_layers.append(target_layer.name)
            print("Successfully copied layer #%d '%s'" \
                % (i, target_layer.name))
        else:
            print("Unable to copy layer #%d '%s'" \
                % (i, target_layer.name))
    return imported_layers
```

Функция вызывается следующим образом:

```
imported_layers = import_matching_layers(pretrained_model, model)
```

3.2.6 Обучение модели

Для обучения модели требуется выбрать алгоритм оптимизации и задать его параметры. neon предоставляет несколько алгоритмов оптимизации [15]. В глубоком обучении широко используется стохастический градиентный спуск (Stochastic Gradient Descend, SGD [16]). Используем его для оптимизации параметров модели. Как правило, при переносе обучения для скопированных слоев используются меньшие значения параметра скорости обучения (learning rate). neon позволяет задать разные алгоритмы и параметры оптимизации для каждого слоя сети. Настроим алгоритм оптимизации так, чтобы для перенесенных слоев использовались небольшие значения скорости обучения. Для этого используем объект типа **MultiOptimizer** [17], реализующий возможность использования разных параметров алгоритма оптимизации. Конструктор объекта типа **MultiOptimizer** принимает параметр, определяющий алгоритмы оптимизации для слоев сети. Параметр представляет собой ассоциативный массив с ключом, являющимся идентификатором слоя, и значением – алгоритмом оптимизации для слоя. Если для слоя отсутствует запись в массиве, то используется элемент по ключу **default**. Обозначим алгоритм оптимизации, используемый по умолчанию, как **GradientDescentMomentum**. Для скопированных слоев сети создадим аналогичные оптимизаторы с измененным параметром скорости обучения. Список скопированных слоев получен ранее и хранится в переменной **imported_layers**.

```
from neon.optimizers import GradientDescentMomentum, MultiOptimizer

base_lr = 0.001
default_optimizer = GradientDescentMomentum(
    base_lr, 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'] = base_lr * 0.001
imported_layers_optimizer = \
    GradientDescentMomentum(**imported_layers_optimizer_params)
optimizers_mapping.update(
    {l: imported_layers_optimizer for l in imported_layers})

optimizer = MultiOptimizer(optimizers_mapping)
```

Для обучения сети необходимо выполнить следующие действия:

```
from neon.callbacks.callbacks import Callbacks

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

Здесь выполняется обучение на тренировочной выборке набора данных. Длительность обучения равна 10 эпохам. Эпоха обучения эквивалентна одному полному обходу набора данных. Для сохранения свойств алгоритма стохастического градиентного спуска требуется обеспечить случайный выбор данных из набора. Это выполнено на шаге перемешивания набора данных во время предварительной их подготовки. Параметр **`callbacks`** позволяет задать функции, которые будут вызываться в процессе обучения. Так, например, можно организовать обход тестового множества данных по завершении эпохи обучения.

3.2.7 Тестирование модели

Для оценки качества классификации используется следующий код:

```
from neon.transforms import Accuracy

accuracy = model.eval(test_set, metric=Accuracy())
print('Accuracy = %.1f%%' % (accuracy * 100))
```

Здесь указывается метрика качества. Для задачи классификации используется показатель точности (**`Accuracy`**), который отражает долю правильно проклассифицированных примеров. Указанная метрика вычисляется для тестового подмножества данных.

3.2.8 Сохранение вывода модели

Конечной целью обучения сети является получение вывода сети на некотором наборе данных. Для получения вывода можно использовать следующий код:

```
outputs = model.get_outputs(test_set)
```

Сохранение вывода в файл можно организовать следующим образом:

```
import numpy as np

def save_inference(output_file_name, outputs, subset):
    with open(output_file_name, 'w') as output_file:
        output_file.write('inference, target\n')
        outputs_iter = iter(outputs)
        try:
            for dataset_batch in subset:
                targets = np.transpose(dataset_batch[1].get())
                for target in targets:
                    output = next(outputs_iter)
                    target_class = np.argmax(target, axis=0)
                    output_file.write('%s, %s\n' % (output, target_class))
        except StopIteration as e: # the last batch might be incomplete
            pass
        output_file.close()

save_inference('inference.txt', outputs, test_set)
```

Данный фрагмент кода выполняет обход набора данных и выходов сети. Результаты сохраняются в файл в формате:

```
[вероятность класса 1, вероятность класса 2], правильный ответ
```

3.3 Запуск обучения и тестирования модели и проведение экспериментов

3.3.1 Запуск скрипта для обучения и тестирования модели без указания параметров

К настоящему моменту предполагается, что разработан скрипт для обучения и тестирования глубокой модели. Для определенности считаем, что он сохранен в файле с именем **main_transfer.py**. Запуск скрипта осуществляется из командной строки посредством вызова команд, приведенных ниже. Вначале инициализируется виртуальное окружение neon, далее выполняется запуск скрипта.

```
. .venv/bin/activate
python main_transfer.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** [18]. 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 = HDF5IteratorOneHot(data_root + '/train.h5')
test_set = HDF5IteratorOneHot(data_root + '/test.h5')
```

