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 №5 Разработка рекуррентной сети 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 рекуррентных нейронных сетей и разработку архитектур указанных моделей с использованием средств Intel® neon™ Framework для решения практических задач. Применение рекуррентных сетей демонстрируется на примере решения задачи классификации пола человека по фотографии. В качестве набора данных используется IMDB-WIKI [4]. Приведенная последовательность разработки 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 построения рекуррентных нейронных сетей и разработать некоторые архитектуры рекуррентных моделей с использованием средств инструмента глубокого обучения Intel® neon™ Framework для решения задачи классификации пола человека по фотографии.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Изучить общую схему построения рекуррентных нейронных сетей.
2. Изучить средства Intel® neon™ Framework для работы с рекуррентными сетями.
3. Разработать основной скрипт для обучения и тестирования глубоких нейросетевых моделей, обеспечивающих решение задачи классификации пола человека по фотографии.
4. Разработать модель рекуррентной нейронной сети и описать ее с использованием средств Intel® neon™ Framework.
5. Выполнить запуск и оценить качество классификации.

2.2 Структура работы

В работе демонстрируется общая схема разработки и применения рекуррентных нейронных сетей в Intel® neon™ Framework. Вначале приводится пример разработки рекуррентной сети. Далее выполняется загрузка тестовых и тренировочных данных. Разрабатывается скрипт, обеспечивающий обучение и тестирование сети, а также 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 лабораторной работы рекомендуется следующая последовательность действий:

1. Изучить возможности работы с рекуррентными нейронными сетями в neon. Для этого можно использовать лекционные материалы и документацию инструмента neon.
2. Разработать модель рекуррентной нейронной сети.
3. Загрузить тренировочные и тестовые данные.
4. Разработать скрипт для обучения и тестирования построенной модели.
5. Добавить сохранение результатов решения задачи в разработанный на предыдущем шаге скрипт.

3 Инструкция по выполнению работы

3.1 Разработка модели рекуррентной сети

3.1.1 Рекуррентный слой

Создание модели включает в себя описание структуры модели нейронной сети и задание целевой функции, используемой во время обучения. Основным составным элементом модели является

слой. neon предоставляет доступ ко множеству типовых слоев, полный перечень которых можно найти в документации [5].

Рекуррентные слои имеют несколько отличий от сверточных и полносвязных слоев. Они предназначены для обработки последовательностей входных данных и имеют внутреннее состояние, сохраняющееся между обработкой элементов последовательности. Обработка очередного элемента входной последовательности представляет собой формирование выходных значений с учетом текущего состояния и обновление внутреннего состояния. Количество выходов рекуррентных слоев зависит от длины входной последовательности. Форма выходных данных рекуррентного слоя имеет вид **(hidden_outputs, sequence_length * batch_size)**, где **hidden_outputs** – количество выходов слоя, **sequence_length** – длина последовательности, обработанной слоем, **batch_size** – размер пачки входных данных. Для объединения результатов рекуррентного слоя, полученных после обработки последовательности, используются соответствующие средства neon, описанные в документации [6].

3.1.2 Создание рекуррентных слоев в Intel® neon™ Framework

Создание простого рекуррентного слоя можно выполнить следующим образом:

```
from neon.layers import Recurrent
from neon.initializers import Gaussian
from neon.transforms import Tanh

layer = Recurrent(output_size=10, init=Gaussian(0.1), activation=Tanh())
```

В этом фрагменте кода создается простой рекуррентный слой с 10 нейронами. Каждый нейрон связан со всеми выходами предыдущего слоя и имеет одно выходное значение. Выходное значение для текущего элемента последовательности вычисляется по формуле

$o_i = F(Uh_{i-1} + Wx_i + b)$, где

h_{i-1} – состояние на шаге $i-1$, U и W – весовые матрицы, x_i – элемент последовательности i, b – вектор смещений, F – функция активации. Значение состояния слоя вычисляется как $h_i = o_i$. Для инициализации весов и состояний слоя использовано нормальное распределение с нулевым математическим ожиданием и среднеквадратичным отклонением, равным 0.1. Дополнительные смещения (вектор b) для нейронов создаются автоматически и инициализируются нулевыми значениями. В качестве функции активации используется гиперболический тангенс **Tanh**. Полную информацию о доступных параметрах слоя можно найти в документации [7].

