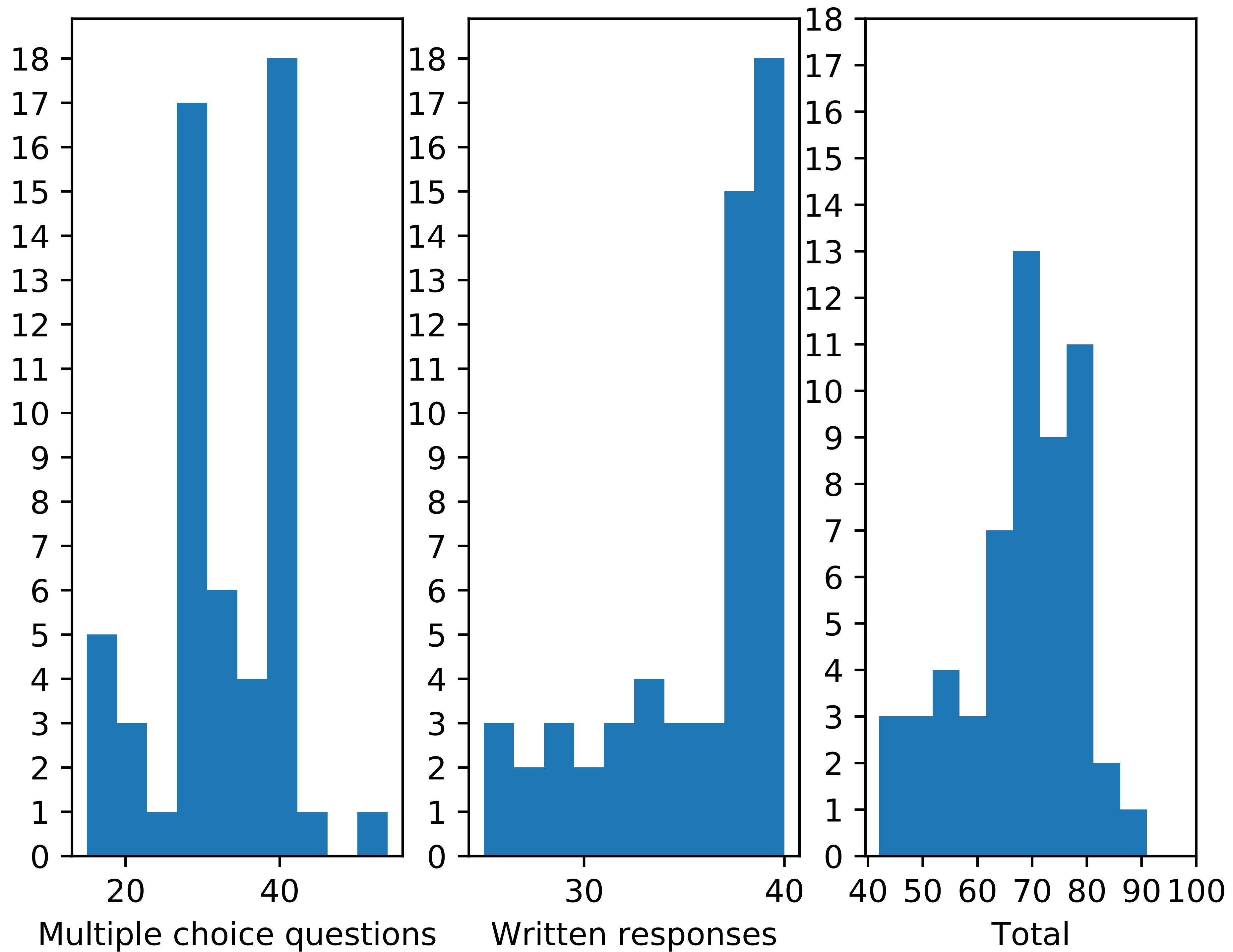
CSCI 5922 - NEURAL NETWORKS AND DEEP LEARNING

DEEP LEARNING SOFTWARE


Histograms of midterm scores
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
20
40
Multiple choice questions
30
40
Written responses
40
50
60
70
80
90
100
Total

HISTORY OF FRAMEWORKS

First frameworks to support define-by-run automatic differentiation.

