Nizhny Novgorod State University
Institute of Information Technologies, Mathematics and Mechanics
Department of Computer Software and Supercomputer Technologies

# Educational course «Introduction to deep learning using the Intel® neon™ Framework»

# Practice №5 The development of recurrent neural networks using the Intel® neon™ Framework

*Supported by Intel*

*Zhiltsov Maxim*

Nizhny Novgorod
2018

# Content

# 1 Introduction

This practice is aimed at recurrent neural networks studying and development of such models with the Intel® neon™ Framework. Recurrent neural networks use being demonstrated on the problem of classifying a person's sex from a photo based on the IMDB-WIKI [4] dataset. The development sequence shown in this tutorial can be used to solve other classification problems if the functions for loading and reading data in the format required by the Intel® neon™ Framework are implemented.

# 2 Guidelines

## 2.1 Goals and tasks

*The goal of this practice is to study general principles of constructing recurrent neural networks and develop the recurrent neural networks using the Intel® neon™ Framework to solve the problem of classifying a person's sex by a photo.*

To achieve goals following actions should be completed:

1. Study general principles of recurrent neural networks work and construction.
2. Study the the Intel® neon™ Framework tools related to recurrent neural networks.
3. Create an executable script for deep neural networks training and testing on image classification tasks.
4. Develop a recurrent neural network using the Intel® neon™ Framework.
5. Train the model and evaluate the quality of the classification.

## 2.2 Practice structure

This practice demonstrates the general scheme of the development and application of recurrent neural networks in the Intel® neon™ Framework. A recurrent neural network development example is represented. Initially, test and train data are loaded. Further, an example of the recurrent network is constructed. A script being developed that enables network training and testing, as well as saving results of problem solving. The work is carried out on the example of the problem of classifying a person's sex from a photo. IMDB-WIKI [4] is used as a dataset.

## 2.3 Recommended study sequence

The recommended study sequence is as follows:

1. Study tools of working with recurrent neural networks in the neon framework. Lecture materials and documentation of the neon framework can be used.
2. Develop a model of a recurrent neural network.
3. Load train and test data.
4. Develop a script for training and testing the constructed model.
5. Save classification results.

# 3 Manual

## 3.1 Recurrent neural network implementation

### 3.1.1 Recurrent layer

Model construction is consists of defining a neural network structure and selecting a cost function, which is used during model optimization. The network model allows to represent the sequence of transforms on the input data. The main component of the model is the layer. neon provides access to a variety of typical layers, a complete list of layers can be found in the documentation [5].

Recurrent layers have several significant differences from convolutional and fully-connected layers. Recurrent layers are purposed for input sequences handling and have an internal state, which is updated during sequence traversal. Processing of each input sequence element involves producing of layer outputs with regard to current state, and internal state updating. The output count of recurrent layers depends on input sequence length and hidden neurons count. Recurrent layer output data shape is

**(hidden_outputs, sequence_length * batch_size)**, where **hidden_outputs** is a number of outputs at the hidden layer, **sequence_length** is a length of an input sequence, **batch_size** is an input batch size. The layer outputs for a sequence can be merged in few ways, described in neon documentation [6].

### 3.1.2 Recurrent layer in the Intel® neon™ Framework

The simple recurrent layer can be created in a following way:

```
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())
```

The simple recurrent layer with 10 hidden neurons is created by this code fragment. Each neuron is fully-connected with outputs of the previous layer or a dataset entry and has a single output value. The output value for a given input sequence element computed as follows:

$$o_i = F(Uh_{i-1} + Wx_i + b),$$

where $h_{i-1}$ is a layer internal state at the step $i-1$, $U$ and $W$ are weight matrices, $x_i$ is the $i$-th input sequence element, $b$ is a bias vector, $F$ is an activation function. Layer internal state is described as follows: $h_i = o_i$. Layer weights and internal states are initialized by values drawn from Gaussian distribution with mean 0 and standard deviation 0.1. Biases for neurons (vector $b$ above) are automatically initialized with a constant value 0. The activation function used is hyperbolic tangent **Tanh**. Complete information about layer parameters is provided in documentation [7].