3.1.3 Описание рекуррентной сети для решения задачи классификации пола человека по фотографии

Для решения задачи классификации с двумя категориями построим описание рекуррентной модели. В [8, 9] описана идея подхода, который реализуется в настоящей практической работе. Предлагается использовать рекуррентные слои для извлечения высокоуровневой пространственной информации об изображении, в то время как сверточные слои используются для извлечения низкоуровневой пространственной информации. На входе сети имеется одно изображение, на выходе – метка класса. Входные данные не содержат естественных последовательностей таких как, например, при работе с кадрами видео. Последовательности, необходимые для работы рекуррентных сетей, формируются из карт активаций сверточных слоев во время обработки входного изображения и не хранятся в наборе данных. Встроенные средства neon не позволяют реализовать этот подход, поэтому потребуется определить несколько дополнительных слоев. Исходный код функции построения модели представлен ниже.

```
def generate_rnn_model(input_shape=(3, 128, 128)):
    iC = input_shape[0]
    iH = input_shape[1]
    iW = input_shape[2]
    class_count = 2
```

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

        # convolutional encoder / feature extractor
        # resolution 1
        BatchNorm(),
        Conv(fshape=(3, 3, 32), padding=2, strides=1, dilation=2,
            init=Kaiming(), bias=Constant(0), activation=Rectlin()),

        BatchNorm(),
        Pooling(fshape=(3, 3), padding=1, strides=2, op='max'),
        # resolution 1/2
        Conv(fshape=(3, 3, 64), padding=2, strides=1, dilation=2,
            init=Kaiming(), bias=Constant(0), activation=Rectlin()),

        BatchNorm(),
        Pooling(fshape=(3, 3), padding=1, strides=2, op='max'),
        # resolution 1/4
        Conv(fshape=(3, 3, 128), padding=2, strides=1, dilation=2,
            init=Kaiming(), bias=Constant(0), activation=Rectlin()),

        BatchNorm(),
        Pooling(fshape=(3, 3), padding=1, strides=2, op='max'),
        # resolution 1/8
        Conv(fshape=(3, 3, 256), padding=2, strides=1, dilation=2,
            init=Kaiming(), bias=Constant(0), activation=Rectlin()),

        BatchNorm(),
        SpatialRNN(input_shape=(256, iH // 8, iW // 8),
            block_shape=(256, 2, 2),
            RNN=BiRNN,
            RNN_params={'output_size': 256,
                        'init': GlorotUniform(),
                        'activation': Tanh()}
        ), # outputs: (2 * 256, iH // 16, iW // 16)
        # # resolution 1/16

        BatchNorm(),
        SpatialRNN(input_shape=(512, iH // 16, iW // 16),
            block_shape=(512, 2, 2),
            RNN=BiRNN,
            RNN_params={'output_size': 512,
                        'init': GlorotUniform(),
                        'activation': Tanh()}
        ), # outputs: (2 * 512, iH // 32, iW // 32)
        # # resolution 1/32

        # classifier
        BatchNorm(),
        Conv(fshape=(1, 1, class_count), padding=1, strides=1, dilation=1,
            init=Kaiming(), bias=Constant(0), activation=Rectlin()),
        Pooling(fshape='all', padding=0, strides=1, op='avg'),
        Activation(Softmax())
    ]

    model = Model(layers=layers)
    cost = GeneralizedCost(costfunc=CrossEntropyMulti())
    return (model, cost)
```

В этой модели определяется слой непараметрического входного преобразования данных – деления на константу 128. Такое преобразование позволяет имитировать деление на среднеквадратичное отклонение в случае достаточно большой и разнообразной выборки изображений. Далее создаются 4 сверточных слоя с разными параметрами и функцией активации **`ReLU`**. После сверточных слоев добавляются слои пространственного объединения по максимуму – **`Pooling`**. Они выбирают максимальное значение в каждой группе размера 3x3 входных карт активаций и уменьшают размер карт активаций в два раза по каждой стороне (параметр **`stride`**). После сверточных слоев применяются рекуррентные блоки **`SpatialRNN`**, структура которых рассматривается в следующем разделе. В результате работы рекуррентного блока размер изображения уменьшается в 2 раза по каждой стороне. Последним шагом в обработке является применение классификатора. Классификатор является сверточным слоем с количеством фильтров равным количеству классов. Вектор логарифмов вероятностей классов формируется с помощью операции **`Pooling`**, вычисляющей среднее значение каждой из выходных карт активаций сверточного слоя. Целевой функцией для оптимизации параметров сети выбрана бинарная кросс-энтропия. Для улучшения сходимости алгоритмов оптимизации и дополнительной регуляризации используются слои нормализации выходов по пачке данных (**`BatchNorm`**).

Для удобства поместим код генерации модели в отдельный файл **`models.py`**. Загрузку модели из других скриптов можно выполнить с помощью кода:

```
import models

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

Описание всех дополнительных средств, необходимых для построения модели, выполним в отдельном файле **`layers.py`**.

3.1.4 Описание рекуррентного блока, использованного при решении задачи

Рассмотрим рекуррентные блоки, использованные для описания сети. Один рекуррентный блок описывается функцией **`SpatialRNN`**. В функции выполняются следующие действия:

- Входные данные делятся на фрагменты указанного размера.
- Фрагменты объединяются в последовательности.
- Для каждой последовательности создается рекуррентный слой.
- Последовательности распределяются между слоями, обрабатываются, результаты обработки объединяются.
- Объединенные результаты преобразуются к исходному формату.

Указанные действия выполняются дважды. В первый раз последовательности формируются так, что рекуррентные слои выполняют обход изображения по вертикали, во второй раз – по горизонтали. Обходы последовательностей выполняются одновременно в прямом и обратном направлениях с помощью слоя **`BiRNN`** (Bidirectional RNN) [10]. Предполагается, что использование внутреннего состояния рекуррентными слоями позволяет выделить высокоуровневые пространственные признаки на основе низкоуровневых, выделенных ранее сверточными слоями.

Разделение входных данных на фрагменты. Входными данными рекуррентного блока являются карты признаков, полученные от предыдущих слоев. Данные представляются числовым тензором формы **`(C, H, W, N)`**, где **`C`** – число каналов, **`H`** и **`W`** – высота и ширина, **`N`** – размер пачки. Для входных данных выполняется разделение на фрагменты заданного размера. Для каждой размерности тензора используется свой размер фрагмента. Например, для входных карт формы $(32, 64, 64, 1)$ и размера фрагмента $(2, 4, 4, 1)$ получим $32/2 * 64/4 * 64/4 * 1/1$ фрагментов.

После разделения на фрагменты данные объединяются в последовательности. Первым выполняется вертикальный обход изображения, поэтому фрагменты объединяются по вертикали.

Переменные, соответствующие размеру входа **input_shape** и размеру фрагмента **block_shape**, передадим в качестве параметров функции **SpatialRNN**. Определим переменную размера входа после деления на фрагменты **front_shape**. Будем рассматривать разделение на блоки только для размерностей **C**, **H**, **W**, сохраняя размерность **N** неизменной. Иллюстрация разделения на фрагменты для простого примера показана ниже (рис. 1).

```
front_shape = [input_shape[d] // block_shape[d] \
               for d in range(len(input_shape))]

fC = front_shape[0]; fH = front_shape[1]; fW = front_shape[2]
bC = block_shape[0]; bH = block_shape[1]; bW = block_shape[2]
```

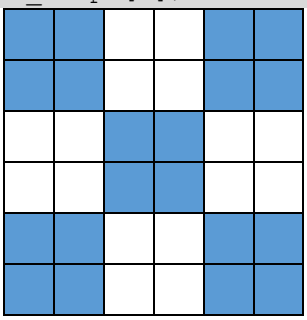

Рис. 1. Разделение на фрагменты размера 1x2x2x1 (переменная **block_shape**) для входных данных формы 1x6x6x1 (переменная **input_shape**). В результате деления имеется $1/1 * 6/2 * 6/2 * 1/1 = 9$ фрагментов. Переменная **front_shape** имеет значение (1/1, 6/2, 6/2, 1/1) = (1, 3, 3, 1).