FIRST GENERATION

Framework	Year
Torch	2002
Theano	2010
Torch7	2011
Theano	2012
Pylearn2	2013
Keras	2015
Torchnet	2015

SECOND GENERATION

Framework	Year
Autograd	2015
Chainer	2015
MXNet	2015
TensorFlow	2016
Theano	2016
DyNet	2017
PyTorch	2017
Ignite	2018
TensorFlow	2019

Wrappers

First full frameworks to support CUDA/GPUs.

TORCH - 2002 - HTTP://TORCH.CH

While this version of Torch pre-dates deep learning, it is the prototype for contemporary machine learning frameworks. Most contemporary deep learning frameworks, consciously or not, mimic Torch.

Torch is written in C++ and supports machine learning algorithms like multi-layer neural networks, support vector machines, Gaussian mixture models, and hidden Markov models.

It was made available under the BSD license (free to copy and commercial use/proprietary modifications are allowed as long as attribution is preserved).

The core API is inspired by object-oriented programming and design patterns – specifically, by the notions of modularity and separation of interface and implementation. The API contains useful abstractions like:

- DataSet
- Machine
- Measurer
- Trainer

DATASET CLASS

- Responsible for loading data
- Relieves engineer of need to repeatedly write code to read training data and labels
- Provides an abstraction layer that allows data to be read from any source
- Design pattern: Proxy

MACHINE CLASS

- Responsible for learning mapping from inputs to targets
- Several learning algorithms supported
  - Multi-layer neural network
  - Support vector machine
  - "Distribution"
    - Gaussian mixture model
    - Hidden Markov model
- Design pattern: Adapter

MEASURER CLASS

- Responsible for measuring the output of the machine
  - Loss: mean squared error, log loss
  - Metric: accuracy, F1, etc.
- Design pattern: ?

TRAINER CLASS

- Responsible for optimizing the Machine
  - Stochastic gradient trainer (multi-layer neural network)
  - Quadratic constrained trainer (support vector machine)
- Also responsible for ensembling
  - To train with/as an ensemble, an ordinary trainer is a delegate of a bagging or boosting trainer, e.g. (pseudocode):
    - BaggingTrainer(QuadraticConstrainedTrainer(…))
    - BoostingTrainer(StochasticGradientTrainer(…))
- Design pattern: Controller

