Deep Learning on Azure

Introduction to Deep Learning with CNTK

Jarek Kazmierczak
MTC Silicon Valley

Microsoft Cognitive Toolkit

- Microsoft's open-source deep-learning toolkit
  - https://github.com/Microsoft/CNTK   Star 10,661   Fork 2,692
  - Created by Microsoft Speech researchers (Dong Yu et al.) in 2012, "Computational Network Toolkit"
  - On GitHub since Jan 2016 under MIT license
  - Renamed from CNTK to "Cognitive Toolkit"
  - Community contributions e.g. from MIT, Stanford and NVidia

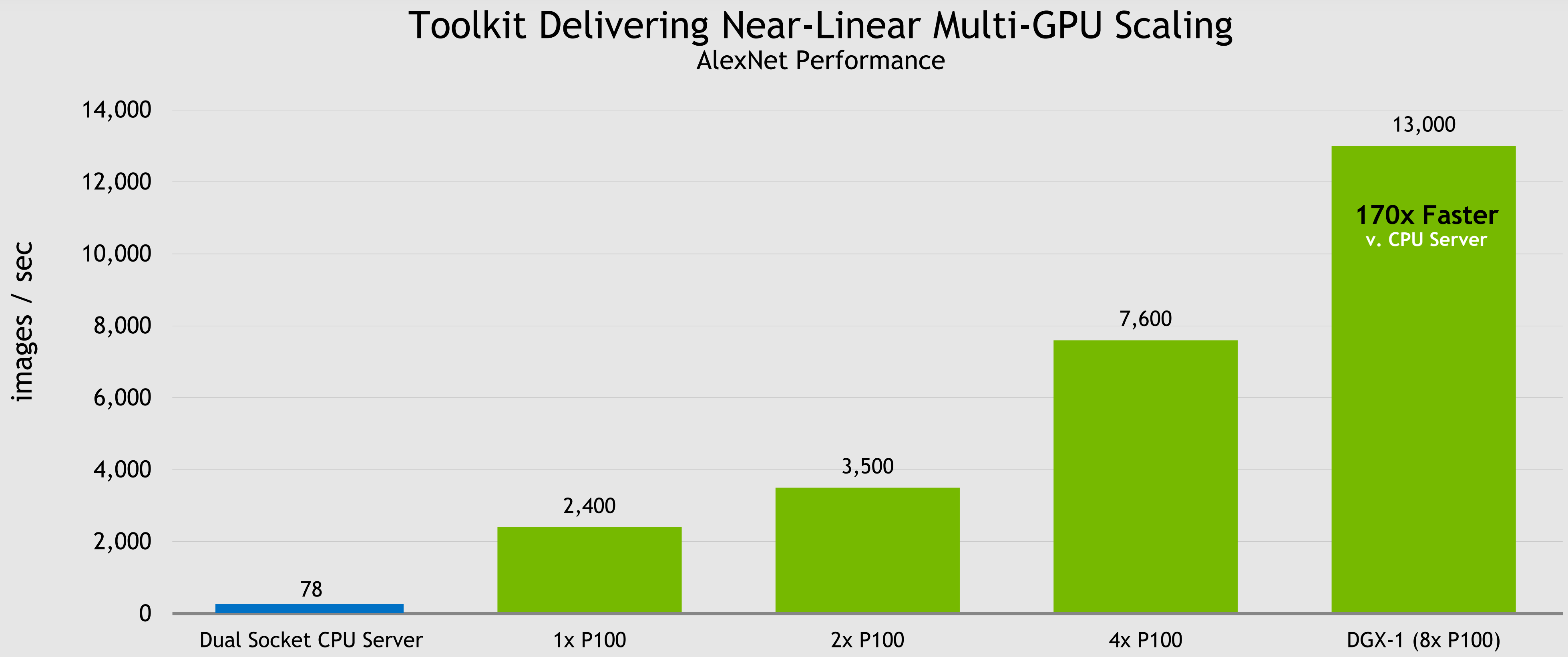
Microsoft Cognitive Toolkit
aka.ms/CognitiveToolkit
Microsoft

