



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»**

**Introduction
to the Intel® neon™ Framework**

Supported by Intel

Valentina Kustikova,
Phd, lecturer, department of Computer software
and supercomputer technologies

Content

- Overview of the Intel® neon™ Framework
- Installation of the Intel® neon™ Framework
- The typical workflow for a deep learning model
 - Generate a backend
 - Load data
 - Specify model architecture
 - Train model
 - Evaluate model
- Example of a multilayer fully-connected network for solving the problem of predicting a person's sex from a photo



OVERVIEW OF THE INTEL® NEON™ FRAMEWORK



Overview of the Intel® neon™ Framework (1)

- Intel® neon™ Framework (neon) is a deep learning tool developed by Intel Corporation
- The purpose of the development is the easy use of the deep learning library and its extensibility
- Neon provides maximum performance on all hardware (demonstrates better performance than Caffe, Theano, Torch and TensorFlow)
- Neon is used by the Intel Nervana team to solve internal tasks from different subject areas



Overview of the Intel® neon™ Framework (2)

- Apache 2.0 License

- Source codes
[<https://github.com/NervanaSystems/neon>]

- Documentation (API + tutorials)
[<http://neon.nervanasys.com/docs/latest>]

- Intel AI Academy Page
[<https://software.intel.com/ru-ru/ai-academy/frameworks/neon>]

- A set of trained models
[<https://github.com/NervanaSystems/ModelZoo>]



Overview of the Intel® neon™ Framework (3)

- Supported OS: Linux, Mac OS X
- Programming language: Python
- Supported platforms:
 - CPU (+ Intel® Math Kernel Library)
 - GPU (Pascal, Maxwell, Kepler)
 - Nervana hardware
- Multi-GPU support



Free cloud compute Intel® AI DevCloud

- Intel® AI DevCloud
[<https://software.intel.com/en-us/ai-academy/tools/devcloud>]
- Intel® AI DevCloud powered by Intel® Xeon® Scalable processors for machine learning and deep learning training and inference compute needs
- Available frameworks and tools:
 - Intel® neon™ Framework
 - Intel® Optimization for Theano*
 - Intel® Optimization for TensorFlow*
 - Intel® Optimization for Caffe*
 - Intel® Distribution for Python* (including NumPy, SciPy, and scikit-learn*)
 - Keras* library



INSTALLATION OF THE INTEL® NEON™ FRAMEWORK (LINUX)



Installation (1)

- Python 2.7 or Python 3.4+
- Required dependencies:
 - ***python-pip*** is a tool to install Python packages
 - ***libhdf5-dev*** is a library to load data in HDF5 format
 - ***libyaml-dev*** is a library to parse a deep model description and parameters for its training represented in YAML format
 - ***pkg-config*** retrieves information about installed libraries
 - ***python-virtualenv*** allows to create the isolated virtual environments in Python 2.7 (creates an environment that has its own installation directories and, if necessary, does not have access to the global libraries)



Installation (2)

- Optional dependencies:
 - OpenCV is required to activate data loader [aneon](#) (ffmpeg to process audio and video is required)
 - Intel® Math Kernel Library is used to support multithreading execution on CPUs
 - CUDA SDK and drivers to execute on GPUs
- **Note:** using [Intel® Distribution for Python*](#) (including MKL, NumPy, SciPy, scikit-learn*) instead of Python+MKL allows to accelerate training and testing deep models



Installation (3)

```
# clone repository
git clone https://github.com/NervanaSystems/neon.git
# change directory to neon
cd neon
# go to the latest stable release
git checkout latest
# build the framework (by default with Intel® MKL)
make
```

- **Note:** during the installation, a virtual environment is initialized, within which all the necessary dependencies are installed



Execute a sample from the Intel® neon™ Framework (1)

```
cd neon/  
# activate virtual environment  
. .venv/bin/activate
```

```
[kustikova_v@master neon]$ . .venv/bin/activate  
(.venv2) [kustikova_v@master neon]$ █
```

```
# execute on CPU with Intel® MKL support (without GPU)  
python examples/mnist_mlp.py -b mkl
```

```
(.venv2) [kustikova_v@master neon]$ python examples/mnist_mlp.py -b mkl  
Epoch 0 [Train |          | 469/469 batches, 0.74 cost, 2.03s]  
Epoch 1 [Train |          | 469/469 batches, 1.35 cost, 2.00s]  
Epoch 2 [Train |          | 469/469 batches, 0.64 cost, 2.09s]  
Epoch 3 [Train |          | 468/468 batches, 0.14 cost, 2.06s]  
Epoch 4 [Train |          | 468/468 batches, 0.12 cost, 2.02s]  
Epoch 5 [Train |          | 468/468 batches, 0.11 cost, 2.04s]  
Epoch 6 [Train |          | 468/468 batches, 0.10 cost, 1.98s]  
Epoch 7 [Train |          | 468/468 batches, 0.10 cost, 2.07s]  
Epoch 8 [Train |          | 468/468 batches, 0.09 cost, 1.94s]  
Epoch 9 [Train |          | 468/468 batches, 0.08 cost, 2.03s]  
2018-03-12 10:16:40,221 - neon - DISPLAY - Misclassification error = 2.6%
```



Execute a sample from the Intel® neon™ Framework (2)

```
# execute on CPU (without Intel® MKL)  
python examples/mnist_mlp.py -b cpu
```

```
(.venv2) [kustikova_v@master neon]$ python examples/mnist_mlp.py -b cpu  
Epoch 0  [Train | 469/469 batches, 0.24 cost, 2.99s]  
Epoch 1  [Train | 469/469 batches, 0.21 cost, 2.18s]  
Epoch 2  [Train | 469/469 batches, 0.17 cost, 2.57s]  
Epoch 3  [Train | 468/468 batches, 0.14 cost, 2.44s]  
Epoch 4  [Train | 468/468 batches, 0.12 cost, 2.29s]  
Epoch 5  [Train | 468/468 batches, 0.10 cost, 2.31s]  
Epoch 6  [Train | 468/468 batches, 0.09 cost, 2.30s]  
Epoch 7  [Train | 468/468 batches, 0.08 cost, 2.27s]  
Epoch 8  [Train | 468/468 batches, 0.08 cost, 2.27s]  
Epoch 9  [Train | 468/468 batches, 0.07 cost, 2.30s]  
2018-03-12 10:09:33,411 - neon - DISPLAY - Misclassification error = 2.5%
```

```
# execute on GPU (GPU is required during installation)  
python examples/mnist_mlp.py -b gpu
```



Execute a sample from the Intel® neon™ Framework (3)

```
# execute a sample described in YAML format  
neon examples/mnist_mlp.yaml  
  
# deactivate a virtual environment  
deactivate
```



THE TYPICAL WORKFLOW FOR A DEEP LEARNING MODEL



The typical workflow for a deep learning model (1)

The typical workflow for a deep learning model:

1. Generate a backend

- The backend defines where computations are executed: CPU, CPU+MKL, GPU (Pascal, Maxwell or Kepler)

2. Load data

- Load well-known datasets (MNIST, CIFAR10, PASCAL VOC, UCF101 etc.)
- Implement data loaders

3. Specify model architecture

- Layers
- Activation functions
- Initializers



The typical workflow for a deep learning model (2)

4. *Train model*

- Cost function
- Metric
- Optimization method
- Learning schedule

5. *Evaluate model*

- Model
- Metric

- **Note:** steps #2-5 represent a classical scheme for solving the task using deep learning methods



The typical workflow for a deep learning model

The typical workflow for a deep learning model:

1. ***Generate a backend***
 - The backend defines where computations are executed: CPU, CPU+MKL, GPU (Pascal, Maxwell or Kepler)
2. ***Load data***
3. ***Specify model architecture***
4. ***Train model***
5. ***Evaluate model***



Generate a backend (1)

- Automatic backend generation assumes the use of the internal class `neon.util.argparser.NeonArgparser` to parse command line arguments
- Main supported options:
 - `[-b {cpu, mkl, gpu}]` is a backend
 - `[-e EPOCHS]` is a number of epochs
 - `[-z BATCH_SIZE]` is mini-batch size

```
from neon.util.argparser import NeonArgparser

# parse the command line arguments
parser = NeonArgparser(__doc__)
args = parser.parse_args()
```



Generate a backend (2)

- Direct access to the backend is important when creating custom layers and cost functions
- This issue is beyond the scope of this lecture
- Link to the documentation
[\[http://neon.nervanasys.com/docs/latest/backends.html\]](http://neon.nervanasys.com/docs/latest/backends.html)



The typical workflow for a deep learning model

The typical workflow for a deep learning model:

1. ***Generate a backend***
2. ***Load data***
 - Load well-known datasets (MNIST, CIFAR10, PASCAL VOC, UCF101 etc.)
 - Implementation of data loaders
3. ***Specify model architecture***
4. ***Train model***
5. ***Evaluate model***



Load well-known datasets

- ❑ `neon.data.datasets.Dataset` is a base class to load datasets
- ❑ The Intel® neon™ Framework contains abstractions presenting datasets to process this data
- ❑ Sample of loading MNIST dataset:

```
from neon.data import MNIST

mnist = MNIST(path='path/to/save/downloadeddata/ ')
train_set = mnist.train_iter
valid_set = mnist.valid_iter
```



Implementation of data loaders (1)

- Two components for working with data in Intel® neon™ Framework:
 - **neon.data.dataiterator.NervanaDataIterator** is a data iterator, that feeds the model with mini-batches of data during training or evaluation
 - **neon.data.datasets.Dataset** is a dataset class, which handles the loading and preprocessing of the data. When working with your own data, the latter is optional although highly recommended



Implementation of data loaders (2)

- Data iterators are python iterables. They implement the `__iter__` method, which returns a new mini-batch of data with each call
- If your data is small enough to fit into memory:
 - `neon.data.dataiterator.ArrayIterator` is an iterator for image data or other data in the form of numpy-arrays

```
from neon.data import ArrayIterator
import numpy as np

x = np.random.rand(10000,3072) # x.shape = (10000, 3072)
y = np.random.randint(0,10,10000) # y.shape = (10000, )

train = ArrayIterator(X=x, y=y, nclass=10,
                      lshape=(3,32,32))
```



Implementation of data loaders (3)

- Data iterators are python iterables. They implement the `__iter__` method, which returns a new mini-batch of data with each call
- If your data is small enough to fit into memory:
 - `neon.data.dataiterator.ArrayIterator` is an iterator for image data or other data in the form of numpy-arrays
 - Specialized iterators (`neon.data.Text` и `neon.data.ImageCaption` for text sets and image captioning)
- If your data is too large:
 - `neon.data.hdf5iterator.HDF5Iterator` is an iterator for data in HDF5 format
 - [Aeon](#) is an external data loader



The typical workflow for a deep learning model

The typical workflow for a deep learning model:

1. ***Generate a backend***
2. ***Load data***
3. ***Specify model architecture***
 - Layers
 - Activation functions
 - Initializers
4. ***Train model***
5. ***Evaluate model***



Deep model architecture (1)

- The deep model architecture is formed by adding layers to the list
- Each layer has several key methods:

| Method | Description |
|---|--|
| <code>configure(self, in_obj)</code> | Defining input/output shape of a layer (<code>out_shape/in_shape</code>) |
| <code>allocate(self, shared_outputs=None)</code> | Allocate the output buffer |
| <code>fprop(self, inputs, inference=False)</code> | Forward propagation |
| <code>bprop(self, error)</code> | Backward propagation |



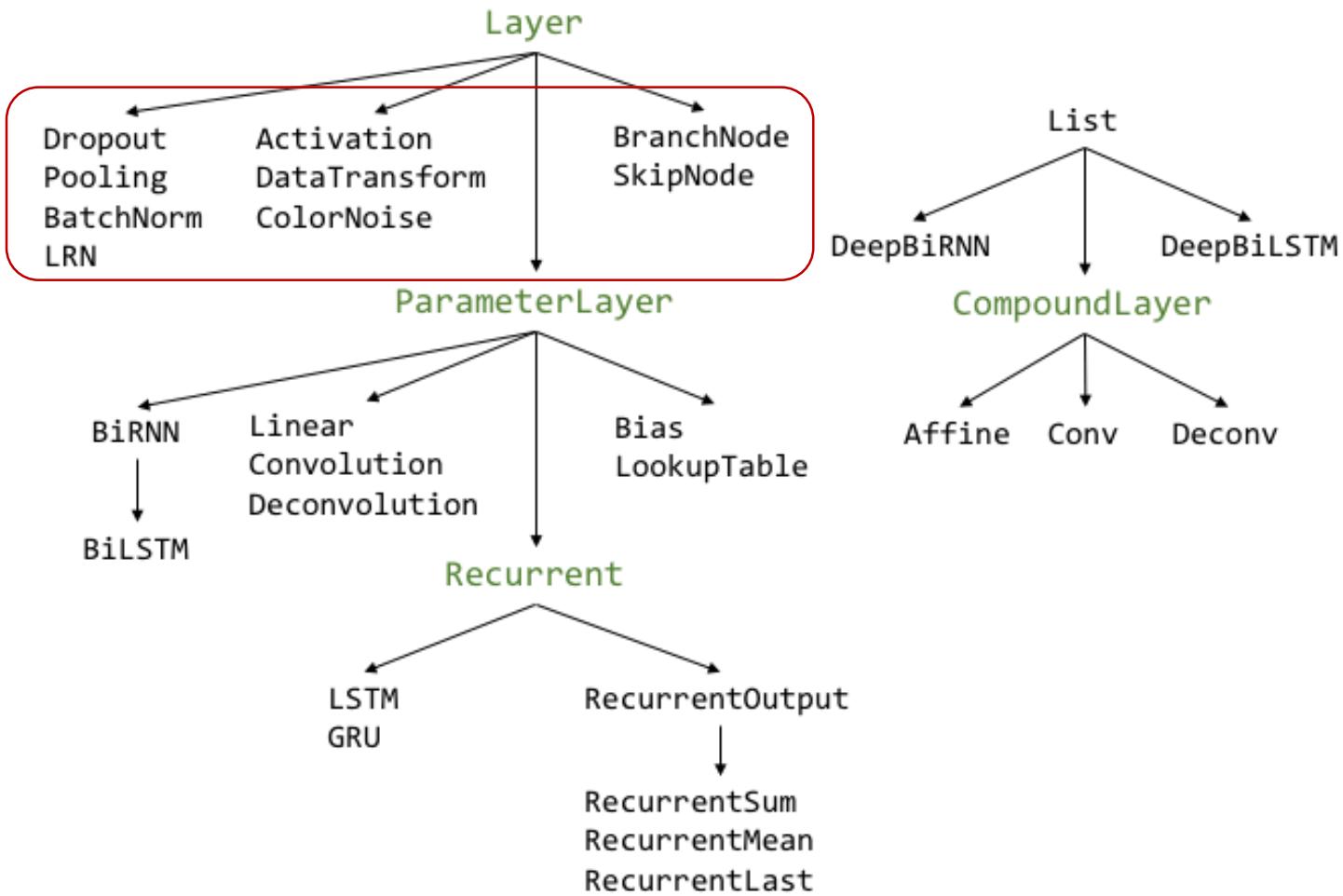
Deep model architecture (2)