### 3.1.3 The development of the recurrent neural network

To solve the task of human gender recognition from images we develop a recurrent model. General ideas of model architecture being used are presented in [8] and [9]. The proposed approach is to use recurrent layers in combination with the convolutional ones. Recurrent layers are used for high-level spatial information extraction while the convolutional layers are used to extract low-level spatial features. The model receives a single image as an input and outputs image class. Note that input data in this task has no essential sequences like, for example, video frames in video processing tasks. Instead, input sequences for recurrent layers are generated inside the model processing the input image from activations of convolutional layers and are not stored in dataset. Built-in the neon capabilities are not sufficient to implement described approach, so new layers are required.

The model implementation of the represented approach can be created in a following way:

```
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)
```

This model contains the front layer of non-parametric input data transform imitating division by a standard deviation in the case of large and various enough image dataset. Layer is followed by 4 convolutional layers with different parameters and **`ReLU`** activation functions. Each convolutional layer is followed by a pooling layer. Pooling layers choose a maximum value in each input block of size 3x3 and decrease feature map size by 2 times for each dimension (note the **`stride`** parameter). After convolutional part follow two recurrent blocks created by **`SpatialRNN`** function, which will be described later. These blocks reduce feature map size by 2 times. A last part of the model is a classifier. This classifier consists of the convolutional and pooling layers. The convolutional layer has 2 filters conforming to number of categories in the task. The pooling layer after the convolutional one computes the average of each feature map received from the convolutional layer providing vector of class log-probabilities. The cost function specified for model parameters optimization is a binary cross-entropy. Batch normalization is used to improve the convergence properties of optimization algorithms.

For convenience we put model generation function to separate source file called **rnn_models.py**. The model can be constructed using the following code:

```
import rnn_models

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

Additional tools, related to the model construction, we put to the file **layers.py**.

### 3.1.4 The recurrent block description

Let us consider recurrent blocks represented in the model described earlier. A single recurrent block is defined by **SpatialRNN** function. Function implements the following actions:

- Splits input data into fragments of the specified size.
- Orders fragments into sequences.
- Processes each sequence with a separate recurrent layer.
- Merges the results of sequences processing.
- Restores the input data format.

These actions are performed twice. In the first time the fragments are ordered so an image is traversed vertically by recurrent layers in the block. The second time an image is traversed horizontally. Traversals are performed in direct and reverse order simultaneously by bidirectional recurrent layers [10]. The idea of the recurrent block is that the usage of internal states by recurrent layers during traversals can be exploited to extract high-level spatial features based on low-level feature extracted from an input image.

**Splitting input data into fragments.** Input data for the recurrent blocks is provided by previous layers. The data in the neon is presented by 4D numerical tensors of the shape **(C, H, W, N)**, where **C** is a channel count, **H** and **W** are image height and width, **N** is a batch size. The data splitting is performed into fragments of the specified size. Each input dimension is able to have a separate fragment size, so fragment size is described by a tuple with 4 elements. For instance, if we have an input tensor of the shape $(32, 64, 64, 1)$ and the fragment size is $(2, 4, 4, 1)$, so the data shape after the splitting will be $(32/2, 64/4, 64/4, 1/1)$ with a total number of fragments $32/2 * 64/4 * 64/4 * 1/1$.

The next step is to order fragments into sequences. First time the vertical image traversal is performed, so the fragments are ordered to construct vertical sequences.

Let us consider an implementation of this step. We define variables **input_shape** (input data shape), **block_shape** (shape of the fragment) and **front_shape** (shape after splitting). Variables **input_shape** and **block_shape** are **SpatialRNN** function arguments. Implementation will support separation only for dimensions **C**, **H** and **W**, so the dimension **N** is considered fixed. Define additional variables to simplify descriptions. An illustration of fragment split is shown below (fig. 1). The code is as follows:

```
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]
```

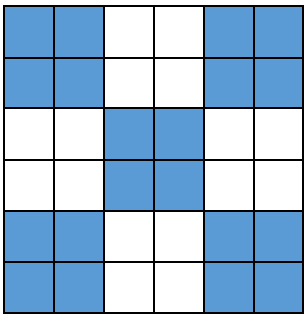

Fig. 1. Splitting on fragments of the size 1x2x2x1 (**block_shape**) of the input data of the shape 1x6x6x1 (**input_shape**). After the split there are $1/1 * 6/2 * 6/2 * 1/1 = 9$ fragments. The **front_shape** variable is (1/1, 6/2, 6/2, 1/1) = (1, 3, 3, 1)

To perform splitting during the computations the **Split** layer is implemented. The implementation will be considered later. This layer has two parameters: the first one is a fragment shape and the second one is a dimensions' transposition, which is performed after the split. The data shape before the split can be expressed as **(fC * bC, fH * bH, fW * bW, N)**. Immediately after the splitting the data shape becomes **(fC, bC, fH, bH, fW, bW, N)**. Count of the vertical sequences is equal **fW**. Length of each sequence is **fH**. Sequence elements' size is **(fC * bC) * bH * bW**. Data should be reordered in memory to be of the shape **(fW, fC, bC, bH, bW, fH, N)**, which is expected by following layers. The second layer parameter describes the new order of data dimensions obtained by the split.

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

To form sequences we implement **Reshape** layer. This layer has a single parameter, which contains the new shape of a data. It is expected that the data size is not changed during reshaping. One of the dimensions can be equal to **-1** to be automatically computed.

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

After these manipulations data has the shape expected by following layers. This shape can be interpreted as an array of fragment sequences.

**Vertical image traversal.** This step includes processing of created vertical fragment sequences by recurrent layers. Processed results are merged. Recurrent layers' count matches sequence count and equals **fW**.

In the neon, results of computations, performed by multiple layers, can be merged by **MergeBroadcast** [11] container. This container consists of multiple sub-networks. The data passed to each sub-network is unchanged (broadcasted) and processed by them. Processing results from sub-networks are concatenated to form container output. There are different concatenation modes, some of which based on sub-networks output shapes. A container itself is also a layer. Container can be created in the following way:

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

In the example above a container is created. This container merges outputs from sub-networks by a simple concatenation (**stack** parameter), that means only output sizes from sub-networks are considered during the concatenation, not their shapes.

Data passed to the container sub-networks currently forms an array of sequences. Sequences should be separated between sub-networks to be processed independently by recurrent layers. To separate the data the **Extract** layer is implemented. The layer extracts the data, related to the specific sub-network, from the input array. This layer has two parameters: input data dimension and list of element indices to extract. **Extract** layer is used in the following way:

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

After sub-network data is extracted from the input tensor, the data has a shape expected by following recurrent layers.

The input sequences are to be processed in direct and reverse order simultaneously. A bidirectional recurrent layer **BiRNN**, presented in the neon, is able to do such sequence traversal. To create this layer the following code is used:

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

In the code fragment the bidirectional recurrent layer is created. The layer expects input data to be of shape **(I, L * N)** or **(I, L, N)**, where **I** is a sequence element size, **L** is a sequence length, **N** is a batch size. The layer has 256 hidden neurons. Each layer neuron produces two outputs, so an output shape of layer is **(output_size * 2, L * N)**, where **output_size** is a layer neurons' count. Weights and internal states are initialized by uniformly distributed values with boundaries based on the layer parameter count. The specified activation function is a hyperbolic tangent **Tanh**. Additional information on the layer usage is represented in the neon documentation [10].

We extract the layer parameters to **SpatialRNN** function arguments. The layer type to be created is specified by **RNN** argument and the layer parameters are described by **RNN_params** argument.

The following code demonstrates the implementation of the vertical traversal.

```
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')
```

The **Reshape** layer is used as a workaround for **MergeBroadcast** incompatibility with **BiRNN** layer during gradients backpropagation in neon 2.6.0. After sub-network results are merged, the data shape is **(fW, 2 * RNN_size, fH * N)**, where **RNN_size** is a neuron count in a recurrent layer.

**Horizontal image traversal.** To complete such traversal a horizontal sequences should be created from the data. After the vertical traversal, the data shape is **(fW, 2 * RNN_size, fH * N)**. Horizontal sequences are obtained by **fW** and **fH** dimensions swap. To realize it the **DimShuffle** layer is implemented. The layer has a single parameter that is the index permutation to be performed. Horizontal sequences are obtained in the following way:

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

In this code the data dimensions are distinguished and then swapped. The result is reshaped to be a shape expected by recurrent layers.

Recurrent layers performing the horizontal traversal are created in the same manner as vertical ones above. After the traversal the data shape is **(fH, 2 * RNN_size, fW * N)**.

**Initial data format restoration.** To make **SpatialRNN** block consistent with build-in neon layers the initial data format should be restored. Currently, the data shape is **(fH, 2 * RNN_size, fW * N)** which is to be transformed to **(C, H, W, N)**.

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

After these steps the recurrent block has output shape **(2 * RNN_size, fH, fW, N)**.

The layers in the block should form a list. This list is a **SpatialRNN** function return value. The block invented is used the same way as built-in neon layers. The complete code of SpatialRNN function is as follows:

