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 №3 Application of transfer learning to solve a given problem using the Intel® neon™ Framework

Supported by Intel

Zhiltsov Maxim, Kustikova Valentina

Nizhny Novgorod
2018

Content

1 Introduction

This practice is aimed at studying the technique of the transfer learning of neural networks using the Intel® neon™ Framework [4]. The application of the transfer learning is demonstrated using the example of solving the problem of classifying a person's sex by a photo. The original task is the image classification with a large number of categories (1000 categories), the target one is the classification of the person's sex by a photo (2 categories). The original deep model that is applied to the target task, is the convolutional neural network VGG-16 [5], trained on the ImageNet dataset [6]. IMDB-WIKI [7] is used as the dataset for solving the target task. The manual for transfer learning 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 the general scheme of the transfer learning and apply this approach to solve the problem of classifying a person's sex by a photo using the Intel® neon™ Framework.

To achieve this goal, it is necessary to solve the following tasks:

1. Study the general scheme of transfer learning.
2. Study the tools of the Intel® neon™ Framework to implement the transfer learning.
3. Develop a script for training and testing of deep models to solve the problem of classifying a person's sex by the photo based on transfer learning approach.
4. Develop a model of the convolutional neural network VGG-16 and describe it using the Intel® neon™ Framework.
5. Carry out three types of experiments on transfer learning:
   - using the structure of a deep model constructed to solve the original task, with the aim of training a similar model for solving the target task;
   - using a model constructed to solve the original task as a method of feature extraction in constructing a model that solves the target task;
   - fine-tuning the parameters of the model constructed to solve the original task, with the aim of solving the target task.
6. Train the models and evaluate the quality of the classification.

2.2 Practice structure

The practice demonstrates the general scheme of implementing the transfer learning in the Intel® neon™ Framework. First, the structure of a convolutional network based on VGG-16 is being developed. Then the train and test data are loaded. A script is developed that provides training and testing of the network in accordance with the possible types of experiments on the transfer learning, and the results of solving the problem are also saved. The work is carried out on the example of the problem of classifying a person's sex by a photo. IMDB-WIKI is used as the dataset [4].

2.3 Recommended study sequence

The recommended study sequence is as follows:

1. Study the tools of working with convolutional neural networks in the neon framework. You can use the lecture materials and documentation of the neon framework.
2. Study the technique of transfer learning. Theoretical bases are represented in the lecture material, as well as in the literature suggested in the lecture.
3. Develop a model of a convolutional neural network based on the VGG-16.
4. Load train and test data.
5. Develop a script for training and testing the constructed model based on transfer learning approach.
6. Carry out the experiments on the transfer learning.

3 Manual

3.1 Convolutional neural network

The model construction includes a description of the neural network structure and the assignment of the cost function used during training. The transfer learning implies the presence of a pre-trained model, from which it is necessary to transfer model structure or some of the trained layers to a new model. As a consequence, the new model is similar in structure to the original one. The transfer is meaningful only for parametric layers of the network. To perform the layer transfer from one model to another, the necessary condition is the coincidence of the parameter number in the source layer and in the target layer. As a rule, the correspondence between layers is established by layer identifiers (or names).

To solve the problem of classifying a person's sex by a photo, we use the well-known VGG-16 model [5], designed to classify 224x224 images into 1000 object categories. The structure of the model and the trained model parameters are available in the neon repository [8]. To download the model (~500 MB) you can use the command below.

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

After executing the command, the file **`VGG_D.p`** containing the trained model appears in the current directory.