- Calling methods during training a network:
 - Forward propagation of training data and calling method **configure** to set shapes of output feature maps
 - Calling method **allocate** at each layer to allocate memory for feature maps



Layer taxonomy (1)

Layers that do not have weighting parameters (do not require initialization)

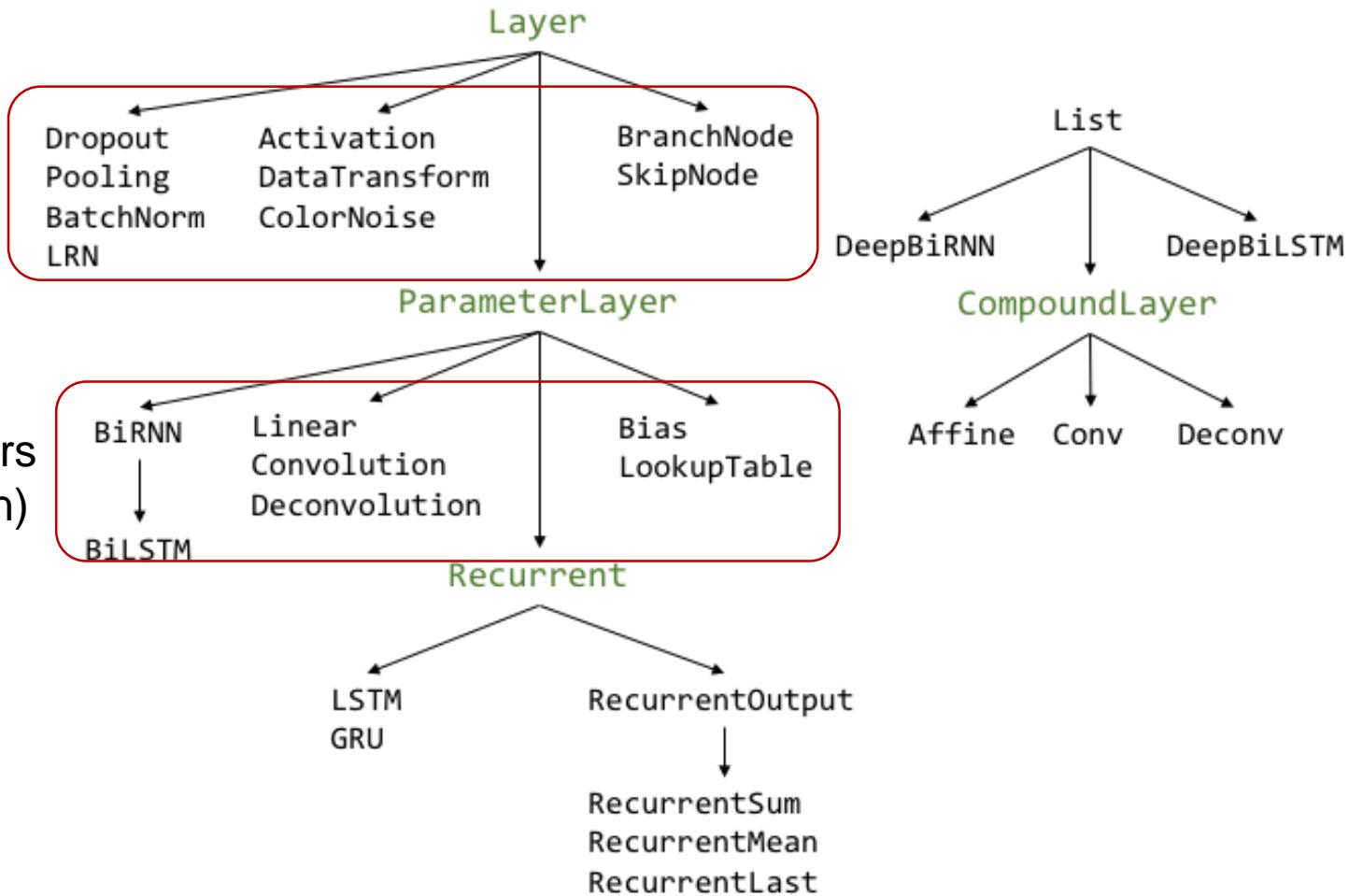


* Intel® neon™ Framework documentation
[<http://neon.nervanasys.com/docs/latest/layers.html>].