```
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
```

Function parameters **`input_shape`** and **`block_shape`** are tuples of 3 integers (**`int`**), defining the block inputs shape and a fragment shape. Arguments **`RNN`** and **`RNN_params`** are correspond to the recurrent layer type and its parameters.

### 3.1.5 Implementation details of the additional layers

Recurrent block function implementation relies on few additional layers. These layers are **`Split`**, **`Reshape`**, **`DimShuffle`** and **`Extract`**. This section contains implementation details of **`Split`** and **`Extract`** layers, since **`Reshape`** and **`DimShuffle`** layers are parts of **`Split`** layer implementation.

**The definition of basic shape transformation layer.** All layers in neon are supposed to be subclasses of **`Layer`** [12] class or its subclasses. Here we define **`ShapeTransform`** – the basic layer for data shape transformations. This layer supposed only to make transformations that keep data values unchanged while data shape may change during forward and backward passes. Subclasses of this class should implement a constructor, initialization function, and forward and backward passes. The class constructor is defined as follows:

```
from neon.layers import Layer

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

We create an additional functions helping to define input and output data shapes of the layer with respect to batch size. Variables **`in_shape`** and **`out_shape`** are to contain shapes without batch size, while **`in_shape_t`** and **`out_shape_t`** variables contain shapes with batch size.

```
    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
```

**The function to initialize layer parameters.** This function is used to initialize layer parameters based on input data shape. For instance, such parameters are layer input shape and output shape. Function has one parameter describing a layer input data. Possible layer inputs are dataset entries and other layers. We introduce a function to be overridden by class successors, which has a main purpose of initialization **`out_shape_t`** class variable on the base of .**`in_shape_t`** value. The base class initialization function sets values for variables **`in_shape`**, **`in_shape_t`** and **`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
```

**Forward and backward pass functions.** These functions are called during the forward and backward passes to transform data. We extract two functions to be redefined in subclasses, leaving a common code

in the base class. The **_fprop** method receives input data tensor **input** of shape **in_shape_t**. The **_bprop** method takes gradients tensor **error** of shape **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
```

This way each subclass should define 3 functions. Class is derived by 4 classes **Split**, **DimShuffle**, **Reshape** and **Extract**.

**Split layer implementation.** The layer is purposed to split an input data by blocks. There are 2 layer parameters: block size and data dimensions permutation. During the backward pass layer transformations are to be applied in a reverse order. This involves an inverse permutation of data dimensions. In a following code example the inverse permutation is computed automatically.

```
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)
```

**The function to initialize layer parameters.** We introduce an auxiliary function to determine data shape immediately after the split. In a case of the input shape **(C, H, W, N)** and block shape **(Cb, Hb, Wb)** data shape after the split is **(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)
```

Output data shape is obtained by a transposition of data shape after the split.

```
    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])
```

**Forward and backward pass functions.** Layer transformation can be expressed as a composition of **`Reshape`** and **`DimShuffle`** operations. neon 2.6.0 does not support arbitrary tensor dimensions permutations, so we use NumPy module instead. In the implementation the data is copied to host memory, transposed, and returned back to device memory.

```
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
```

The backward pass applies the inverse transformations to the data.

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

**Extract layer implementation.** The layer extracts a slice of input data tensor. Layer parameters are data dimension to extract from and set of indices to be extracted. This layer slightly differs from others in its usage, so it must own its input data and gradients. This is due to the layer is being used as a first layer in **`MergeBroadcast`**'s sub-networks. Class constructor is implemented as following:

```
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
```

**The function to initialize layer parameters.** Layer output data has a shape of tensor slice specified. This shape can be obtained by the following actions:

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

**Forward and backward pass functions.** During the forward pass input data slice is extracted. neon 2.6.0 supports tensor slicing only for one index on GPUs, so layer implementation is constrained by this restriction. A single index slicing is enough for **`SpatialRNN`** implementation. Forward pass transformation copies input data slice to layer outputs.

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

During the backward pass input gradients are copied to corresponding layer gradients tensor slice.