To solve the classification problem with two categories, we will create a description of a new model based on VGG-16. This requires a description of the original model [9]. The model presented in the repository is stored in the obsolete neon format. The model description, prepared for the transfer learning and adapted for the version of neon 2.6.0, is presented below. Highlighted code fragments correspond to the changes made in VGG-16 to implement the transfer learning. Essentially, the layers corresponding to the classifier have been replaced.

```
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'))
```

The original VGG-16 network has 1000 outputs, so the last layer of the network is changed so that the number of outputs becomes two. Let's pay attention to the layer parameter **name**. This parameter allows you to set the layer identifier. You can see that the value of this parameter is specified for all layers in the network description. When using the transfer learning, the layer identifier is a convenient and flexible way of identifying layers that need to be transferred from the original model to the new one. In the process of copying weights, only layers with the same identifiers are considered. If the layer is not supposed to be copied, you must define a unique identifier for it. The original network is trained for images, the size of which does not exceed 224x224 pixels. Suppose that the input images of the dataset have a different size, for example 256x256. The original network contains parametric layers of two types: convolutional and fully-connected layers. One of the useful properties of convolutional layers is the invariance of the parameter number relative to the size of the input image, so convolutional layers can be copied unchanged. In the case of fully-connected layers, the number of synaptic connections coincides with the size of the input image or the output of the previous layer. When you change the size of the input, the number of parameters also changes. Thus, fully-connected layers cannot be copied. To avoid copying the parameters of fully-connected layers, we define new identifiers for them.

Further, we need to create a list of network layers and a model.

```
from neon.models import Model

layers = []
# the list of layers of the network presented above
# ...
model = Model(layers)
```

As a result, a new model is being constructed that allows solving the classification problem with two categories. The developed model can be trained using weights from the original model.

Define the cost function to optimize the network parameters. In this case, the binary cross-entropy function is used.

```
from neon.transforms import CrossEntropyBinary

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

For convenience, we will place the model construction code in a separate file **tl_models.py**. For the model, we create a separate function of the following form:

```
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
```

The model can be created using the following code:

```
import tl_models

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

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 the target model. Provides for the call of functions developed earlier in this practice.
4. Loading the original model and transferring weights to the target model.
5. Training the target model. Assumes selection of the optimization method, setting its parameters and calling the backpropagation method.
6. Testing the target model. It implies the feed forward of the neural network for the samples of the test dataset and calculating the quality of the constructed model.
7. Save the output of the target model. 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 using neon, you must initialize the framework. 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`** [10]:

```
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`** [11] 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 description, construct the model by calling the corresponding function.

```
import models

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

3.2.5 Transferring target model

neon provides tools for saving and loading models. To load the trained model VGG-16 from a file, use the **`load_obj`** function [12].

```
from neon.util.persist import load_obj

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

Further, we need to copy layers from the trained model to the target one. First, we need to get a set of layers from the target model. This can be done by calling the **`get_description`** [13] method for the model object or by directly accessing the **`layers`** field.

```
pdesc = model.get_description(get_weights=True)
# pdesc['model']['config']['layers'] is a dictionary with descriptions
# of layers and weights

layers = model.layers.layers
# layers is a dictionary with descriptions of layers and weights
```

Let us implement a function for copying weights from one model to another. The function performs the following steps:

- Creates associative arrays of layers of the original model and the target one. The key is the layer name.
- Constructs an array of layers for which the weights are copied.
- For each layer of the target model, checks for a layer in the source model and, if found, copies the weights. To copy weights, we use the method **`load_weights`** [14] for the layer object.
- Returns an array of layers for which weights are copied.

```
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
```

The function is called as follows:

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

3.2.6 Training model

To train the model, we need to set an optimization algorithm and its parameters. neon provides several optimization algorithms [15]. In deep learning, stochastic gradient descent (SGD) [16] is widely used. We use it to optimize the model parameters. As a rule, for copied layers the smaller values of the learning rate are used during transfer learning. neon allows you to specify different optimization algorithms and parameters for each network layer. We will adjust the optimization algorithm so that small values of the learning rate are used for the transferred layers. We use an object of the type **`MultiOptimizer`** [17], which allows to use different parameters of the optimization algorithm. The constructor of **`MultiOptimizer`** takes a parameter that defines optimization algorithms for network layers. The parameter is an associative array with the key of the layer identifier and the value of the optimization algorithm for the layer. If there is no entry for the layer in the array, then the element is used by **`default`** key. Let's designate the optimization algorithm, used by default, as **`GradientDescentMomentum`**. For the copied layers of the network, we will create similar optimizers with the changed learning rate parameter. The list of copied layers was obtained earlier and is stored in the variable **`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)
```

To train the network, you must perform the following actions:

```
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.

3.2.7 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.

3.2.8 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 class
```

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_transfer.py`**. The script is launched from the command line:

```
. .venv/bin/activate
python main_transfer.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`** [18] 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')
```

Let us specify the parameter of the transferred model:

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

We use a new parameter for loading weights on demand:

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

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>**.
- **source_model <path.prm>** provides a file with trained parameters of the original model.

