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 №1
The development of fully-connected neural networks 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 general steps of solving the machine learning tasks using the Intel® neon™ Framework. Demonstration of these steps is conducted on the example of solving the problem of classifying a person's sex from a photo using multilayered fully-connected neural networks. IMDB-WIKI is used as the dataset [6].

2 Guidelines

2.1 Goals and tasks

The goal of this practice is to study the workflow of deep models in the Intel® neon™ Framework based on the example of the fully-connected neural networks constructed for solving the problem of classifying a person's sex by the photo.

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

1. Study the general scheme of solving the machine learning tasks.
2. Study the workflow of deep models in the Intel® neon™ Framework.
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 a model of a fully-connected neural network and describe it using the Intel® neon™ Framework.
5. Train the model and evaluate the quality of the classification.

2.2 Practice structure

The practice describes a workflow of deep models in the Intel® neon™ Framework. First, the test and train data are loaded. Further, an example of a multilayered fully-connected network is represented. A script is developed that provides training and testing of the network, as well as saving the results of solving the problem. The work is carried out on the example of the problem of classifying a person's sex by the photo. IMDB-WIKI [6] is used as the dataset.

2.3 Recommended study sequence

The recommended study sequence is as follows:

1. Study the general scheme for solving the machine learning tasks.
2. Study the workflow of deep neural networks in the neon framework. You can use the lecture materials and documentation of the neon framework.
3. Develop a model of a multilayered fully-connected network.
4. Load train and test data.
5. Develop a script for training and testing the constructed model.
6. Save the classification results.

3 General scheme of solving machine learning tasks

The solution of the machine learning task includes the following stages:

1. Data preprocessing. It assumes the preliminary processing of data and conversion to the format required for subsequent use.
2. Model construction. Assumes the choice of the machine learning method and its parameters.
3. Training the model. Implies tuning the parameters of the machine learning model.
4. Testing the model. It assumes the use of the constructed model to solve the task on the test data.
5. Preservation of the results and their analysis.

Further, the implementation of the listed stages using the Intel® neon™ Framework based on the example of the problem of classifying a person's sex by a photo on IMDB-WIKI data [6] is represented.

4 Manual

4.1 Multilayered fully-connected neural network

The model construction includes a description of the neural network structure and the assignment of the cost function used during training. 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 [7].

To solve the classification problem with two categories, we will create a description of a simple model containing one hidden fully-connected layer (the Rosenblatt's perceptron [4]). Constructing a fully-connected layer can be performed as follows:

```
from neon.layers import Affine
from neon.initializers import Gaussian, Constant
from neon.transforms import Logistic

layer = Affine(nout=2,
    init=Gaussian(0.1),
    bias=Constant(1),
    activation=Logistic(shortcut=True)
)
```

In this code, a fully-connected layer with two neurons is created. To initialize the weights (connections of neurons with the previous layer), we used a Gaussian (normal) distribution with zero mean and the standard deviation of 0.1. When creating a fully-connected layer with **`Affine`** class, the **`bias`** parameter is available, indicating the need to create additional parameters – offsets. Their initialization is performed by a constant value equal to one. The logistic activation function is specified. Complete information about the parameters of this layer can be found in the documentation [8].

Further, you need to create a list of network layers and a model that is a container of layers.

```
from neon.models import Model

layers = [ layer ]
model = Model(layers)
```

After that, we obtain a model that consists of one fully-connected layer and has 2 outputs.

Find an example of a more complicated model (Rumelhart's multilayered perceptron) below.

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

    Affine(nout=128, init=Xavier(), bias=Uniform(0.3, 0.7),
        activation=Tanh()),
    Affine(nout=64, init=Kaiming(), bias=Constant(0)),
    Affine(nout=2, init=Gaussian(scale=0.1),
        activation=Logistic(shortcut=True))
]
model = Model(layers)
```

In this model, a layer of non-parametric input data conversion is specified – division by a constant of 128. Such transform allows simulating the division by the standard deviation in the case of a sufficiently large and diverse set of images. Three fully-connected layers with different initialization parameters and the number of neurons are created. Such network performs the serial transforms specified by the layers in the network description.

Specify the cost function, which is used during optimization of network parameters. In classification problems, as a rule, the cost function is the cross-entropy. 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 **`mlp_models.py`**. For each model, we create a separate function of the following form:

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

        Affine(nout=128, init=Xavier(), bias=Uniform(0.3, 0.7),
            activation=Tanh()),
        Affine(nout=64, init=Kaiming(), bias=Constant(0)),
        Affine(nout=2, init=Gaussian(scale=0.1),
            activation=Logistic(shortcut=True))
    ]
    model = Model(layers)
    cost = GeneralizedCost(costfunc=CrossEntropyBinary())
    return model, cost
```

The model can be constructed using the following code:

```
import mpl_models

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

4.2 Training and testing the network

4.2.1 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 previous 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.

4.2.2 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`** [9]:

```
from neon.backends import gen_backend

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

4.2.3 Loading data

The steps for preparing the data are described in detail in the previous 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`** [10] 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’)
```

4.2.4 Creating model

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

```
import mlp_models

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

4.2.5 Training model

To train the model, you need to set an optimization algorithm and its parameters. neon provides several optimization algorithms [11]. In deep learning, stochastic gradient descent (Stochastic Gradient Descend, SGD) is widely used. The following code allows to define optimizer object that implements a Nesterov's Accelerated Gradient (NAG) algorithm [5].

```
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 [12].

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.

4.2.6 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.

4.2.7 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 class
```

4.3 Execution of training and testing model

4.3.1 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
```

4.3.2 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`** [13] 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')
```

4.3.3 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.

Full sources of the example can be found in the materials of this course:
`Practice1_mlp/main_classify.py` and **`models/mlp_models.py`**.

5 Literature

5.1 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. Rosenblatt F. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. – CORNELL AERONAUTICAL LAB INC BUFFALO NY, 1961. – №. VG-1196-G-8.
5. Nesterov Y. et al. Gradient methods for minimizing composite objective function. – 2007.

5.2 References

6. IMDB-WIKI dataset [https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki].
7. Intel® neon™ Framework: layers [http://neon.nervanasys.com/docs/latest/layers.html].
8. Intel® neon™ Framework: Affine layer [http://neon.nervanasys.com/docs/latest/generated/neon.layers.layer.Affine.html#neon.layers.layer.Affine].
9. Intel® neon™ Framework: backend initialization [http://neon.nervanasys.com/docs/latest/generated/neon.backends.gen_backend.html#neon.backends.gen_backend].
10. Intel® neon™ Framework: HDF5Iterator [http://neon.nervanasys.com/docs/latest/generated/neon.data.hdf5iterator.HDF5Iterator.html#neon.data.hdf5iterator.HDF5Iterator].
11. Intel® neon™ Framework: optimizers [http://neon.nervanasys.com/docs/latest/optimizers.html].
12. Intel® neon™ Framework: GradientDescentMomentum [http://neon.nervanasys.com/docs/latest/generated/neon.optimizers.optimizer.GradientDescentMomentum.html#neon.optimizers.optimizer.GradientDescentMomentum].

13. Intel® neon™ Framework: NeonArgparser [http://neon.nervanasys.com/docs/latest/generated/neon.util.argparser.NeonArgparser.html#neon.util.argparser.NeonArgparser].
14. Intel® neon™ Framework: FCNN sample [http://neon.nervanasys.com/docs/latest/mnist.html].