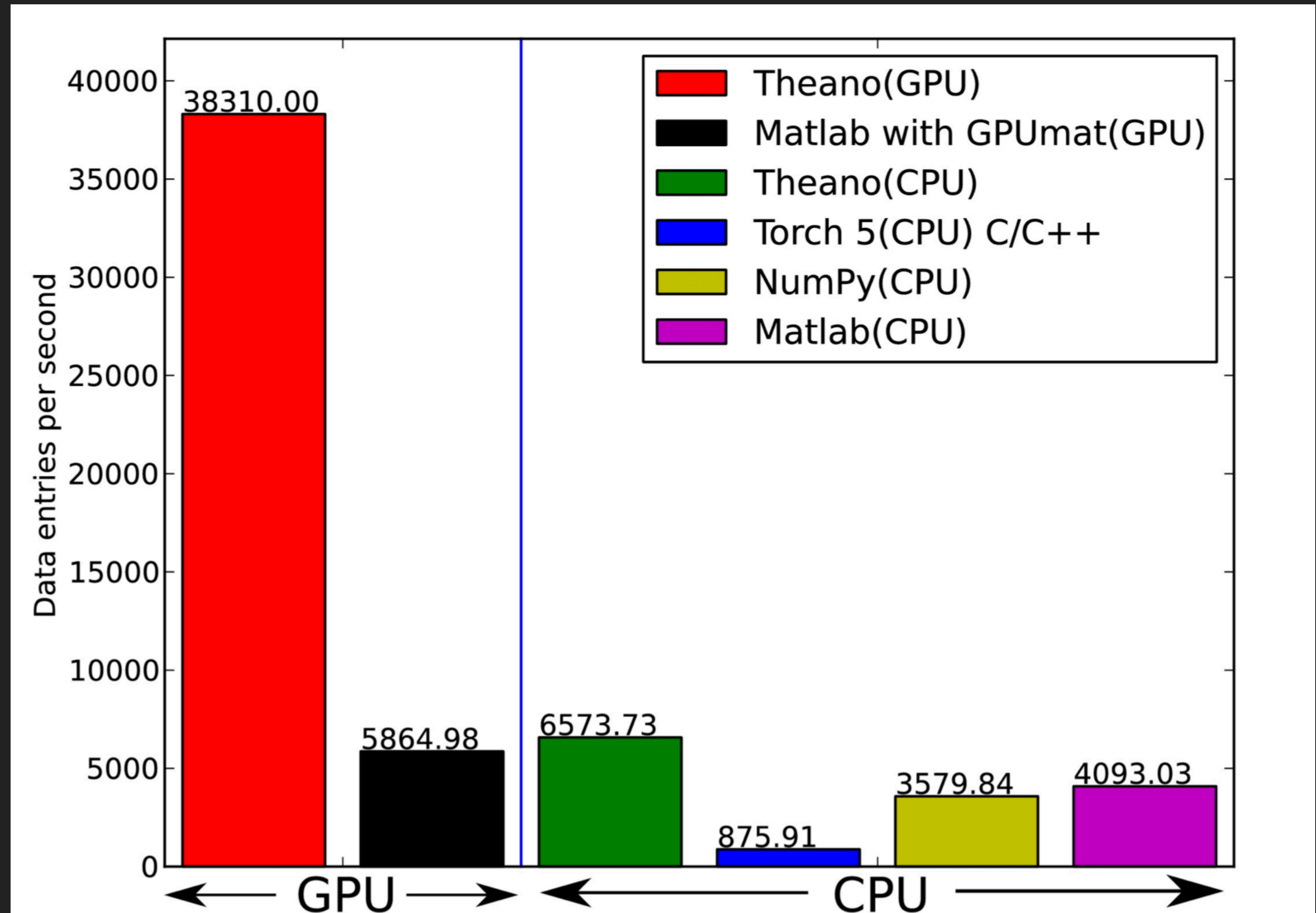
- Large-scale deep unsupervised learning using graphics processors, Raina, Madhavan, and Ng, 2009 [PDF]
- Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition, Ciresan, Meier, Gambardella, and Schmidhuber, 2010 [arXiv]
- ImageNet Classification with Deep Convolutional Neural Networks, Krizhevsky, Sutskever, and Hinton, 2012

ENTER GRAPHICS PROCESSING UNITS

The computational workhorse of multi-layer neural networks is matrix multiplication, which has complexity O(n^3).

Matrix multiplication can be seen a set of dot products between rows of the left operand and the columns of the right operand matrix. Naively, each dot product can be dispatched to a different core on a GPU.

With cores numbering in the thousands, GPUs can run many operations on matrices faster than CPUs can, even with slower clock cycles.



Bergstra et al, Scipy Conference, 2010

THEANO - 2011 - HTTP://WWW.DEEPLEARNING.NET/SOFTWARE/THEANO/

- Mathematical **symbolic** expression compiler
- Written in Python - better than C++ for rapid prototyping
- Attempts to conform to NumPy syntax and semantics
- With transparent support for GPUs

THEANO - 2011 - HTTP://WWW.DEEPLEARNING.NET/SOFTWARE/THEANO/

```python
import numpy as np
import theano
import theano.tensor as T

# Define the symbolic expression
x = T.scalar('x')
y = T.scalar('y')
z = x + y

# Define the function inputs and outputs. Calling this
# function causes C source code to be generated and compiled,
# either for the CPU or GPU, depending on your configuration.
f = theano.function(inputs=[x, y], outputs=z)

# Then call the function, which returns a numpy array.
results = f(1, 10)
```

```
# Now run it from the command line.
$ THEANO_FLAGS='device=cpu' python theano_example_1.py
11.0
```

THEANO - 2011 - [HTTP://WWW.DEEPLEARNING.NET/SOFTWARE/THEANO/](http://www.deeplearning.net/software/theano/)

LINEAR REGRESSION IN TWO SLIDES.

```python
from collections import OrderedDict

from sklearn.datasets import make_regression
import numpy as np
import theano
import theano.tensor as T


# Define the symblic expression, including the model parameters.
w = theano.shared(0., name='w')
b = theano.shared(0., name='b')
x = T.vector('x')
y = T.scalar('y')
output = w*x + b

# Define the scalar cost and state what gradients to compute.
cost = ((output - y)**2).mean()
gw, gb = T.grad(cost, [w, b])

# Define the updates.
updates = OrderedDict()
updates[w] = w-0.1*gw
updates[b] = b-0.1*gb

# Define the function inputs and outputs.
f = theano.function(inputs=[x, y], outputs=output, updates=updates)
```