Для выполнения разделения далее реализуется слой **Split**. Слой имеет параметры размера фрагмента и набора индексов перестановки размерностей после разделения данных. Форму данных до разделения можно представить в виде **(fC * bC, fH * bH, fW * bW, N)**. Непосредственно после разделения форма данных становится **(fC, bC, fH, bH, fW, bW, N)**. Количество вертикальных последовательностей, которые необходимо сформировать, равно **fW**. Длина каждой последовательности **fH**. Размер элемента **(fC * bC) * bH * bW**. Порядок размещения данных в памяти необходимо привести к виду: **(fW, fC, bC, bH, bW, fH, N)**. Для перемещения данных в памяти укажем порядок перестановки индексов размерностей.

```
Split(block_shape=(bC, bH, bW),
      shuffle_dim=(4, 0, 1, 3, 5, 2, 6))
```

Для объединения фрагментов в последовательности реализуем слой **Reshape**. Параметром слоя является новая форма данных. Предполагается, что размер данных не меняется. Значение одного параметра может быть **-1**, что означает автоматическое вычисление его значения.

```
Reshape((fW, (fC * bC) * bH * bW, fH, -1))
```

После этих преобразований форма данных соответствует той, которую ожидают последующие слои. Данные представляют собой вектор последовательностей фрагментов.

Вертикальный обход изображения. На этом шаге создаются рекуррентные слои, обрабатывающие вертикальные последовательности фрагментов. Результаты обработки всех слоев объединяются. Количество слоев совпадает с количеством последовательностей и равно **fW**.

Для объединения результатов рекуррентных слоев будем использовать контейнер **MergeBroadcast** [11], представленный среди инструментов neon. Он агрегирует в себе несколько подсетей, выполняющих последовательные вычисления. Входные данные передаются в подсети в неизменном виде, обрабатываются, и результаты конкатенируются. Контейнер также является слоем. Создание контейнера выполняется следующим образом:

```
branches = [ [layers...], [layers...], containers...]
MergeBroadcast(branches, 'stack')
```

Здесь создается контейнер, объединяющий несколько подсетей. Способ конкатенации выходов указан как **stack**, что означает простую конкатенацию без учета формы данных.

Данные, передающиеся в подсети, представлены в виде массива последовательностей. Для разделения сформированных последовательностей между рекуррентными слоями реализуем слой **Extract**. Слой извлекает данные, относящиеся к соответствующей подсети. Параметрами слоя являются индекс размерности во входных данных и индекс элемента для извлечения. Слой используется следующим образом:

```
Extract(dim=0, indices=[i])
```

После извлечения указанного элемента (последовательности) из входного тензора получим данные, форма которых соответствует ожидаемой в рекуррентном слое.

Создание рекуррентного слоя, выполняющего обход последовательности одновременно в двух направлениях, можно выполнить следующим образом:

```
BiRNN(output_size=256, init=GlorotUniform(), activation=Tanh())
```

В этом фрагменте кода создается слой, имеющий 256 нейронов. Форма входных данных **(I, L * N)** или **(I, L, N)**, где **I** – размер элемента последовательности, **L** – длина последовательности, **N** – размер пачки. Каждый нейрон имеет 2 выхода, поэтому форма выходных данных **(output_size * 2, L * N)**, где **output_size** – количество нейронов. Для инициализации весов и состояний использовано равномерное распределение с границами, основанными на количестве параметров слоя. Функция активации гиперболический тангенс **Tanh**. Дополнительную информацию о параметрах слоя можно найти в документации [10].

Вынесем параметры создания рекуррентных слоев в аргументы функции **SpatialRNN**. Для этого определим аргументы **RNN** и **RNN_params**, являющиеся, соответственно, функцией создания слоя и словарем параметров этой функции.

Создание подсетей, выполняющих вертикальный обход, реализуется так, как показано ниже.

```
layers_vertical = []
for i in range(fW):
    RNN_size = RNN_params['output_size']
    layers_vertical.append([
        Extract(dim=0, indices=[i]),
        RNN(**RNN_params),
        Reshape((2 * RNN_size, -1)) # workaround for MergeBroadcast
                                      incompatibility with RNN shape
    ])
MergeBroadcast(layers_vertical, 'stack')
```

Слой **Reshape** используется для обхода несовместимости контейнера **MergeBroadcast** со слоем **BiRNN** во время фазы обратного распространения градиента в версии neon 2.6.0. После объединения результатов данные имеют вид **(fW, 2 * RNN_size, fH * N)**, где **RNN_size** – число нейронов в рекуррентном слое.

