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 №4 Initial pre-training the weights of the most perspective architectures of fully-connected networks for the subsequent solution of a given problem in supervised style using the Intel® neon™ Framework

Supported by Intel

Zhiltsov Maxim, Kustikova Valentina

Nizhny Novgorod
2018

Content

1 Introduction

This practice is aimed at studying approaches to initial pre-training the deep model parameters using Intel® neon™ Framework. The initial pre-training is implemented by constructing a stack of autoencoders. The application of this approach is demonstrated using the example of solving the problem of classifying a person's sex by a photo. IMDB-WIKI [4] is used as the dataset. The described network development sequence 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 basic approaches to the pre-training the deep model parameters and to pre-train some previously developed fully-connected networks by constructing a stack of autoencoders using the Intel® neon™ Framework.

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

1. Study approaches to initial pre-training the parameters of deep neural networks.
2. Study the general scheme of constructing the stack of autoencoders.
3. Develop a script for training and testing of deep models to solve the problem of classifying a person's sex by the photo.
4. Develop neural network models to train the stack of autoencoders, and describe those using Intel® neon™ Framework.
5. Train the stack of autoencoders and the final deep model, evaluate the quality of the classification.

2.2 Practice structure

The practice demonstrates the general scheme of development and application of the stack of autoencoders in the Intel® neon™ Framework. First, an example is given for developing neural networks necessary for training the stack of autoencoders. A model of a fully-connected network is described for solving the problem of classifying a person's sex by a photo. Then the test and train datasets are loaded. A script is implemented that provides training and testing the autoencoders and the final network, as well as the preservation of the classification results. IMDB-WIKI is used as the dataset [4].

2.3 Recommended study sequence

The recommended study sequence is as follows:

1. Study approaches to initial pre-training the parameters of deep neural networks. You can use lecture materials and recommended literature on the course.
2. Study the general scheme of constructing the stack of autoencoders.
3. Develop models of neural networks necessary for training the stack of autoencoders and the final fully-connected model.
4. Load train and test data.
5. Develop a script for training and testing the constructed model.
6. Save the classification results.

3 Manual

3.1 Development of a network model

Using the stack of autoencoders involves the following steps:

- Development of the target model, for which the stack of autoencoders is constructed.
- Identification of the stages for training the target model and the stack of autoencoders. A separate model is assumed to be trained for each stage.
- Training models corresponding the stack of autoencoders.

– Creating the final model, transfer of the trained parameters from the stack of autoencoders and training the model on the target dataset.

As the target model, we use a fully-connected network with three parametric layers. The network structure is described as follows:

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

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

Note that identifiers of network layers are specified by the **`name`** parameter. The binary cross-entropy is used as the cost function.

Let us consider the stages of network training, on which the stack of autoencoders is constructed. The network has three parametric layers. We will pre-train the parameters of the first two layers, we construct a separate autoencoder for each of them. It is necessary to prepare a description of the network for each of the autoencoders. Creation of the final model based on the stack of autoencoders is performed using the transfer learning.

Autoencoders consist of two main components, one (encoder) of which provides a compressed representation of the input data, the other (decoder) one restores the original representation. The input and output data of the network are the same. We take these features into account when constructing additional networks.

The network that describes the first autoencoder is designed to pre-train the parameters of the first layer of the target network. The input and output data at this stage are images. For definiteness, we will take the image size equal to 128x128 pixels. The image has 3 color channels. The network structure for the first autoencoder is described by the following function:

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

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

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

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

The function receives the size of the input image as a parameter. The parameter is required to indicate the number of network outputs. The network contains two layers: an encoder and a decoder. The encoder responds to the first layer of the target network. The decoder serves to restore the original dimension of the

data from the compressed representation obtained as a result of encoding. It can be seen that the identifier of the first layer coincides with the identifier of the first layer in the description of the target network. This is used later during the transfer learning. The sum of squares of deviations is specified as the cost function in training, since the linear regression problem for each pixel of the input image is solved.

The creation of the model corresponding to the first autoencoder is performed by the following call:

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

Define the network corresponding to the second autoencoder. At this stage, the parameters of the second layer of the target network are pre-trained. The input and output data are the output of the first encoder. To obtain the target network, you must perform the transfer learning for the layers trained at different stages. This can be done in several ways.