```
    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
```

**Reshape and DimShuffle layers** are implemented the same way as **`Split`** layer, so are not described here in details.

neon 2.6.0 has a number of incompatibilities in implementations of **BiRNN**, **Sequential** and **MergeBroadcast** types, which leads to incorrect work of **SpatialRNN** block. A complete layer implementations and compatible implementations for the listed neon types are presented in the file **models/layers.py**.

## 3.2 Training and testing the network

### 3.2.1 Script structure

The script for training and testing models consists of several logical parts:

1. Initialization. Provides, in particular, a specification of the device on which the model is trained and tested.
2. Preparing and loading data. Assumes the call of functions developed in the preliminary practice.
3. Creating a model. Provides for the call of functions developed earlier in this practice.
4. Training the model. Assumes setting the optimization method and its parameters, and calling the backpropagation method.
5. Testing the model. It implies the feed forward of the neural network for the test dataset and the calculating the quality of the constructed model.
6. Saving the model output. Saves the network output values for the test dataset.

Let us consider in more detail the development of each logical part of the script.

### 3.2.2 Initialization

Before further usage, the neon framework has to be initialized. You should set the device on which the calculations will be performed, the batch size of training samples, the element data type, and other parameters. Initialization is performed by the function **gen_backend** [13]:

```
from neon.backends import gen_backend

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

### 3.2.3 Loading data

The steps for preparing the data are described in detail in the preliminary practice. After these steps are completed, the data files for the train and test subsets of the IMDB-WIKI dataset must be generated. We assume that the files are located in the **data_wiki** directory and are called **train.h5** and **test.h5**. To load data in HDF5 format, the type **HDF5Iterator** [14] is used. It allows you to load batches from external memory, the batch size is specified when initializing neon. To load the data, you need to create two objects:

```
from neon.data import HDF5Iterator

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

These objects provide access to the elements of the dataset.

**HDF5Iterator** provides the data in the saved format. In the dataset, class labels are stored in a numerical form. In the case of the classification problem, neon requires class labels in one-hot representation, in which the target object category is described not by a number, but by a vector of length equal to the number of categories. The specified vector contains zeros in all elements except one that matches the index of the target class. To automatically convert data from an index representation to one-hot, the type **HDF5IteratorOneHot** is used.

```
from neon.data import HDF5IteratorOneHot

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

### 3.2.4 Creating model

Using the previously created script with model descriptions, construct the model by calling the corresponding function.

```
import rnn_models

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

### 3.2.5 Training model

To train the model, you need to set an optimization algorithm and its parameters. In deep learning, stochastic gradient descent (SGD) is widely used. The following code enables to define optimizer object that implements a Nesterov's Accelerated Gradient (NAG) algorithm.

```
from neon.optimizers import GradientDescentMomentum

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

The learning rate parameter is set to 0.01. In addition, L2-regularization of the model parameters (weight decay) with coefficient 0.0005 is used. A complete list of algorithm parameters is given in the documentation [15].

To train the network, you need to perform the following calls:

```
from neon.callbacks.callbacks import Callbacks

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

Here, training is conducted on the training dataset. Duration of training is equal to 10 epochs. The epoch is equivalent to one complete traversal of the training dataset. To preserve the properties of the stochastic gradient descent algorithm, it is required to ensure random selection of data samples from the set. It was done at the step of mixing the dataset during preprocessing. The **`callbacks`** parameter allows you to specify the functions that will be called during the training. For example, you can organize testing the model at the end of the epoch.

### 3.2.6 Testing model

To assess the classification quality, use the following code:

```
from neon.transforms import Accuracy

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

Here the quality metric is specified. For the classification problem, the **`Accuracy`** is used, this metric reflects the number of correctly classified samples. The specified metric is calculated for the test dataset.