Горизонтальный обход изображения. Для выполнения горизонтального обхода изображения потребуется сформировать горизонтальные последовательности. После этапа вертикального обхода данные имеют форму **(fW, 2 * RNN_size, fH * N)**. Горизонтальные последовательности получаются перестановкой размерностей **fW** и **fH**. Для выполнения перестановки реализуем слой **DimShuffle**. Слой имеет один параметр – набор индексов размерностей для перестановки. Перестановка данных выполняется следующим образом:

```
Reshape((fW, 2 * RNN_size, fH, -1))
DimShuffle((2, 1, 0, 3))
Reshape((fH, 2 * RNN_size, fW, -1))
```

Здесь выделяются размерности для перестановки, и выполняется перестановка. Результат приводится к виду, который ожидают последующие слои.

Создание рекуррентных слоев, выполняющих горизонтальных обход, совпадает с представленным выше вариантом для вертикального обхода. После горизонтального обхода данные имеют форму **(fH, 2 * RNN_size, fW * N)**.

Восстановление исходного представления данных. Для обеспечения совместимости блока **SpatialRNN** со слоями neon восстановим исходный формат данных **(C, H, W, N)**.

```
Reshape((fH, 2 * RNN_size, fW, -1))
DimShuffle((1, 0, 2, 3))
```

Таким образом, выходные данные слоя имеют форму **(2 * RNN_size, fH, fW, N)**.

Приведенные выше слои объединим в список, который станет результатом выполнения функции **SpatialRNN**. Построенный блок используется аналогично слоям, представленным в neon. Полный код функции создания рекуррентного блока следующий:

```
def SpatialRNN(input_shape, block_shape, RNN, RNN_params):
    front_shape = [input_shape[d] // block_shape[d] \
                   for d in range(len(input_shape))]

    fC = front_shape[0]
    fH = front_shape[1]
    fW = front_shape[2]

    bC = block_shape[0]
    bH = block_shape[1]
    bW = block_shape[2]

    RNN_size = RNN_params['output_size']

    block_layers = [
        Split(block_shape=(bC, bH, bW), shuffle_dim=(4, 0, 1, 3, 5, 2, 6)),
        Reshape((fW, (fC * bC) * bH * bW, fH, -1))
    ]
    layers_vertical = []
    for i in range(fW):
        layers_vertical.append([
            Extract(dim=0, indices=[i]),
            RNN(**RNN_params),
            Reshape((2 * RNN_size, -1)) # workaround for MergeBroadcast
                                        incompatibility with RNN shape
        ])  # vertical image traversal
    block_layers.append(MergeBroadcast(layers_vertical, 'stack'))

    block_layers.extend([
        Reshape((fW, 2 * RNN_size, fH, -1)),  # vertical block output shape
        DimShuffle((2, 1, 0, 3)),
        Reshape((fH, 2 * RNN_size, fW, -1)),
    ])
    layers_horizontal = []
    for i in range(fH):
        layers_horizontal.append([
            Extract(dim=0, indices=[i]),
            RNN(**RNN_params),
            Reshape((2 * RNN_size, -1)) # workaround for MergeBroadcast
                                        incompatibility with RNN shape
        ])  # horizontal image traversal
    block_layers.append(MergeBroadcast(layers_horizontal, 'stack'))

    block_layers.extend([
        Reshape((fH, 2 * RNN_size, fW, -1)),  # horizontal block output shape
```

```
        DimShuffle((1, 0, 2, 3))
    ])

    return block_layers
```

Параметры **input_shape** и **block_shape** – кортежи из 3 чисел типа **int**, определяющие размеры входа блока и размеры фрагментов. Параметры **RNN** и **RNN_params** соответствуют типу рекуррентного слоя и словарю его параметров.