- At each stage, a new model of the autoencoder is trained, independent of the previous autoencoder model. The training is performed on the output of the encoder of the previous autoencoder. Models trained at each stage are preserved. To construct the target model, the layer parameters are transferred from all the autoencoder models on the stack.
- At each stage, a new model of the autoencoder is trained, complementing the model of the previous autoencoder. To learn a new model, the parameters of the encoding layers of the previous autoencoders are transferred and the parameters of the layers trained in the previous stages are fixed. The input data at each stage are the original images, the output data are the outputs of the encoder of the network obtained from the previous autoencoder. The target model with pre-trained parameters is constructed by transferring weights from the model of the last autoencoder.

In this paper, the second implementation is considered. Thus, the second autoencoder model includes the encoder of the first autoencoder (first layer of the model) and the second layer of the target model. The decoder consists of one layer, the number of outputs coincides with the number of first layer outputs of the target network. The function of creating the second autoencoder model is given below.

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

        Affine(nout=128, init=Xavier(), bias=Constant(0),
            activation=Tanh(), name='fc_1'),
        Affine(nout=64, init=Xavier(), bias=Constant(0),
            activation=Tanh(), name='fc_2'),
        Affine(nout=128, init=Xavier(), bias=Constant(0),
            activation=Tanh(), name='fc_-2')
    ]
    model = Model(layers=layers)
    cost = GeneralizedCost(costfunc=SumSquared())
    return (model, cost)
```

The structure of the model is similar to the structure of the autoencoder model from the previous stage. The **`input_shape`** function parameter was introduced for the uniformity of function interfaces that create stack of models, and is not used here.

For each of the autoencoders, you need to create an additional model consisting only of the encoder. This is required to generate the output data used to train the next network in the stack of autoencoders. The function of constructing such network for the first autoencoder is:

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

        Affine(nout=128, init=Gaussian(scale=0.1), bias=Constant(0),
```

```
            activation=Tanh(), name='fc_1'),
    ]
    model = Model(layers)
    cost = GeneralizedCost(costfunc=SumSquared())
    return (model, cost)
```

The function of creating such network for the second autoencoder is similar to the corresponding function describing the structure of the target network without the last classification layer.

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

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

All the above functions of creating models are placed in a file **`ae_models.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. Repeat steps 3 and 4 for all models in the stack of autoencoders and the target model.
6. Testing the target model. It implies the feed forward of the neural network for the test dataset and the calculating the quality of the constructed model.
7. Saving the target model output. Saves the network output values for the test dataset.

To train and test the target model, a script developed in the previous practice is used. 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`** [5]:

```
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 practe. After these steps are completed, the files for the train and test datasets of the IMDB-WIKI dataset must be generated. We will 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`** [6] is used. It allows you to load data from external memory with batches, the size of which is specified when initializing neon. When training the autoencoders the iterator type **`HDF5IteratorAutoencode`**r is used [7]. They return input values as output values. To load the data, you will need the following:

```
from neon.data import HDF5IteratorAutoencoder

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

These objects provide access to the elements of the dataset.

3.2.4 Creating model

Using the previously created file with model descriptions, we will define a list of model generation functions for each of the autoencoders. In addition, we define the list of functions that create encoder models at each stage.

```
import models

step_model_generators = [
    models.generate_mlp_ae_stacked_step1_model, # the first autoencoder
    models.generate_mlp_ae_stacked_step2_model  # the second autoencoder
]
step_encoder_generators = [
    models.generate_mlp_ae_stacked_step1_encoder_model,
    models.generate_mlp_ae_stacked_step2_encoder_model
]
```

The model is created by calling the corresponding function from the list.

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

3.2.5 Training model

Let us consider the training of the first autoencoder of the stack. Optimization of the model parameters can be done with the help of the stochastic gradient descend (SGD) algorithm.

```
from neon.optimizers import GradientDescentMomentum

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

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

```
from neon.callbacks.callbacks import Callbacks

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

Further, you need to generate the data for the next model in the stack and perform its training. To generate the data, we will create an encoder network and transfer the weights from the model, which was trained at the current stage. The transfer leaning is implemented using the function **`import_matching_layers`**, demonstrated in the previous practice.