### 3.2.7 Saving classification results

The ultimate goal of training network is to get the network output on some data set. You can use the following code:

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

Saving output to a file can be implemented as follows:

```
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)
```

This code traverses the dataset and network outputs. The results are saved to a file in the format:

```
[confidence of the 1st class, confidence of the 2d class], correct answer
```

## 3.3 Execution of training and testing model

### 3.3.1 Execution without specifying the parameters

It is assumed that a script for training and testing a deep model has been developed. For definiteness, we assume that it is stored in a file called **`main_classify.py`**. The script is launched from the command line:

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

### 3.3.2 Startup parameterization

The startup process has many parameters related to working with neon, a data source, a neural network, training and testing parameters. All these parameters can be fixed in the script, changing them as necessary. Another way is to add command line arguments.

neon provides class for processing and using command line arguments. The type **`NeonArgparser`** [16] is used for processing command line arguments. The following code demonstrates how to create the object and parse arguments.

```
from neon.util.argparser import NeonArgparser

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

After executing this code, the **`args`** variable will contain a set of command line parameters. neon initialization will be performed automatically taking into account the input parameters. You can perform custom settings as follows:

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

After parsing the parameters, the value of the new parameter is extracted as follows:

```
data_root = args.data_root
train_set = HDF5IteratorOneHot(data_root + '/train.h5')
test_set = HDF5IteratorOneHot(data_root + '/test.h5')
```

### 3.3.3 Execution with input parameters

To run the script, you need to set some input parameters. You can run it using the following command:

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

Let us consider in details the list of parameters.

- **`b <cpu, mkl, gpu>`** provides a backend initialization.
- **`e <number>`** sets the number of epochs.
- **`z <number>`** sets the batch size.
- **`data_root <dir>`** specifies the directory containing input data.
- **`serialize <number>`**, **`save_path <name.prm>`** ensures that the model is saved during training after **`<number>`** epochs in a file **`<name.prm>`**.

A full list of available parameters can be obtained by passing the **`--help`** option.

To improve the usability, it is recommended to create a shell-script containing the required command line. In such script, you can additionally perform initialization of environment variables and activation of the virtual environment.

Complete sources of the example can be found in following materials of this course:

**`Practice5_rnn/main_classify.py`**, **`models/rnn_models.py`**, **`models/layers.py`**.

# 4 Literature

## 4.1 Books


1. Haykin S. Neural Networks: A Comprehensive Foundation. – Prentice Hall PTR Upper Saddle River, NJ, USA. – 1998.
2. Osovsky S. Neural networks for information processing. – 2002.
3. Goodfellow I., Bengio Y., Courville A. Deep Learning. – MIT Press. – 2016. – [http://www.deeplearningbook.org].


## 4.2 References


4. IMDB-WIKI dataset [https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki].
5. Intel® neon™ Framework: layers [http://neon.nervanasys.com/docs/latest/layers.html].
6. Intel® neon™ Framework: summary layers [http://neon.nervanasys.com/docs/latest/layers.html#summary-layers].
7. Intel® neon™ Framework: Recurrent layer [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. – C. 426-433. [http://openaccess.thecvf.com/content_cvpr_2016_workshops/w12/papers/Visin_ReSeg_A_Recurrent_CVPR_2016_paper.pdf]
10. Intel® neon™ Framework: bidirectional recurrent layer [http://neon.nervanasys.com/docs/latest/generated/neon.layers.recurrent.BiRNN.html#neon.layers.recurrent.BiRNN]
11. Intel® neon™ Framework: MergeBroadcast container [http://neon.nervanasys.com/docs/latest/generated/neon.layers.container.MergeBroadcast.html#neon.layers.container.MergeBroadcast]
12. Intel® neon™ Framework: base layer type [http://neon.nervanasys.com/docs/latest/generated/neon.layers.layer.Layer.html#neon.layers.layer.Layer]
13. Intel® neon™ Framework: backend initialization [http://neon.nervanasys.com/docs/latest/generated/neon.backends.gen_backend.html#neon.backends.gen_backend].
14. Intel® neon™ Framework: HDF5Iterator [http://neon.nervanasys.com/docs/latest/generated/neon.data.hdf5iterator.HDF5Iterator.html#neon.data.hdf5iterator.HDF5Iterator].
15. Intel® neon™ Framework: GradientDescentMomentum [http://neon.nervanasys.com/docs/latest/generated/neon.optimizers.optimizer.GradientDescentMomentum.html#neon.optimizers.optimizer.GradientDescentMomentum].

16. Intel® neon™ Framework: NeonArgparser [http://neon.nervanasys.com/docs/latest/generated/neon.util.argparser.NeonArgparser.html#neon.util.argparser.NeonArgparser].