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.

3.3.4 Carrying out different types of transfer learning experiments

Using the structure of a deep model constructed to solve the original task, with the aim of training a similar model for solving the target task. To implement this transfer experiment, it is sufficient to train the developed model with randomly initialized weights. Thus it is necessary to establish uniform value of the learning rate.

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

The weights of the trained model VGG-16 is not imported. It is sufficient not to set the trained model as an input parameter to the script. Below is the command line for running the script for this experiment.

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

Fine-tuning the parameters of the model constructed to solve the original task, with the aim of solving the target task. For this experiment, you need to import the weights of the trained model VGG-16 into the target model and train the parameters of all layers. At the same time for imported layers it makes sense to set small values of the learning rate.

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

In this case, the launch of the script must be performed using the following command:

```
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
```

Using a model constructed to solve the original task as a method of feature extraction in constructing a model that solves the target task. It is required to carry out the transfer learning and to establish for the transferred layers the reduced learning rate. When implementing this experiment, it is necessary to import the weights of the trained model VGG-16 into the target model, fix their values and train the parameters of the fully-connected layers. The learning rate for the transferred layers should be set to zero.

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

The script is launched by the command shown below.

```
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
```

Full sources of the example can be found in the materials of this course:
`Practice3_tl/main_transfer.py` и **`models/tl_models.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. 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. ImageNet Large Scale Image Recognition Challenge [http://www.image-net.org/challenges/LSVRC].
7. IMDB-WIKI dataset [https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki].
8. Intel® neon™ Framework: VGG-16 [https://github.com/NervanaSystems/ModelZoo/tree/master/ImageClassification/ILSVRC2012/VGG]
9. Intel® neon™ Framework: VGG-16, model description [https://github.com/NervanaSystems/ModelZoo/blob/master/ImageClassification/ILSVRC2012/VGG/vgg_neon.py].
10. Intel® neon™ Framework: backend initialization [http://neon.nervanasys.com/docs/latest/generated/neon.backends.gen_backend.html#neon.backends.gen_backend].
11. Intel® neon™ Framework: HDF5Iterator [http://neon.nervanasys.com/docs/latest/generated/neon.data.hdf5iterator.HDF5Iterator.html#neon.data.hdf5iterator.HDF5Iterator].
12. Intel® neon™ Framework: load_obj [http://neon.nervanasys.com/docs/latest/generated/neon.util.persist.load_obj.html#neon.util.persist.load_obj].
13. Intel® neon™ Framework: get_decription [http://neon.nervanasys.com/docs/latest/generated/neon.models.model.Model.html#neon.models.model.Model.get_description].
14. Intel® neon™ Framework: load_weights [http://neon.nervanasys.com/docs/latest/generated/neon.layers.layer.Layer.html#neon.layers.layer.Layer.load_weights].

15. Intel® neon™ Framework: optimization methods [http://neon.nervanasys.com/docs/latest/optimizers.html].
16. Intel® neon™ Framework: GradientDescentMomentum [http://neon.nervanasys.com/docs/latest/generated/neon.optimizers.optimizer.GradientDescentMomentum.html#neon.optimizers.optimizer.GradientDescentMomentum].
17. Intel® neon™ Framework: MultiOptimizer [http://neon.nervanasys.com/docs/latest/generated/neon.optimizers.optimizer.MultiOptimizer.html#neon.optimizers.optimizer.MultiOptimizer].
18. Intel® neon™ Framework: NeonArgparser [http://neon.nervanasys.com/docs/latest/generated/neon.util.argparser.NeonArgparser.html#neon.util.argparser.NeonArgparser].