3.1.5 Создание дополнительных слоев

Для работы функции **SpatialRNN** необходимо реализовать несколько новых слоев. Такими слоями являются **Split**, **Reshape**, **DimShuffle** и **Extract**. Здесь приведем реализацию слоев **Split** и **Extract**. Слои **Reshape** и **DimShuffle** являются составными частями слоя **Split**.

Создание базового типа слоя, изменяющего форму данных. Все слои в neon являются наследниками типа **Layer** [12] или его потомков. Создадим базовый тип слоя, выполняющего изменение формы данных, **ShapeTransform**. Предполагается, что у наследников этого класса операции прямого и обратного прохода не изменяют значения в массиве данных, только его форму. В классе необходимо реализовать конструктор, функцию инициализации параметров слоя, функции прямого и обратного прохода. Реализация конструктора выглядит следующим образом:

```
from neon.layers import Layer

class ShapeTransform(Layer):
    def __init__(self, name=None):
        super(ShapeTransform, self).__init__(name)
        self.owns_output = False
```

Опишем вспомогательные функции, определяющие формы входных и выходных данных с учетом размера пачки. Переменные **in_shape** и **out_shape** хранят форму данных без учета пачки, **in_shape_t** и **out_shape_t** учитывают размер пачки.

```
    def _get_total_in_shape(self, in_shape):
        # Remember batch size
        in_shape_t = None
        if isinstance(in_shape, tuple):
            if len(in_shape) == 2:
                in_shape_t = (in_shape[0], in_shape[1] * self.be.bsz)
            else:
                in_shape_t = tuple([d for d in in_shape] + [self.be.bsz])
        else:
            in_shape_t = (in_shape, self.be.bsz)
        return in_shape_t

    def _get_out_shape(self, out_shape_t):
        # Forget batch size
        out_shape = None
        if len(out_shape_t) == 2:
            out_shape = (out_shape_t[0], out_shape_t[1] // self.be.bsz)
        else:
            out_shape = tuple([int(d) for d in out_shape_t[:-1]])
        return out_shape
```

Метод инициализации параметров слоя. Здесь инициализируются параметры слоя, зависящие от формы входных данных. К таким параметрам относятся формы входов и выходов. Входным параметром функции является форма объекта на входе. Этим объектом может быть элемент набора данных или предыдущий слой. Введем функцию инициализации, предназначенную для переопределения наследниками класса. Эта функция должна установить значение поля **out_shape_t** на основе значения поля **in_shape_t**. Функция в базовом классе выполняет инициализацию полей **in_shape**, **in_shape_t** и **out_shape**.

```
    def _configure_shape(self, in_shape_t):
        raise NotImplementedError

    def configure(self, in_obj):
        super(ShapeTransform, self).configure(in_obj)

        self.in_shape_t = self._get_total_in_shape(self.in_shape)
        self._configure_shape(self.in_shape_t)
        self.out_shape = self._get_out_shape(self.out_shape_t)

        return self
```

Метод прямого и обратного прохода. Их реализация выносится во внутренние функции для изменения наследниками. Результаты вычислений сохраняются в полях объекта. **`Inputs`** – тензор размера входов текущего слоя (поле **`in_shape_t`**). **`Error`** – тензор размера выходов текущего слоя (поле **`out_shape_t`**).

```
    def _fprop(self, inputs):
        raise NotImplementedError

    def fprop(self, inputs, inference=False):
        self.inputs = inputs
        self.outputs = self._fprop(self.inputs)
        return self.outputs

    def _bprop(self, error):
        raise NotImplementedError

    def bprop(self, error, alpha=1.0, beta=0.0):
        self._bprop(error)
        return self.deltas
```

Таким образом, в классах-наследниках потребуется определить 3 функции. Наследниками этого класса являются **`Split`**, **`DimShuffle`**, **`Reshape`**, **`Extract`**.