Microsoft Cognitive Toolkit

- Runs over 80% Microsoft internal DL workload
- 1st-class on Linux and Windows, docker support
- Training: Python, C++,
- Evaluation: C#, Java, Scale up evaluation in Spark
- Internal == External
- New in GA:
  - Keras backend support (Beta)
  - Java support, Spark support
  - Model compression (Fast binarized evaluation)

MICROSOFT COGNITIVE TOOLKIT

First Deep Learning Framework Fully Optimized for Pascal



AlexNet training batch size 128, Grad Bit = 32, Dual socket E5-2699v4 CPUs (total 44 cores)
CNTK 2.0b3 (to be released) includes cuDNN 5.1.8, NCCL 1.6.1, NVLink enabled

SCALABILITY

Image: Cray

Microsoft, Cray claim deep learning breakthrough on supercomputers

Steve Ranger
ZDNet

A team of researchers from Microsoft, Cray, and the Swiss National Supercomputing Centre (CSCS) have been working on a project to speed up the use of deep learning algorithms on supercomputers.

The team have scaled the Microsoft Cognitive Toolkit -- an open-source suite that trains deep learning algorithms -- to more than 1,000 Nvidia Tesla P100 GPU accelerators on the Swiss centre's Cray XC50 supercomputer, which is nicknamed Piz Daint.

CNTK expresses (nearly) **arbitrary neural networks** by composing simple building blocks into complex **computational networks**, supporting relevant network types and applications.

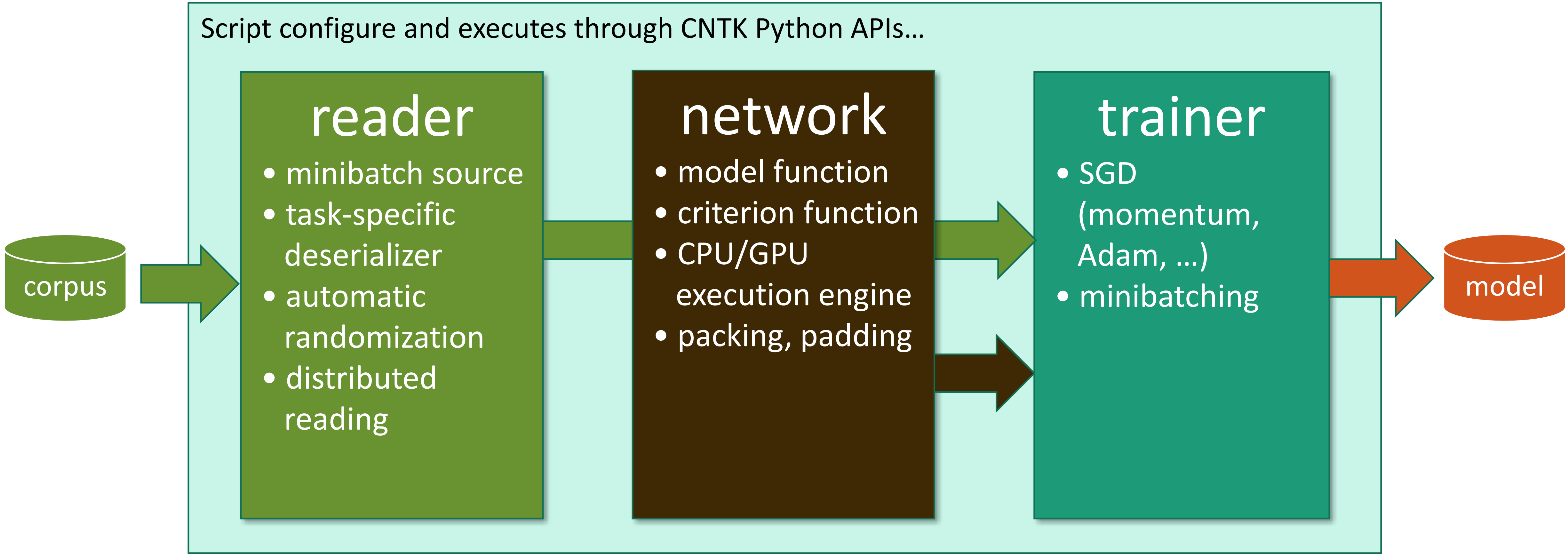
CNTK Programming Model

- Networks are Function Objects
- Complicated networks can be composed as hierarchies of simpler ones
- Similar to Keras, Chainer, Dynet, Pytorch and Sonnet
- The Function object is a single abstraction used to represent different operations, which are only distinguished by convention
- The Function object is a Python wrapper around a C++ graph structure
- There are two styles of API:
  - Lower level Graph API
  - Higher level Functional API