Layer taxonomy (2)

Layers that do not have weighting parameters (do not require initialization)

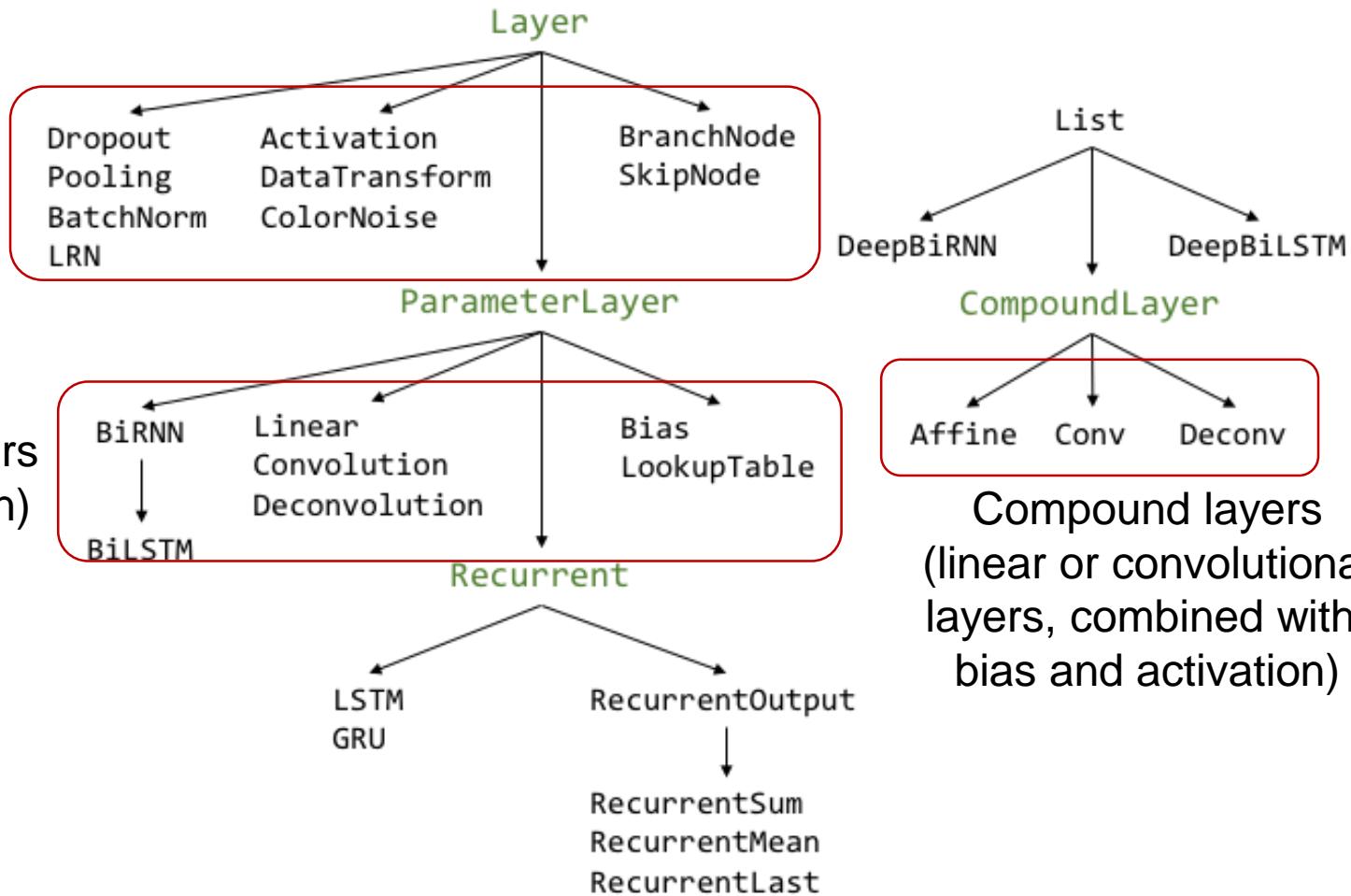


Layers that have weighting parameters (require initialization)

* Intel® neon™ Framework documentation
[<http://neon.nervanasys.com/docs/latest/layers.html>].

Layer taxonomy (3)

Layers that do not have weighting parameters (do not require initialization)



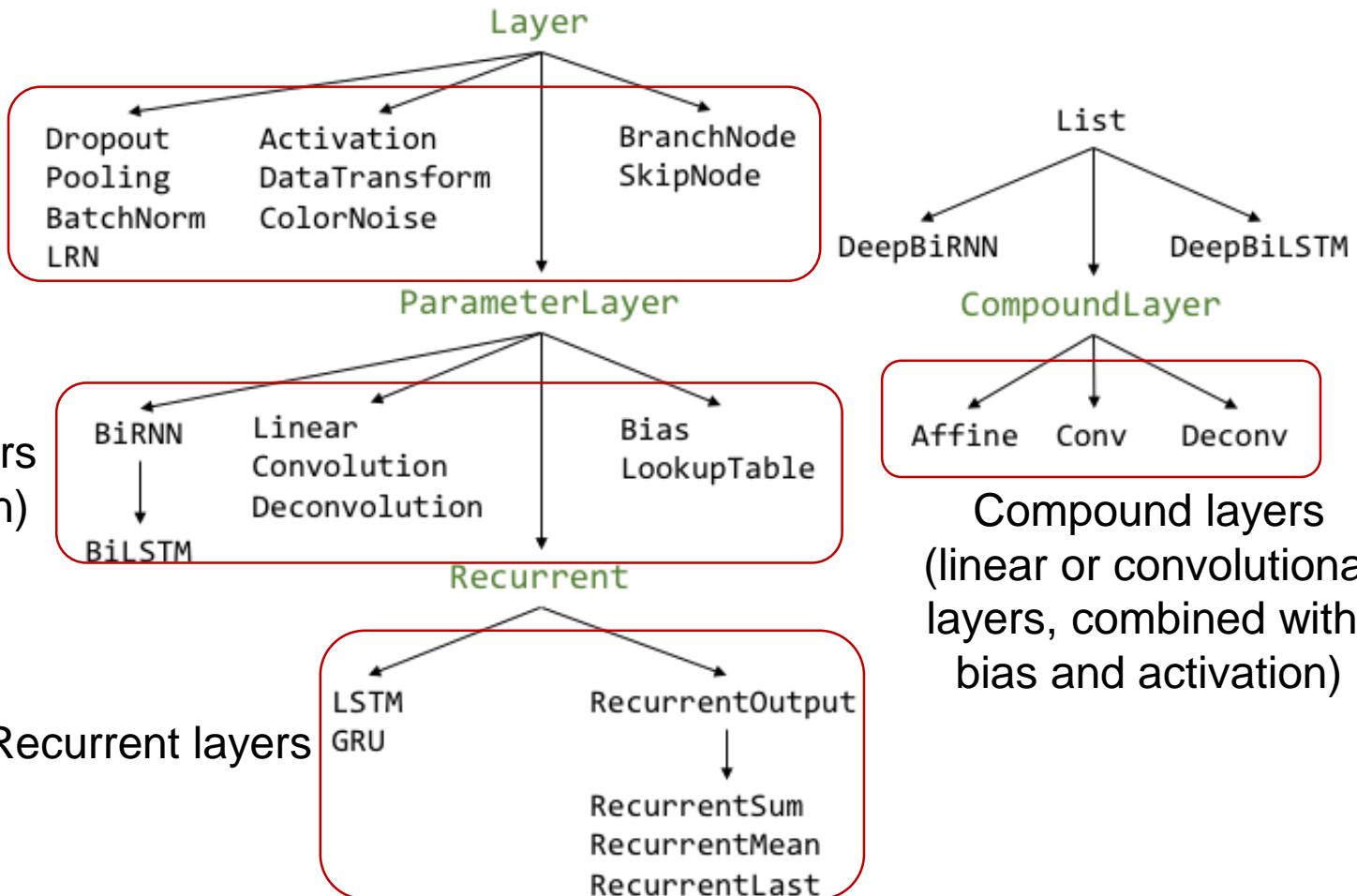
Layers that have weighting parameters (require initialization)

Compound layers (linear or convolutional layers, combined with bias and activation)

* Intel® neon™ Framework documentation
[<http://neon.nervanasys.com/docs/latest/layers.html>].