LINEAR REGRESSION IN TWO SLIDES.

```
# Now make a regression dataset with a known coefficient and bias.
bias=66.
n_samples=20
X, y, coef = make_regression(
    n_samples=n_samples, n_features=1, coef=True, bias=bias)

# Then iterate over the training examples. The updates are automatically applied.
for i in range(len(X)):
    results = f(X[i], y[i])

print('True coefficient {:.03f} bias {:.03f}'.format(coef, bias))
print('Estimated coefficient {:.03f} bias {:.03f}'.format(w.get_value(), b.get_value()))
```

```
# Now run it from the command line.
$ THEANO_FLAGS='device=cpu,floatX=float64' python theano_example_2.py
True coefficient 82.199 bias 66.000
Estimated coefficient 83.316 bias 66.895
```

THEANO - 2011 - [HTTP://WWW.DEEPLEARNING.NET/SOFTWARE/THEANO/](http://www.deeplearning.net/software/theano/)

```
import numpy as np

M = np.random.normal(size=(10, 5))
result = M.copy()
k = 10
for i in range(k):
    result = result * M
```

COMPUTING ELEMENT-WISE POWERS OF A MATRIX IN NUMPY

COMPUTING THE SAME IN THEANO

```
import theano
import theano.tensor as T

k = T.iscalar('k')
A = T.vector('A')

# Symbolic description of the result
result, updates = theano.scan(
    fn=lambda prior_result, A: prior_result * A,
    outputs_info=T.ones_like(A),
    non_sequences=A,
    n_steps=k)

# We only care about A**k, but scan has provided us with A**1 through
# A**k. Discard the values that we don't care about. Scan is smart enough
# to notice this and not waste memory saving them.
final_result = result[-1]

# Compiled function that returns A**k
power = theano.function(inputs=[A,k], outputs=final_result, updates=updates)

results = power(range(10), 2)
```

SOME LIMITATIONS OF THEANO

- Poor expressivity
  - Example: theano.scan
- Corollary: Not a true automatic differentiation framework
- Because of compilation step (either to CPU or GPU), there can be a substantial delay between program invocation and execution. The delays for recurrent networks could be substantial.
- No longer being actively developed, because of success of other frameworks

- Torch7 was a continuation of the earlier versions of Torch, with nice, modular design.
- GPU support - easy to move tensors to and from GPU
- Define-then-run, but **not** symbolic
- Written in Lua (!)
- Rationale for Lua was ease of extensibility (in C++)
- Super fast

TOP-LEVEL PACKAGES

- torch - numerical library
- nn - neural networks
- optim - optimization
- image - image loading, preprocessing, and manipulation
- paths - filesystem-related functions

TORCH7 - 2012

THE NOTION OF CONTAINERS IN E.G. KERAS ORIGINATED IN TORCH.

A CONTAINER IS AN INSTANCE OF THE MODULE CLASS, AND INSTANCES OF MODULE ARE ADDED TO THE CONTAINER.

A CONTAINER CAN BE ADDED TO ANOTHER CONTAINER.

```
require 'nn';

model = nn.Sequential()

# First convolution.
model:add(nn.SpatialConvolutionMM(1, 32, 5, 5))
model:add(nn.Tanh())
model:add(nn.SpatialMaxPooling(2, 2, 2, 2))

# Second convolution.
model:add(nn.SpatialConvolutionMM(32, 64, 5, 5))
model:add(nn.Tanh())
model:add(nn.SpatialMaxPooling(2, 2, 2, 2))

# Fully-connected layers.
model:add(nn.Reshape(64 * 4 * 4))
model:add(nn.Linear(64 * 4 * 4, 200))
model:add(nn.Tanh())
model:add(nn.Linear(200, 10))
```

TORCHNET - 2015

CAN MOVE TENSORS TO AND FROM GPU AT WILL.

ENGINE IN TORCHNET SIMILAR TO TRAINER IN 2002 TORCH.