CNTK Data Model

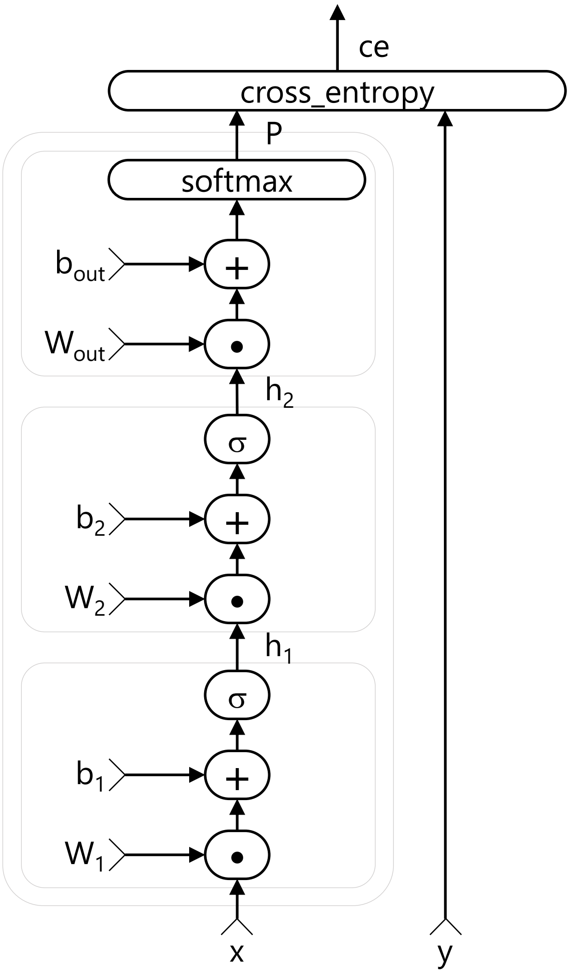
- CNTK operates on two types of data:
  - Tensors
  - Sequences of tensors
- Tensors are N-dimensional arrays that can be dense or sparse
- Tensors have statics axes (dimensions)
- Sequences are like tensors but have an additional dynamic axis – variable length axis which length depends on input data

Anatomy of a CNTK training job



Neural networks as graphs

CNTK expresses (nearly) arbitrary neural networks by composing simple building blocks into complex computational networks, supporting relevant network types and applications