Layer taxonomy (4)

Layers that do not have weighting parameters (do not require initialization)



* Intel® neon™ Framework documentation
[<http://neon.nervanasys.com/docs/latest/layers.html>].



Typical layers (1)

| | | |
|--------------------------------------|--|---|
| <code>neon.layers.Linear</code> | <code>nout</code> | Fully connected layer with <code>nout</code> units |
| <code>neon.layers.Convolution</code> | <code>fshape</code> , <code>strides</code> , <code>padding</code> | Convolutional layer with filter parameters <code>fshape=(height, width, num_filters)</code> |
| <code>neon.layers.Pooling</code> | <code>fshape</code> , <code>op</code> , <code>strides</code> , <code>padding</code> | Pooling with shape parameters <code>fshape=(height, width, num_filters)</code> and operation <code>op ("avg"/"sum")</code> |
| <code>neon.layers.Activation</code> | <code>transform</code> | Appling <code>transform</code> to the input |



Typical layers (2)

| | | |
|---------------------------------------|---|---|
| <code>neon.layers.layer.Affine</code> | <code>nout, init, bias, activation, name</code> | Linear layer with number of units nout , bias and activation function activation |
| <code>neon.layers.Conv</code> | <code>fshape, init, strides, padding, dilation, bias, batch_norm, activation, name</code> | Convolutional layer with bias and activation function activation |



Typical activation functions

| | |
|---------------------------------|---|
| neon.transforms.Identity | $f(x) = x$ |
| neon.transforms.Rectlin | $f(x) = \max\{x, 0\}$ |
| neon.transforms.Softmax | $f(x_j) = \frac{e^{x_j}}{\sum e^{x_i}}$ |
| neon.transforms.Tanh | $f(x) = th(x)$ |
| neon.transforms.Logistic | $f(x) = \frac{1}{1 + e^{-x}}$ |



Typical initializers

| | |
|---|---|
| <code>neon.initializers.Constant</code> | Initialize all tensors with a constant value <code>val</code> |
| <code>neon.initializers.Uniform</code> | Uniform distribution from <code>low</code> to <code>high</code> |
| <code>neon.initializers.Gaussian</code> | Gaussian distribution with mean $\mu=\text{loc}$ and standard deviation $\sigma=\text{scale}$ |



Sample of deep model (1)

```
from neon.initializers import Gaussian
from neon.layers import Conv, Pooling, Affine
from neon.transforms import Rectlin, Softmax
...
layers = [Conv((11, 11, 64), init=Gaussian(scale=0.01),
               activation=Rectlin(), padding=3,
               strides=4),
          Pooling(3, strides=2),
          Conv((5, 5, 192), init=Gaussian(scale=0.01),
                activation=Rectlin(), padding=2),
          Pooling(3, strides=2),
          Conv((3, 3, 384), init=Gaussian(scale=0.03),
                activation=Rectlin(), padding=1),
```



Sample of deep model (2)

```
    Conv( (3, 3, 256) , init=Gaussian(scale=0.03) ,  
          activation=Rectlin() , padding=1) ,  
    Conv( (3, 3, 256) , init=Gaussian(scale=0.03) ,  
          activation=Rectlin() , padding=1) ,  
    Pooling(3, strides=2) ,  
    Affine(nout=4096, init=Gaussian(scale=0.01) ,  
           activation=Rectlin()) ,  
    Affine(nout=4096, init=Gaussian(scale=0.01) ,  
           activation=Rectlin()) ,  
    Affine(nout=1000, init=Gaussian(scale=0.01) ,  
           activation=Softmax())  
]
```



The typical workflow for a deep learning model

The typical workflow for a deep learning model:

1. ***Generate a backend***
2. ***Load data***
3. ***Specify model architecture***
4. ***Train model***
 - Cost function is a minimized functional during the training of the deep models
 - Metric is a measurement of the problem decision quality
 - Optimization method
 - Learning schedule
5. ***Evaluate model***



Cost functions (1)

| | |
|---|---|
| <code>neon.transforms.CrossEntropyBinary</code> | Binary cross-entropy $-t \log y - (1 - t) \log(1 - y)$ |
| <code>neon.transforms.CrossEntropyMulti</code> | Multinomial cross-entropy $-\sum t_i \log y_i$ |
| <code>neon.transforms.SumSquared</code> | Square error $\sum (y_i - t_i)^2$ |
| <code>neon.transforms.MeanSquared</code> | Normalized square error $\frac{1}{N} \sum (y_i - t_i)^2$ |



Cost functions (2)

- Implementing cost functions:
 - Developing of the `neon.transforms.Cost` class heir
 - Implementing methods `__call__()` и `bprop()` to compute cost function value and its derivative



Metrics (1)

| | |
|--|--|
| neon.transforms.Misclassification | Misclassification error: $\frac{fp + fn}{tp + tn + fp + fn}$ |
| neon.transforms.Accuracy | Classification accuracy: $\frac{tp + tn}{tp + tn + fp + fn}$ |
| neon.transforms.PrecisionRecall | Precision/Recall: $precision = \frac{tp}{tp + fp}$ $recall = \frac{tp}{tp + fn}$ |

- ❑ fp = false positives, tp = true positives
- ❑ fn = false negatives, fp = false positives



Metrics (2)

- Developing metrics:
 - Developing of the `neon.transforms.cost.Metric` class heir
 - Implementing method `__call__()` to compare layout with the network output



Optimization methods

| | |
|--|--|
| <code>neon.optimizers.GradientDescentMomentum</code> | Stochastic gradient descent with momentum |
| <code>neon.optimizers.RMSProp</code> | Root Mean Square propagation |
| <code>neon.optimizers.Adagrad</code> | Adagrad method for adapting the learning rate |
| <code>neon.optimizers.Adadelta</code> | Adadelta method for adapting the learning rate |
| <code>neon.optimizers.Adam</code> | Adaptive Moment Estimation (Adam optimization algorithm) |
| <code>neon.optimizers.MultiOptimizer</code> | Class for assigning optimizers to different layers |