HOOKS ALLOW USER TO RUN CODE AT CERTAIN POINTS IN EXECUTION.

```lua
require 'nn'
require 'torchnet'
require 'cunn'

local net = nn.Sequential():add(nn.Linear(784,10))
local criterion = nn.CrossEntropyCriterion()

-- Put network and loss function on GPU.
net = net:cuda()
criterion = criterion:cuda()

-- CudaTensor is put on GPU by default.
local input = torch.CudaTensor()
local target = torch.CudaTensor()

local engine = torchnet.SGDEngine()


--[[
Each time the engine receives a new sample from the dataset iterator,
resize the input and target tensors to match the sizes in the minibatch,
and copy from the CPU to the GPU.
]]--
engine.hooks.onSample = function(state)
  input:resize(
    state.sample.input:size()
  ):copy(state.sample.input)

  target:resize(
    state.sample.target:size()
  ):copy(state.sample.target)

  state.sample.input = input
  state.sample.target = target
end
```

SOME LIMITATIONS OF TORCH7

- Define-then-run
- Lua (particularly for NLP tasks, but even for vision)

PYLEARN2 - 2013

PYLEARN2 WAS A WRAPPER FOR THEANO. IT WAS WRITTEN AT THE UNIVERSITY OF MONTREAL IN THE SAME LAB THAT CREATED THEANO.

A YAML FILE DECLARED THE ELEMENTS OF THE SYSTEM.

MUCH LIKE THE ORIGINAL TORCH, IT HAD NOTIONS OF DATASET, MODEL, AND OPTIMIZER (ALGORITHM).

```
!obj:pylearn2.train.Train {
  "dataset": !obj:pylearn2.datasets.dense_design_matrix.DenseDesignMatrix &dataset {
    "X" : !obj:numpy.random.normal { 'size':[5,3] },
  },
  "model": !obj:pylearn2.models.autoencoder.DenoisingAutoencoder {
    "nvis" : 3,
    "nhid" : 4,
    "irange" : 0.05,
    "corruptor": !obj:pylearn2.corruption.BinomialCorruptor {
      "corruption_level": 0.5,
    },
    "act_enc": "tanh",
    "act_dec": null,    # Linear activation on the decoder side.
  },
  "algorithm": !obj:pylearn2.training_algorithms.sgd.SGD {
    "learning_rate" : 1e-3,
    "batch_size" : 5,
    "monitoring_dataset" : *dataset,
    "cost" : !obj:pylearn2.costs.autoencoder.MeanSquaredReconstructionError {},
    "termination_criterion" : !obj:pylearn2.termination_criteria.EpochCounter {
      "max_epochs": 1,
    },
  },
  "save_path": "./garbage.pkl"
}
```

KERAS WAS ORIGINALLY A WRAPPER FOR THEANO.

IT EMULATED TORCH7. NOTICE SIMILARITIES SUCH AS SEQUENTIAL AND MODEL.ADD.

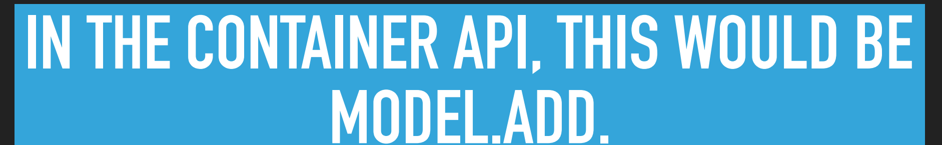
```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

# Generate dummy data
import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(
    np.random.randint(10, size=(1000, 1)), num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(
    np.random.randint(10, size=(100, 1)), num_classes=10)

model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units. In the
# first layer, you must specify the expected input data shape: here,
# 20-dimensional vectors.
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(
    loss='categorical_crossentropy', optimizer=sgd,
metrics=['accuracy'])

model.fit(x_train, y_train, epochs=20, batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

KERAS - 2015

The Keras functional API does away with containers and connects layers to their successors in the computational graph by passing the predecessor as an argument.

This replaces model.add with Python's __call__ method – effectively associating predecessors and successors with a pseudo-closure.

How might this be more flexible than a sequential container?




```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor.
inputs = Input(shape=(784,))

# A layer instance is callable on a tensor, and returns a tensor.
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes the Input layer and three Dense
# layers.
model = Model(inputs=inputs, outputs=predictions)
model.compile(
    optimizer='rmsprop',
    loss='categorical_crossentropy',
    metrics=['accuracy'])
model.fit(data, labels)
```