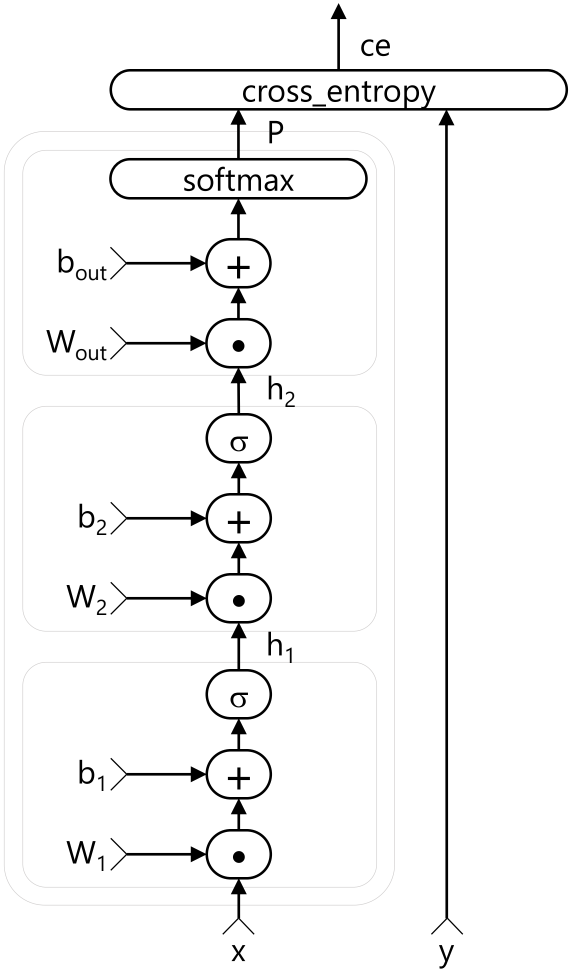


- nodes: functions (primitives)
  - can be composed into reusable composites
- edges: values
  - incl. tensors, sparse
- automatic differentiation and SGD
  - $\partial\mathcal{F} / \partial in = \partial\mathcal{F} / \partial out \cdot \partial out / \partial in$
- deferred computation → execution engine
- editable, clonable

Graphs are the “assembly language” of DNN tools

How to: network

- "model function"
  - *features* → *predictions*
  - defines the model structure & parameter initialization
  - holds parameters that will be learned by training
- "criterion function"
  - (*features, labels*) → (*training loss, additional metrics*)
  - defines training and evaluation criteria on top of the model function
  - provides gradients w.r.t. training criteria



How to: network

example: 2-hidden layer feed-forward NN

$$
\begin{aligned}
h_1 &= \sigma(\mathbf{W}_1\, x + b_1) \\
h_2 &= \sigma(\mathbf{W}_2\, h_1 + b_2) \\
P &= \mathrm{softmax}(\mathbf{W}_{\mathrm{out}}\, h_2 + b_{\mathrm{out}})
\end{aligned}
$$

with input $x \in \mathrm{R}^M$ and one-hot label $y \in \mathrm{R}^J$
and cross-entropy training criterion

$$
\begin{aligned}
ce &= y^{\mathrm{T}} \log P \\
\textstyle\sum_{\mathrm{corpus}} ce &= \max
\end{aligned}
$$

How to: network

example: 2-hidden layer feed-forward NN

$$
\begin{aligned}
h_1 &= \sigma(\mathbf{W}_1\, x + b_1) \\
h_2 &= \sigma(\mathbf{W}_2\, h_1 + b_2) \\
P &= \mathrm{softmax}(\mathbf{W}_{\mathrm{out}}\, h_2 + b_{\mathrm{out}})
\end{aligned}
$$

```
h1 = sigmoid (x  @ W1   + b1)
h2 = sigmoid (h1 @ W2   + b2)
P  = softmax (h2 @ Wout + bout)
```

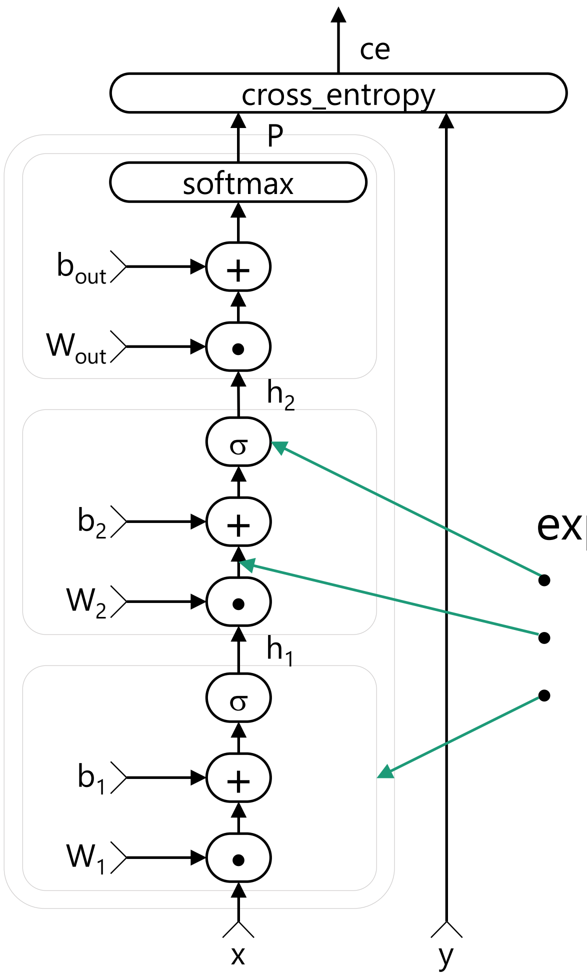
with input $x \in \mathrm{R}^M$ and one-hot label $y \in \mathrm{R}^J$
and cross-entropy training criterion

$$
\begin{aligned}
ce &= y^{\mathrm{T}} \log P \\
\textstyle\sum_{\mathrm{corpus}} ce &= \max
\end{aligned}
$$

```
ce = cross_entropy (P, y)
```