Определим параметр пути до исходной модели:

```
parser.add_argument('--source_model', default=None)
```

Используем новый параметр для загрузки весов по требованию:

```
Imported_layers = []
if (args.source_model is not None):
    pretrained_model = load_obj(args.source_model)
    imported_layers = import_matching_layers(pretrained_model, model)
```

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_transfer.py -b gpu -e 10 -z 32 --data_root data_wiki \
    --serizalize 5 --save_path model.prm --source_model VGG_D.p
```

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>**.
- **source_model <path.prm>** указывает файл с параметрами исходной модели.

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 Выполнение разных видов экспериментов по переносу обучения

Использование структуры переносимой модели и ее обучение для решения целевой задачи. Для реализации данного эксперимента по переносу достаточно обучить разработанную модель со случайно проинициализированными весами. При этом необходимо установить единое значение параметра скорости обучения.

```
imported_layers_optimizer_params['learning_rate'] = base_lr
```

При этом веса обученной модели VGG-16 не импортируются. Для этого достаточно не передавать обученную модель в качестве входного параметра скрипта. Ниже приведена командная строка запуска скрипта для проведения данного эксперимента.

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

Дообучение исходной модели для использования в условиях целевой задачи. Для данного эксперимента необходимо импортировать веса обученной модели VGG-16 в целевую модель и обучить параметры всех слоев. При этом для импортированных слоев имеет смысл установить небольшие значения параметра скорости обучения.

```
imported_layers_optimizer_params['learning_rate'] = base_lr * 0.001
```

В данном случае запуск скрипта для обучения и тестирования модели необходимо выполнить с использованием команды следующего вида:

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

Использование сверточной части модели, построенной для решения исходной задачи, в качестве метода извлечения признаков. Требуется выполнить перенос обучения и установить для перенесенных слоев сниженную скорость обучения. При реализации данного эксперимента необходимо импортировать веса обученной модели VGG-16 в целевую модель, зафиксировать их значения и обучать только параметры полносвязных слоев. В связи с этим параметр скорости обучения у импортированных слоев должен быть выставлен в значение, равное нулю.

```
imported_layers_optimizer_params['learning_rate'] = 0
```

Запуск скрипта осуществляется командой, представленной ниже.

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

Полные исходные коды примера можно найти в материалах данного курса:

Practice3_tl/main_transfer.py и **models/tl_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 Intel® neon™ Framework: [https://github.com/nervanasystems/neon]
5. Simonyan K., Zisserman A. Very deep convolutional networks for large-scale image recognition //arXiv preprint arXiv:1409.1556. – 2014. [https://arxiv.org/pdf/1409.1556].
6. Домашняя страница соревнования ImageNet Large Scale Image Recognition Challenge: [http://www.image-net.org/challenges/LSVRC].
7. Домашняя страница набора данных IMDB-WIKI: [https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki].
8. Примеры Intel® neon™ Framework: VGG-16 [https://github.com/NervanaSystems/ModelZoo/tree/master/ImageClassification/ILSVRC2012/VGG] .
9. Примеры Intel® neon™ Framework: VGG-16, описание модели [https://github.com/NervanaSystems/ModelZoo/blob/master/ImageClassification/ILSVRC2012/VGG/vgg_neon.py].
10. Документация Intel® neon™ Framework: инициализация [http://neon.nervanasys.com/docs/latest/generated/neon.backends.gen_backend.html#neon.backends.gen_backend].
11. Документация Intel® neon™ Framework: тип HDF5Iterator [http://neon.nervanasys.com/docs/latest/generated/neon.data.hdf5iterator.HDF5Iterator.html#neon.data.hdf5iterator.HDF5Iterator].
12. Документация Intel® neon™ Framework: функция load_obj [http://neon.nervanasys.com/docs/latest/generated/neon.util.persist.load_obj.html#neon.util.persist.load_obj].
13. Документация Intel® neon™ Framework: функция get_decription [http://neon.nervanasys.com/docs/latest/generated/neon.models.model.Model.html#neon.models.model.Model.get_description].
14. Документация Intel® neon™ Framework: функция load_weights [http://neon.nervanasys.com/docs/latest/generated/neon.layers.layer.Layer.html#neon.layers.layer.Layer.load_weights].
15. Документация Intel® neon™ Framework: алгоритмы оптимизации [http://neon.nervanasys.com/docs/latest/optimizers.html].
16. Документация Intel® neon™ Framework: тип GradientDescentMomentum [http://neon.nervanasys.com/docs/latest/generated/neon.optimizers.optimizer.GradientDescentMomentum.html#neon.optimizers.optimizer.GradientDescentMomentum].
17. Документация Intel® neon™ Framework: тип MultiOptimizer [http://neon.nervanasys.com/docs/latest/generated/neon.optimizers.optimizer.MultiOptimizer.html#neon.optimizers.optimizer.MultiOptimizer].
18. Документация Intel® neon™ Framework: тип NeonArgparser [http://neon.nervanasys.com/docs/latest/generated/neon.util.argparser.NeonArgparser.html#neon.util.argparser.NeonArgparser].