Слой Split. Слой выполняет операцию разделения данных на блоки. Параметрами слоя являются размер блока и порядок перестановки размерностей после разделения. Во время обратного прохода потребуется восстановить исходную форму данных. Для корректного восстановления потребуется обращение перестановки размерностей. В примере далее автоматически вычисляется обратная перестановка. Конструктор слоя имеет следующую реализацию:

```
def inverse_permutation(permutation):
    result = [0] * len(permutation)
    for i, v in enumerate(permutation):
        result[v] = i
    return result

class Split(ShapeTransform):
    def __init__(self, block_shape=None,
                 shuffle_dim=(0, 2, 4, 1, 3, 5, 6), name=None):
        super(Split, self).__init__(name)

        self.block_shape = block_shape
        self.shuffle_dim = shuffle_dim
        self.inverse_shuffle = inverse_permutation(shuffle_dim)
```

Метод инициализации параметров, зависящих от входных данных. Введем вспомогательную функцию, определяющую форму данных непосредственно после разделения. Для входа формы **`(C, H, W, N)`** и размера блока **`(Cb, Hb, Wb)`** результат разделения имеет форму **`(Co, Cb, Ho, Hb, Wo, Wb, N)`**.

```
    def make_internal_shape(self, in_shape_t):
```

```
        internal_shape = in_shape_t[:-1]
        tiled_internal_shape = \
            [0] * 2 * len(internal_shape) + [in_shape_t[-1]]
        for d in range(len(internal_shape)):
            tiled_internal_shape[2 * d + 0] = \
                internal_shape[d] // self.block_shape[d]
            tiled_internal_shape[2 * d + 1] = self.block_shape[d]
        return tuple(tiled_internal_shape) # (Co, Cb, Ho, Hb, Wo, Wb, N)
```

Форма выходных данных получается перестановкой элементов формы данных, полученной после разделения.

```
    def _configure_shape(self, in_shape_t):
        self.internal_shape_t = self.make_internal_shape(in_shape_t)
        self.out_shape_t = \
            tuple([self.internal_shape_t[d] for d in self.shuffle_dim])
```

Метод прямого и обратного прохода. Операцию разделения данных можно представить как композицию операций интерпретации формы (**`Reshape`**) и перестановки размерностей (**`DimShuffle`**). neon в версии 2.6.0 не поддерживает перестановку размерностей у тензора. Используем модуль NumPy для перестановки. Потребуется скопировать данные с устройства в основную память, переставить размерности, и вернуть данные на устройство.

```
import numpy as np

def transpose_inplace(array, shuffle_dim):
    transposed = array.get().transpose(shuffle_dim)
    array = array.dimension_reorder(shuffle_dim)
    array[:] = np.ascontiguousarray(transposed)
    return array

    def _fprop(self, inputs):
        inputs = inputs.reshape(self.internal_shape_t)
        transpose_inplace(inputs, self.shuffle_dim)
        return inputs
```

Во время обратного прохода операции выполняются в обратном порядке.

```
    def _bprop(self, error):
        transpose_inplace(error, self.inverse_shuffle)
        self.deltas = error.reshape(self.in_shape_t)
        return self.deltas
```

Слой Extract извлекает срез тензора с входными данными. Параметрами слоя являются размерность (ось) и набор индексов для извлечения из входных данных. Слой отличается от других тем, что должен владеть данными входов и градиентов, т.к. он является первым слоем в подсетях контейнера **`MergeBroadcast`**. Конструктор класса реализуется следующим образом:

```
class Extract(ShapeTransform):
    def __init__(self, dim=None, indices=None, name=None):
        super(Extract, self).__init__(name)

        self.dim = dim
        self.indices = indices
        self.owns_output = True
        self.owns_delta = True
```

Метод инициализации параметров, зависящих от входных данных. Выходные данные слоя имеют форму указанного среза входных данных.

```
    def _configure_shape(self, in_shape_t):
        self.out_shape_t = \
            tuple([len(self.indices)] + list(in_shape_t[self.dim + 1:]))
```

Метод прямого и обратного прохода. Во время прямого прохода извлекается срез входных данных по первому индексу. В версии neon 2.6.0 не поддерживается срез тензора на GPU для более, чем одного индекса, поэтому ограничим реализацию лишь одним индексом. Для реализации рекуррентного блока этого достаточно.

На шаге прямого прохода выполним копирование среза входных данных в выходной массив.

```
    def _fprop(self, x):
        self.outputs[:] = x[self.indices[0]]
        return self.outputs
```

Во время обратного прохода выполняется копирование пришедших градиентов в соответствующий срез тензора градиентов слоя.

```
    def _bprop(self, error):
        deltas_view = self.deltas.reshape(self.in_shape_t)
        error_view = error.reshape(self.out_shape_t)
        deltas_view[self.indices[0]][:] = error_view
        return self.deltas
```

Слои Reshape и DimShuffle реализуются аналогично слою **`Split`**, поэтому не рассматриваются здесь подробно.