Networks as graphs

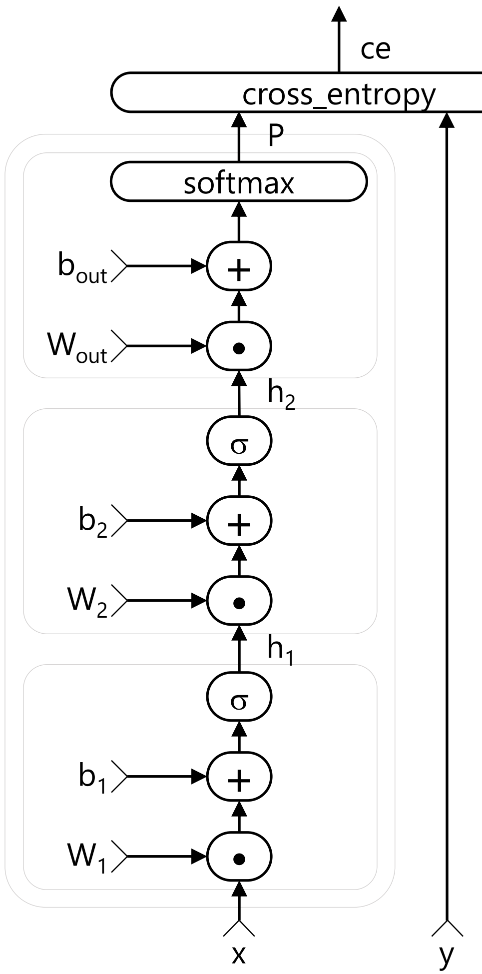
Graphs are the "assembly language" of DNN tools



expression tree with
- primitive ops
- values (tensors)
- composite ops

```
h1 = sigmoid (x  @ W1   + b1)
h2 = sigmoid (h1 @ W2   + b2)
P  = softmax (h2 @ Wout + bout)
ce = cross_entropy (P, y)
```

Authoring networks as functions

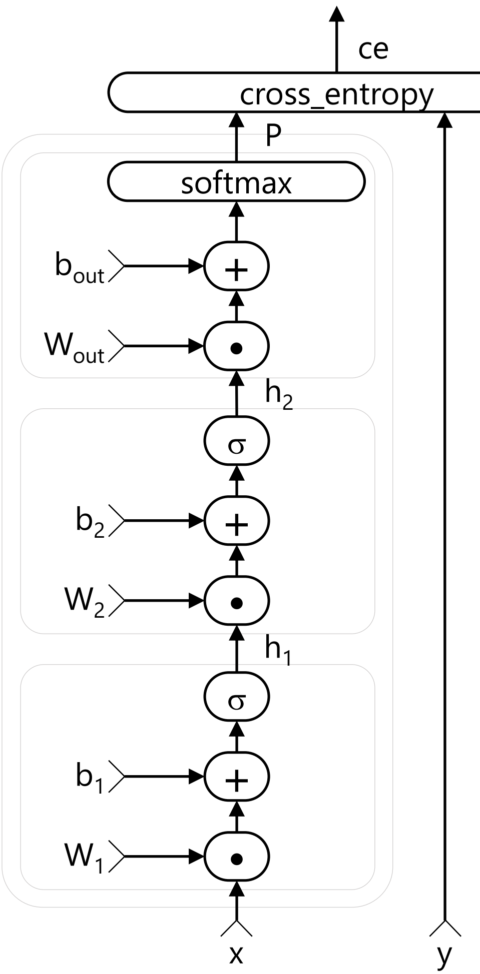



```
# --- graph building with function objects ---
M = 40 ; H = 512 ; J = 9000  # feat/hid/out dim
#  - function objects own the learnable parameters
#  - here used as blocks in graph building
x = Input(M) ;  y = Input(J)  # feat/labels
h1 = Dense(H, activation=sigmoid)(x)
h2 = Dense(H, activation=sigmoid)(h1)
P  = Dense(J, activation=softmax)(h2)
ce = cross_entropy(P, y)
```