Learning schedules

- Learning rate schedule to adjust the learning rate over epochs:

| | |
|-------------------------------------|--|
| neon.optimizers.Schedule | Constant or step learning rate |
| neon.optimizers.ExpSchedule | Exponential decay $\alpha(t) = \frac{\alpha_0}{1+\beta t}$, α_0 is an initial learning rate, t is an epoch number |
| neon.optimizers.PolySchedule | Polynomial learning rate $\alpha(t) = \alpha_0 \times \left(1 - \frac{t}{T}\right)^\beta$, T is a number of epochs |



Sample of optimizer creation

```
from neon.optimizers import GradientDescentMomentum,  
                           MultiOptimizer, Schedule  
  
...  
# learning rate changes on the iterations #22, 44 and 65  
weight_sched = Schedule([22, 44, 65],  
                        (1 / 250.)**(1 / 3.))  
opt_gdm = GradientDescentMomentum(0.01, 0.0,  
                                   wdecay=0.0005, schedule=weight_sched)  
  
opt = MultiOptimizer({'default': opt_gdm})
```



The typical workflow for a deep learning model

The typical workflow for a deep learning model:

1. *Generate a backend*
2. *Load data*
3. *Specify model architecture*
4. *Train model*
5. *Evaluate model*
 - Model
 - Metric



Model as a container for combining data, a deep model and learning parameters (1)

- The class `neon.models.model.Model`
- Constructor parameters:
 - `layers` is a list of layers
- Some methods:
 - `fit(...)` is a method to train a deep model
 - `eval(...)` is a method to evaluate (test) a deep model
 - `get_outputs(...)` is a method to get outputs of the last neural network layer
 - `save_params(...)` is a method to save parameters of deep model
 - `load_params(...)` is a method to load trained parameters



Model as a container for combining data, a deep model and learning parameters (2)

```
fit(dataset, cost, optimizer,  
     num_epochs, callbacks)
```

- **dataset** is a training set
- **cost** is a cost function
- **optimizer** is an optimization algorithm and its parameters
- **num_epochs** is a number of epochs
- **callbacks** is a set of callbacks performed when the mini-batch processing is completed or the epoch is completed

```
eval(dataset, metric)
```

- **dataset** is a test set
- **metric** is a metric of quality assessment



Model as a container for combining data, a deep model and learning parameters (3)

```
get_outputs(dataset)
```

- ❑ **dataset** is a set to compute network outputs (forward propagation for the input dataset)

```
save_params(param_path, keep_states=True)
```

- ❑ **param_path** is a file name to save parameters of deep model
- ❑ **keep_states** is a flag to save training state

```
load_params(param_path, load_states =True)
```

- ❑ **param_path** is a file name to load parameters
- ❑ **load_states** is a flag to restore training state



Sample of training deep model (1)

```
from neon.models import Model  
from neon.transforms import Misclassification  
  
...  
  
# create a model container  
mlp = Model(layers=layers)  
  
  
# create a container of callbacks  
#   model=mlp is a model  
#   eval_set=valid_set is a validation dataset  
callbacks = Callbacks(mlp, eval_set=valid_set,  
                      **args.callback_args)  
  
...
```



Sample of training deep model (2)

```
# training mlp
mlp.fit(train_set, optimizer=optimizer,
        num_epochs=args.epochs, cost=cost,
        callbacks=callbacks)

# evaluate the model
error_rate = mlp.eval(valid_set,
                      metric=Misclassification())
```



AN EXAMPLE OF A MULTILAYER FULLY-CONNECTED NETWORK



The problem of predicting a person's sex from a photo

□ IMDB-WIKI dataset

[<https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki>]

IMDb



Wikipedia



460,723 images

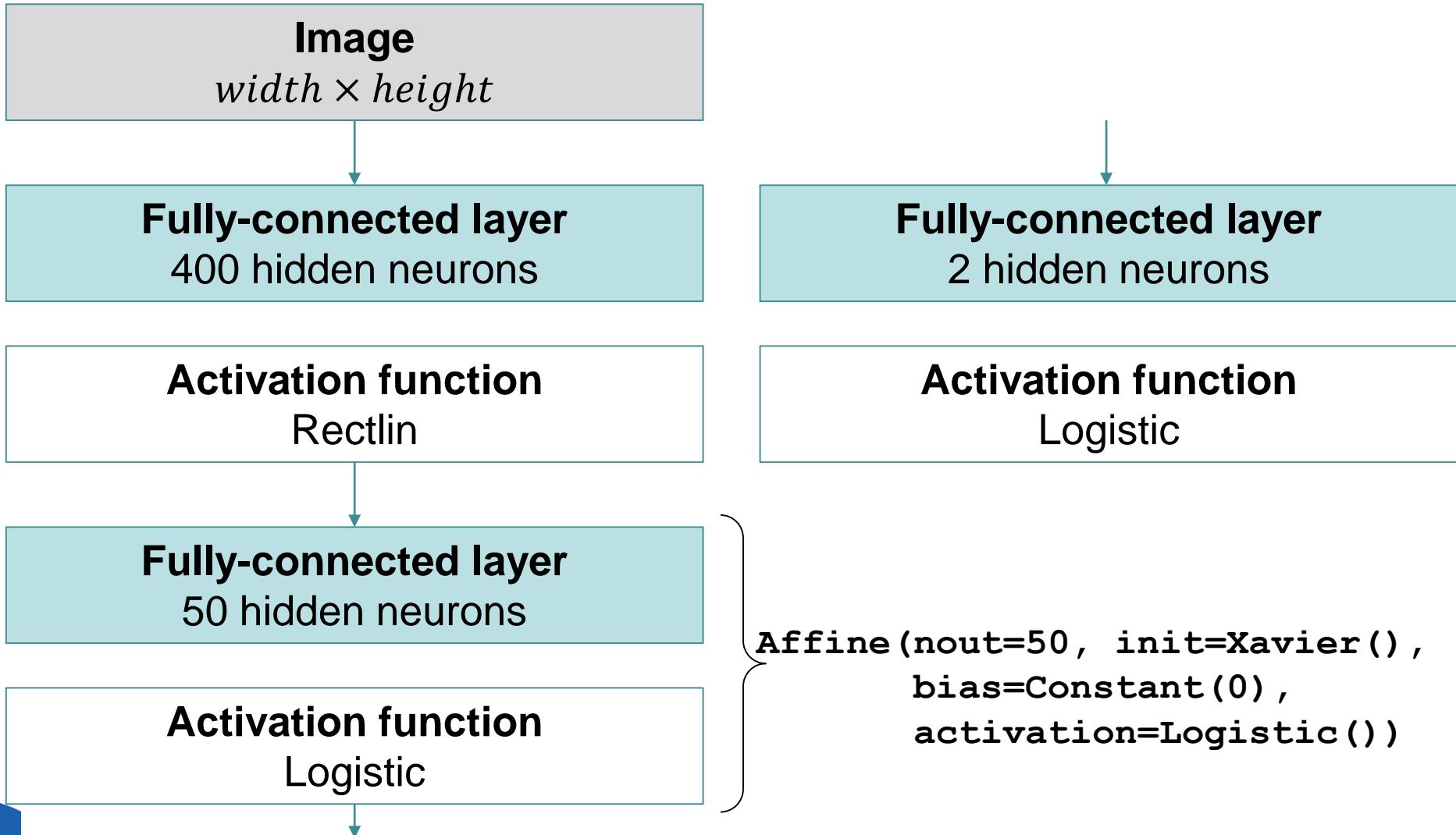
62,328 images

□ Layout:

- Bounding box for faces (is used for face detection)
- Person age (is used for classification problem)
- **Persons' sex (is used for classification problem)**
- ...