Версия neon 2.6.0 имеет ряд неточностей в реализации типов **`BiRNN`**, **`Sequential`** и **`MergeBroadcast`**, приводящих к некорректной работе функции **`SpatialRNN`**. Реализации всех описанных ранее слоев, а также исправленные реализации типов neon представлены в файле: **`Practice/models/layers.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 предыдущей 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. Тестирование модели. Подразумевает прямой проход нейросетевой модели для тестового набора данных и определение качества работы построенной модели.
6. Сохранение вывода 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`** [13]:

```
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** [14]. 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 неизменном виде. В случае задач классификации neon требует предоставления данных в one-hot представлении. При этом 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. Хранение такого представления в наборе данных было бы избыточным. 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 описаниями моделей, создадим модель посредством вызова соответствующей функции.

```
import models

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

3.2.5 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). Используем его для оптимизации параметров модели.

```
from neon.optimizers import GradientDescentMomentum

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

В приведенном фрагменте кода создается объект оптимизатора, реализующий вариацию алгоритма градиентного спуска с инерцией (Nesterov's Accelerated Gradient, NAG). Параметр множителя шага в антиградиентом направлении (learning rate) установлен в значение 0.01. Дополнительно используется L2-регуляризация параметров модели (weight decay) с коэффициентом 0.0005. Полный перечень параметров алгоритма приведен в документации [16].

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.

3.2.6 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.

3.2.7 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.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.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`** [17]. 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')
```

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.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.

Полные исходные коды примера можно найти в материалах данного курса:

`Practice5_rnn/main_classify.py, models/rnn_models.py, models/layers.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: слои [http://neon.nervanasys.com/docs/latest/layers.html].
6. Документация Intel® neon™ Framework: слой объединения данных [http://neon.nervanasys.com/docs/latest/layers.html#summary-layers].
7. Документация Intel® neon™ Framework: простой рекуррентный слой [http://neon.nervanasys.com/docs/latest/generated/neon.layers.recurrent.Recurrent.html#neon.layers.recurrent.Recurrent].
8. Visin F. et al. Renet: A recurrent neural network based alternative to convolutional networks // arXiv preprint arXiv:1505.00393. – 2015. [https://arxiv.org/pdf/1505.00393]
9. Visin F. et al. Reseg: A recurrent neural network-based model for semantic segmentation // Computer Vision and Pattern Recognition Workshops (CVPRW), 2016 IEEE Conference on. – IEEE, 2016. – С. 426-433. [http://openaccess.thecvf.com/content_cvpr_2016_workshops/w12/papers/Visin_ReSeg_A_Recurrent_CVPR_2016_paper.pdf]
10. Документация Intel® neon™ Framework: слой Bidirectional RNN [http://neon.nervanasys.com/docs/latest/generated/neon.layers.recurrent.BiRNN.html#neon.layers.recurrent.BiRNN]
11. Документация Intel® neon™ Framework: контейнер MergeBroadcast [http://neon.nervanasys.com/docs/latest/generated/neon.layers.container.MergeBroadcast.html#neon.layers.container.MergeBroadcast]
12. Документация Intel® neon™ Framework: базовый тип слоя [http://neon.nervanasys.com/docs/latest/generated/neon.layers.layer.Layer.html#neon.layers.layer.Layer]
13. Документация Intel® neon™ Framework: инициализация [http://neon.nervanasys.com/docs/latest/generated/neon.backends.gen_backend.html#neon.backends.gen_backend].
14. Документация Intel® neon™ Framework: тип HDF5Iterator [http://neon.nervanasys.com/docs/latest/generated/neon.data.hdf5iterator.HDF5Iterator.html#neon.data.hdf5iterator.HDF5Iterator].
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 NeonArgparser [http://neon.nervanasys.com/docs/latest/generated/neon.util.argparser.NeonArgparser.html#neon.util.argparser.NeonArgparser].