Layers API

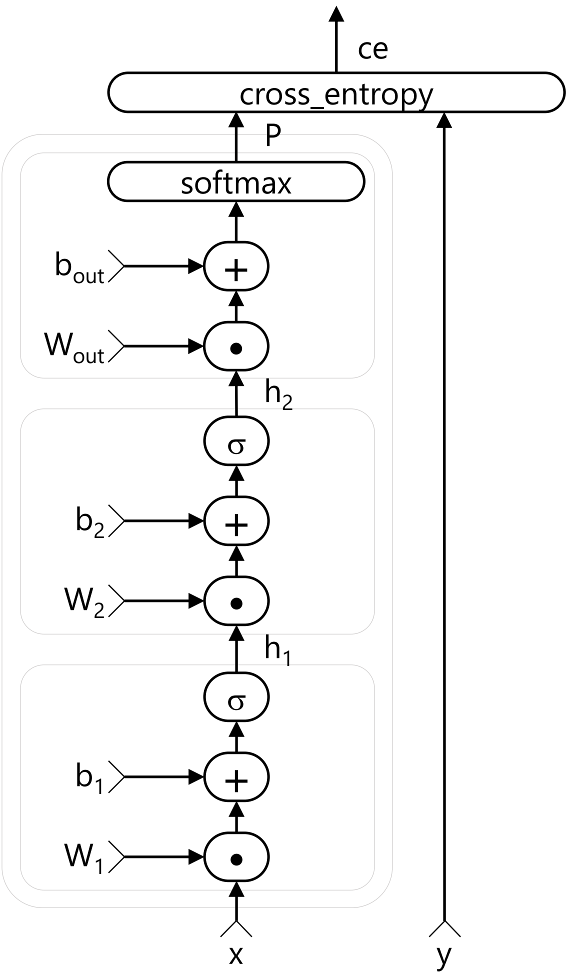
- basic blocks:
  - `LSTM(), GRU(), RNNUnit()`
  - `Stabilizer(), identity`
  - `ForwardDeclaration(), Tensor[], SparseTensor[], Sequence[], SequenceOver[]`
- layers:
  - `Dense(), Embedding()`
  - `Convolution(), Convolution1D(), Convolution2D(), Convolution3D(), Deconvolution()`
  - `MaxPooling(), AveragePooling(), GlobalMaxPooling(), GlobalAveragePooling(), MaxUnpooling()`
  - `BatchNormalization(), LayerNormalization()`
  - `Dropout(), Activation()`
  - `Label()`
- composition:
  - `Sequential(), For(), operator >>, (function tuples)`
  - `ResNetBlock(), SequentialClique()`
- sequences:
  - `Delay(), PastValueWindow()`
  - `Recurrence(), RecurrenceFrom(), Fold(), UnfoldFrom()`
- models:
  - `AttentionModel()`