Fully-connected neural network



Using the Intel® neon™ Framework to train/test a multi-layer fully-connected network (1)

```
from neon.callbacks.callbacks import Callbacks, LossCallback
from neon.models import Model
from neon.optimizers import GradientDescentMomentum
from neon.util.argparser import NeonArgparser
from neon.data import HDF5Iterator, ArrayIterator

import numpy as np
import numpy.linalg

# load functions to read data
from datasets.imdb_wiki_face_dataset import IMDB_WIKI_FACE
...
```



Using the Intel® neon™ Framework to train/test a multi-layer fully-connected network (2)

```
def generate_mlp_model():
    layers = [
        DataTransform(transform=Normalizer(divisor=128.0)),
        Affine(nout=400, init=Xavier(), bias=Constant(0),
               activation=Rectlin()),
        Affine(nout=50, init=Xavier(), bias=Constant(0),
               activation=Logistic()),
        Affine(nout=2, init=Xavier(), bias=Constant(0),
               activation=Logistic(shortcut=True))
    ]
    model = Model(layers=layers)
    cost = GeneralizedCost(costfunc=CrossEntropyBinary())
    return (model, cost)

...
```



Using the Intel® neon™ Framework to train/test a multi-layer fully-connected network (3)

```
if (__name__ == '__main__'):  
    parser = NeonArgparser(__doc__)  
    args = parser.parse_args()  
  
    # IMDB_WIKI_FACE is a class to load data in HDF5 format  
    dataset = IMDB_WIKI_FACE(data_root='data')  
    X_train, y_train, X_val, y_val, lshape =  
        dataset.load(2000, 4000)  
    train_set = ArrayIterator(X=X_train, y=y_train,  
        nclass=2, lshape=lshape)  
    val_set = ArrayIterator(X=X_val, y=y_val,  
        nclass=2, lshape=lshape)  
    model, cost = generate_mlp_model()  
    ...
```



Using the Intel® neon™ Framework to train/test a multi-layer fully-connected network (4)

```
optimizer = GradientDescentMomentum(  
    0.01, momentum_coef=0.9,  
    stochastic_round=args.rounding, wdecay=0.0005)  
  
callbacks = Callbacks(model, eval_set=val_set,  
                      **args.callback_args)  
  
model.fit(train_set, optimizer=optimizer,  
          num_epochs=args.epochs, cost=cost,  
          callbacks=callbacks)  
accuracy = model.eval(val_set, metric=Accuracy())  
print('Accuracy = %.1f%%' % (accuracy * 100))
```

...



Using the Intel® neon™ Framework to train/test a multi-layer fully-connected network (5)

```
print('Making inference')
outputs = model.get_outputs(val_set)
with open('inference.txt', 'w') as output_file:
    output_file.write('inference,target\n')
    for i, entry in enumerate(outputs):
        output_file.write('%s,%s\n' % (entry, y_val[i]))
```



Notes

- The following samples of various deep models in the educational course differ in the implementation of the model generation function
- Calling the main function with different parameters allows us to work with different deep models and set the necessary training parameters



Infrastructure

- CPU: Intel® Xeon® CPU E5-2660 0 @ 2.20GHz
- GPU: Tesla K40s 11Gb
- OS: Ubuntu 16.04.4 LTS
- Frameworks:
 - Intel® neon™ Framework 2.6.0
 - CUDA 8.0
 - Python 3.5.2
 - Intel® Math Kernel Library 2017 (Intel® MKL)



Experiments

| Network id | Network structure | Learning parameters | Accuracy, % | Training time, s |
|-------------------|--|--|--------------------|-------------------------|
| FCNN-1 | Affine(128,0,Tanh), Affine(2,0,Logistic) | batch_size = 128 epoch_count = 30 | 71.2 | 932 |
| FCNN-2 | Affine(128,0,Tanh), Affine(64,0,Tanh), Affine(2,0,Logistic) | backend = gpu GradientDescentMomentum(0.01, momentum_coef=0.9, wdecay=0.0005) | 73.5 | 977 |
| FCNN-3 | Affine(400,0,Rectlin), Affine(50,0,Logistic), Affine(2,0,Logistic) | | 77.7 | 1013 |



Performance analysis for different backends (1)

| Network id | CPU | | GPU | | | CPU+MKL (16 threads) | | | | |
|---------------|-------|-------|-------|------|-------|----------------------|-------|------|-------|------|
| | S1, s | S2, s | S1, s | SS1 | S2, s | SS2 | S1, s | SS1 | S2, s | SS2 |
| FCNN-1 | 3735 | 27 | 932 | 4 | 14 | 1.93 | 2259 | 1.65 | 21 | 1.29 |
| FCNN-2 | 3803 | 27 | 977 | 3.89 | 13 | 2.08 | 2251 | 1.69 | 21 | 1.29 |
| FCNN-3 | 8067 | 36 | 1013 | 7.96 | 15 | 2.4 | 4406 | 1.83 | 30 | 1.2 |

- S1** corresponds to the training step
- S2** corresponds to the testing step
- SS1 = S1(CPU) / S1** is an acceleration training step in comparison with the CPU-version
- SS2 = S2(CPU) / S2** is an acceleration testing step in comparison with the CPU-version



Performance analysis for different backends (2)

| Network id | CPU | | GPU | | | CPU+MKL (16 threads) | | | | |
|---------------|-------|-------|-------|------|-------|----------------------|-------|------|-------|------|
| | S1, s | S2, s | S1, s | SS1 | S2, s | SS2 | S1, s | SS1 | S2, s | SS2 |
| FCNN-1 | 3735 | 27 | 932 | 4 | 14 | 1.93 | 2259 | 1.65 | 21 | 1.29 |
| FCNN-2 | 3803 | 27 | 977 | 3.89 | 13 | 2.08 | 2251 | 1.69 | 21 | 1.29 |
| FCNN-3 | 8067 | 36 | 1013 | 7.96 | 15 | 2.4 | 4406 | 1.83 | 30 | 1.2 |

- The training time using the CPU+MKL backend is less than the corresponding one for the CPU backend for the set of models
- The training time using the GPU backend is less than the corresponding one for the CPU + MKL backend for these models
- Using MKL allows you to improve the efficiency of calculations, which is of interest in the absence of GPUs
- Note:** further in the course GPU-backend is used