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

The function of creating and saving a dataset for the next training step can be implemented as shown below.

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

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

    def make_subset(subset_name, subset):
        outputs = encoder.get_outputs(subset)
        subset_path = save_path_base + "_" + subset_name + ".h5"
```

```
        generate_dataset(subset_name, subset, outputs, subset_path)
        return HDF5Iterator(subset_path), outputs[0].shape
    next_train_set, lshape = make_subset('train', train_set)
    next_val_set, _ = make_subset('val', val_set)

    return next_train_set, next_val_set, lshape
```

The function of creating a set in HDF5 format is:

```
import h5py as h5
from contextlib import closing

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

In these functions, the outputs of the encoder network are obtained from the current stage and stored in HDF5 files. The description of the data storage format in the file and methods of obtaining data is represented in the preliminary practice. Functions return a pair of iterators to the generated data sets (train and test) and the input data shape of the new set. Creating a new dataset is implemented as follows:

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

After creating the data, you can train the next network of the stack. You need to construct the model and transfer the trained weights from the previous model. For the transferred layers, the learning rate is set to zero in the represented implementation. A detailed description of these actions is given in the previous practice. The source code needed to train the new model is shown below.

```
# Create new model
step = 1
prev_model = step_model
step_model, cost = step_model_generators[step](input_shape)
imported_layers = import_matching_layers(
    prev_model.get_description(get_weights=True), step_model)

# Specify the optimization algorithm
default_optimizer = GradientDescentMomentum(
    0.1, momentum_coef=0.9, wdecay=0.0005)
optimizers_mapping = {'default': default_optimizer}

# Set learning rate for the transferred layers
imported_layers_optimizer_params = \
```

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

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

This sequence can be used to train all subsequent models in the stack, if any. You need to extend the list of functions that generate the models.

Steps to train the autoencoder models can be represented by a loop of the following form:

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

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

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

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

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

    prev_model = step_model
```

3.2.6 Testing model

To save the model in a file with the specified name, use the function **`save_params`** [8]. The code for saving the model in the stage of the autoencoder training:

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

3.3 Execution of training and testing model

3.3.1 Execution without specifying the parameters

By this moment, it is assumed that scripts have been developed for transfer learning (file **main_transfer.py**) and training the models of the stack of autoencoders (file **main_train_stack.py**). Running the script for training the stack of autoencoders is done from the command line by calling the commands listed below. Initially, the neon virtual environment is initialized, then the script is launched.

```
. .venv/bin/activate
python main_train_stack.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** [9] 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_train_stack.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.

3.3.4 Training and testing the target model

It is assumed that the training of the stack of autoencoders has been performed. The last step is to train the target model on the given task. For definiteness, we will assume that the results of training the stack, i.e. the model learned at the last step is stored in the file **mlp_stack_final.prm**. It is necessary to transfer

the trained layers to the target model and train it. We use a script that implements the transfer learning, developed in the previous practice. You must specify a new target model. The launch is performed as follows:

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

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

`Practice4_ae/main_train_stacked_autoencoder.py` и **`models/ae_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. IMDB-WIKI dataset [https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki].
5. Intel® neon™ Framework: backend initialization [http://neon.nervanasys.com/docs/latest/generated/neon.backends.gen_backend.html#neon.backends.gen_backend].
6. Intel® neon™ Framework: HDF5Iterator [http://neon.nervanasys.com/docs/latest/generated/neon.data.hdf5iterator.HDF5Iterator.html#neon.data.hdf5iterator.HDF5Iterator].
7. Intel® neon™ Framework: HDF5IteratorAutoencoder [http://neon.nervanasys.com/docs/latest/generated/neon.data.hdf5iterator.HDF5IteratorAutoencoder.html#neon.data.hdf5iterator.HDF5IteratorAutoencoder].
8. Intel® neon™ Framework: save_params [http://neon.nervanasys.com/docs/latest/generated/neon.models.model.Model.html#neon.models.model.Model.save_params].
9. Intel® neon™ Framework: NeonArgparser [http://neon.nervanasys.com/docs/latest/generated/neon.util.argparser.NeonArgparser.html#neon.util.argparser.NeonArgparser].