Authoring networks as functions



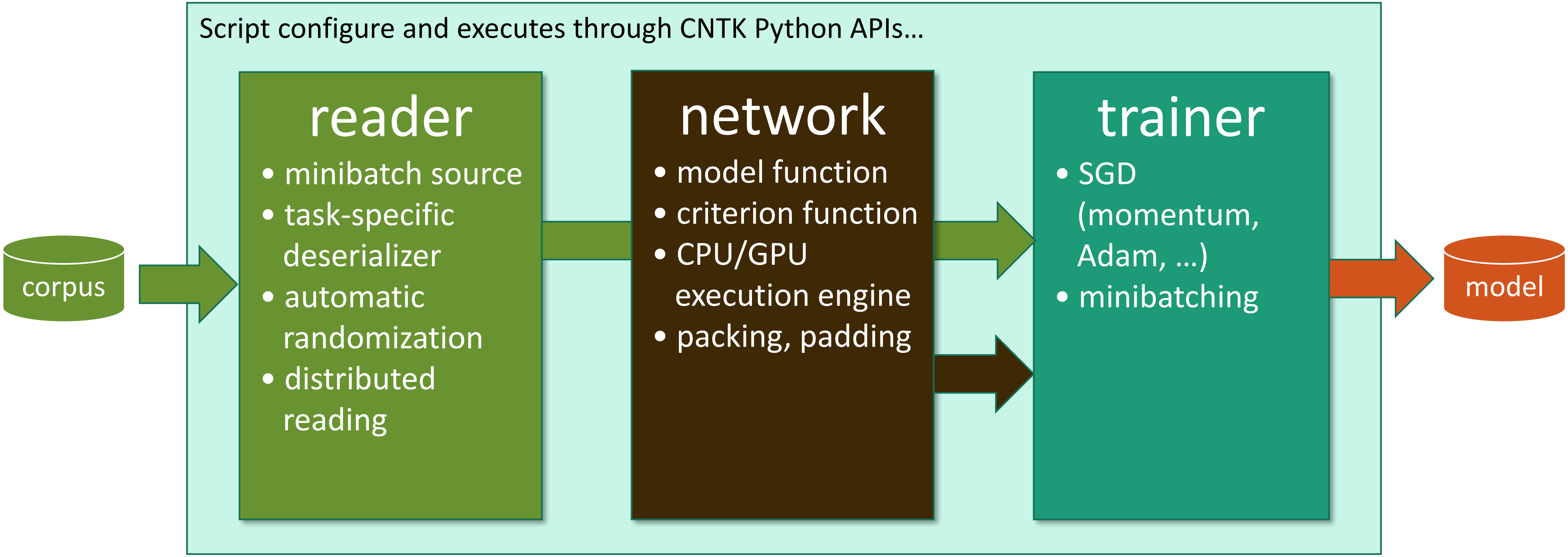

```
# --- model function composition ---
M = 40 ; H = 512 ; J = 9000  # feat/hid/out dim

# function objects compose the model
model = (Dense(H, activation=sigmoid) >>
         Dense(H, activation=sigmoid) >>
         Dense(J, activation=softmax))
# criterion still graph-building
x = Input(M) ; y = Input(J)  # feat/labels
P = model(x)
ce = cross_entropy(P, y)
```

How to: trainer

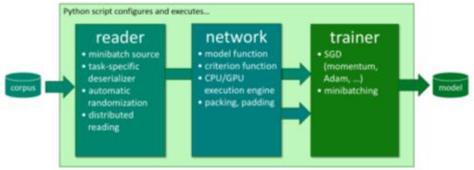



```
# --- model function composition ---
M = 40 ; H = 512 ; J = 9000  # feat/hid/out dim
# function objects compose the model
model = (Dense(H, activation=sigmoid) >>
         Dense(H, activation=sigmoid) >>
         Dense(J, activation=softmax))
# criterion still graph-building
x = Input(M) ; y = Input(J)  # feat/labels
P = model(x) ; ce = cross_entropy(P, y)
learner = sgd(P.parameters, …)
Trainer = Trainer(P, (ce), [learner])
```

Anatomy of a CNTK training job



How to: reader




```python
def create_reader(map_file, mean_file, is_training):
    # image preprocessing pipeline
    transforms = [
        ImageDeserializer.crop(crop_type='Random', ratio=0.8, jitter_type='uniRatio')
        ImageDeserializer.scale(width=image_width, height=image_height, channels=num_channels,
                                interpolations='linear'),
        ImageDeserializer.mean(mean_file)
    ]
    # deserializer
    return MinibatchSource(ImageDeserializer(map_file, StreamDefs(
        features = StreamDef(field='image', transforms=transforms), '
        labels   = StreamDef(field='label', shape=num_classes)
    )), randomize=is_training, epoch_size = INFINITELY_REPEAT if is_training else FULL_DATA_SWEEP)
```

- automatic on-the-fly randomization important for large data sets
- readers compose, e.g. image → text caption

Anatomy of a CNTK training job