Improving performance of training step with Intel® Distribution for Python (1)

- CPU: Intel® Xeon® CPU E5-2660 0 @ 2.20GHz
- GPU: Tesla K40s 11Gb
- OS: Ubuntu 16.04.4 LTS
- Intel® neon™ Framework 2.6.0
- Intel® Distribution for Python 2018.3 (includes Python 3.6 and Intel® Math Kernel Library)
- CUDA 8.0



Improving performance of training step with Intel® Distribution for Python (2)

| Network id | CPU | | GPU | | | CPU+MKL (16 threads) | | | | |
|---------------|-------|-------|-------|------|-------|----------------------|-------|------|-------|------|
| | S1, s | S2, s | S1, s | SS1 | S2, s | SS2 | S1, s | SS1 | S2, s | SS2 |
| FCNN-1 | 1743 | 16 | 912 | 1.91 | 14 | 1.14 | 1495 | 1.17 | 15 | 1.07 |
| FCNN-2 | 1754 | 17 | 978 | 1.79 | 14 | 1.21 | 1543 | 1.14 | 15 | 1.13 |
| FCNN-3 | 3146 | 19 | 993 | 3.17 | 14 | 1.36 | 2505 | 1.26 | 18 | 1.06 |

- S1** corresponds to the training step
- S2** corresponds to the testing step
- SS1 = S1(CPU) / S1** is an acceleration training step in comparison with the CPU-version
- SS2 = S2(CPU) / S2** is an acceleration testing step in comparison with the CPU-version



Improving performance of training step with Intel® Distribution for Python (3)

| Network id | CPU | | | GPU | | | CPU+MKL (16 threads) | | |
|---------------|----------------------|--|-------------|----------------------|--|-------------|----------------------|--|-------------|
| | S1 (Python), s | S1 (Intel Distribution for Python), s | s | S1 (Python), s | S1 (Intel Distribution for Python), s | s | S1 (Python), s | S1 (Intel Distribution for Python), s | s |
| FCNN-1 | 3735 | 1743 | 2.14 | 932 | 912 | 1.02 | 2259 | 1495 | 1.51 |
| FCNN-2 | 3803 | 1754 | 2.17 | 977 | 978 | 1 | 2251 | 1543 | 1.46 |
| FCNN-3 | 8067 | 3146 | 2.56 | 1013 | 993 | 1.02 | 4406 | 2505 | 1.76 |

- S1 (Python)** corresponds to the training step with Python 3.5.2
- S1 (Intel Distribution for Python)** corresponds to the training step with Intel® Distribution for Python 2018.3 (includes Python 3.6 and Intel® Math Kernel Library)
- S = S1 (Python) / S1 (Intel Distribution for Python)**



Improving performance of training step with Intel® Distribution for Python (4)

| Network id | CPU | | | GPU | | | CPU+MKL (16 threads) | | |
|---------------|----------------------|--|-------------|----------------------|--|-------------|----------------------|--|-------------|
| | S1 (Python), s | S1 (Intel Distribution for Python), s | s | S1 (Python), s | S1 (Intel Distribution for Python), s | s | S1 (Python), s | S1 (Intel Distribution for Python), s | s |
| FCNN-1 | 3735 | 1743 | 2.14 | 932 | 912 | 1.02 | 2259 | 1495 | 1.51 |
| FCNN-2 | 3803 | 1754 | 2.17 | 977 | 978 | 1 | 2251 | 1543 | 1.46 |
| FCNN-3 | 8067 | 3146 | 2.56 | 1013 | 993 | 1.02 | 4406 | 2505 | 1.76 |

- ❑ Intel® Distribution for Python 2018.3 allows to improve performance of training step in 2 times approximately on CPU and in 1.5 times on CPU+MKL



COMPARISON OF THE INTEL® NEON™ FRAMEWORK AND OTHER DEEP LEARNING TOOLS



Comparison of the Intel® neon™ Framework and other deep learning tools (1)

| # | Tool | API | OS | FCNN | CNN | RNN | AE | RBM |
|----|------------------------|--|---------------------------------|------|-----|-----|----|-----|
| 1 | TensorFlow | Python, Java, Go | Linux, Windows, Mac OS, Android | + | + | + | + | + |
| 2 | Caffe | C++, Python, MATLAB | Linux, Windows, OS X | + | + | + | + | - |
| 3 | Keras | Python | Linux, Vagrant | + | + | + | + | + |
| 4 | Torch | C, Lua | Linux, iOS, Android | + | + | + | + | + |
| 5 | MXNet | C++, Python, R, Scala, Julia, Perl, MATLAB, JavaScript | Linux, Windows, Mac OS | + | + | + | + | + |
| 6 | Intel® neon™ Framework | Python | Linux, Mac OS | + | + | + | + | - |
| 7 | Theano | Python | Linux, Windows, Mac OS | + | + | + | + | + |
| 8 | Deepnet | Python | Linux | + | + | - | + | + |
| 9 | Deepmat | MATLAB | Linux, Windows, Mac OS, Solaris | + | + | - | + | + |
| 10 | Darch | R | Linux, Windows | + | - | - | + | + |

Comparison of the Intel® neon™ Framework and other deep learning tools (2)

- Easy to deploy unlike most libraries
- The deep models available in the Intel® neon™ Framework are similar to the well-known tools (Caffe, TensorFlow, Torch, MXNet and others)
- Convenient object-oriented interface of the Python language
 - The structure is similar to the Caffe and MXNet
 - Training of a deep model does not require direct implementation of back propagation algorithm, as in the Torch or TensorFlow
 - The possibility of extending the existing functionality (typical layers, activation functions, etc.)
- High performance on different hardware due to assembler-level optimization



Conclusion

- Overview of the Intel® neon™ Framework is introduced
- The general loop of deep model development in the Intel® neon™ Framework is considered
- An example of developing a deep fully-connected network using the Intel® neon™ Framework is explained
- Comparing the performance of training on different backends demonstrates that using MKL allows to increase the efficiency of calculations in comparison with the CPU
- Comparing the capabilities of the Intel® neon™ Framework with other deep learning tools shows that a similar basic models are implemented in neon. At the same time, higher performance is ensured



Literature

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



Authors

□ **Kustikova Valentina Dmitrievna**

Phd, lecturer, department of Computer software and supercomputer technologies, Institute of Information Technologies, Mathematics and Mechanics, Nizhny Novgorod State University

valentina.kustikova@itmm.unn.ru

□ **Zhiltssov Maxim Sergeevich**

master of the 1st year training, Institute of Information Technology, Mathematics and Mechanics, Nizhny Novgorod State University

zhiltsov.max35@gmail.com

□ **Zolotykh Nikolai Yurievich**

D.Sc., Prof., department of Algebra, geometry and discrete mathematics, Institute of Information Technologies, Mathematics and Mechanics, Nizhny Novgorod State University

nikolai.zolotykh@itmm.unn.ru

