Microsoft's Cognitive Toolkit (CNTK)

The Computer Science Behind the Microsoft Open-Source Toolkit for Large-Scale Deep-Learning

Frank Seide

fseide@microsoft.com

Principal Researcher Speech Recognition, CNTK Architect
Microsoft Research

With many contributors:

A. Agarwal, E. Akchurin, C. Basoglu, B. Bozza, G. Chen, S. Cyphers, W. Darling, J. Droppo, A. Eversole, B. Guenter, M. Hillebrand, X.-D. Huang, Z. Huang, W. Richert, R. Hoens, V. Ivanov, A. Kamenev, N. Karampatziakis, P. Kranen, O. Kuchaiev, W. Manousek, C. Marschner, A. May, B. Mitra, O. Nano, G. Navarro, A. Orlov, P. Parthasarathi, B. Peng, M. Radmilac, A. Reznichenko, W. Richert, M. Seltzer, M. Slaney, A. Stolcke, T. Will, H. Wang, W. Xiong, K. Yao, D. Yu, Y. Zhang, G. Zweig, and many more

Special acknowledgment to Dong Yu, Jasha Droppo, John Langford, Jacob Devlin, Sean McDirmid (Y-Combinator Research), Bruno Bozza

Microsoft

deep learning at Microsoft

- Microsoft Cognitive Services
- Skype Translator
- Cortana
- Bing
- HoloLens
- Microsoft Research

Cortana

Bing ads

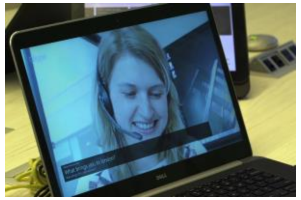
Bing

Microsoft HoloLens

¡Hola!

Hello!

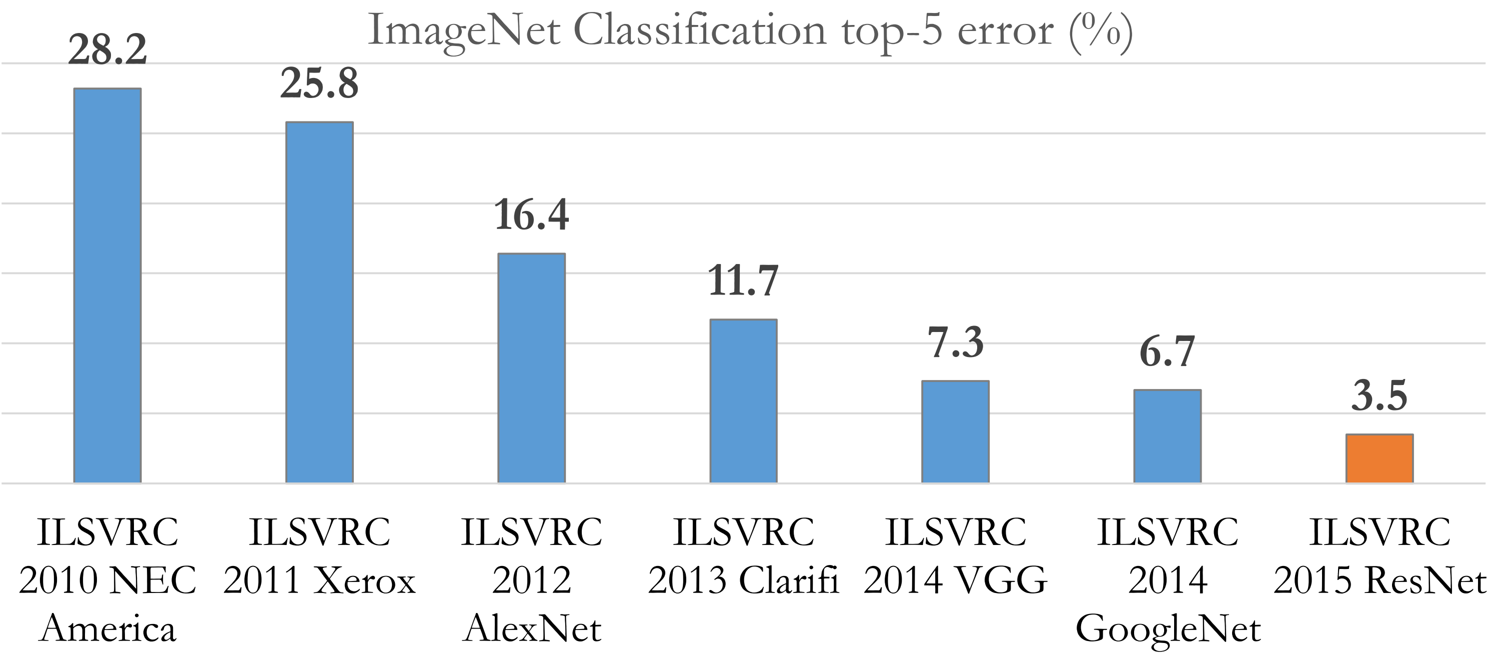
skype



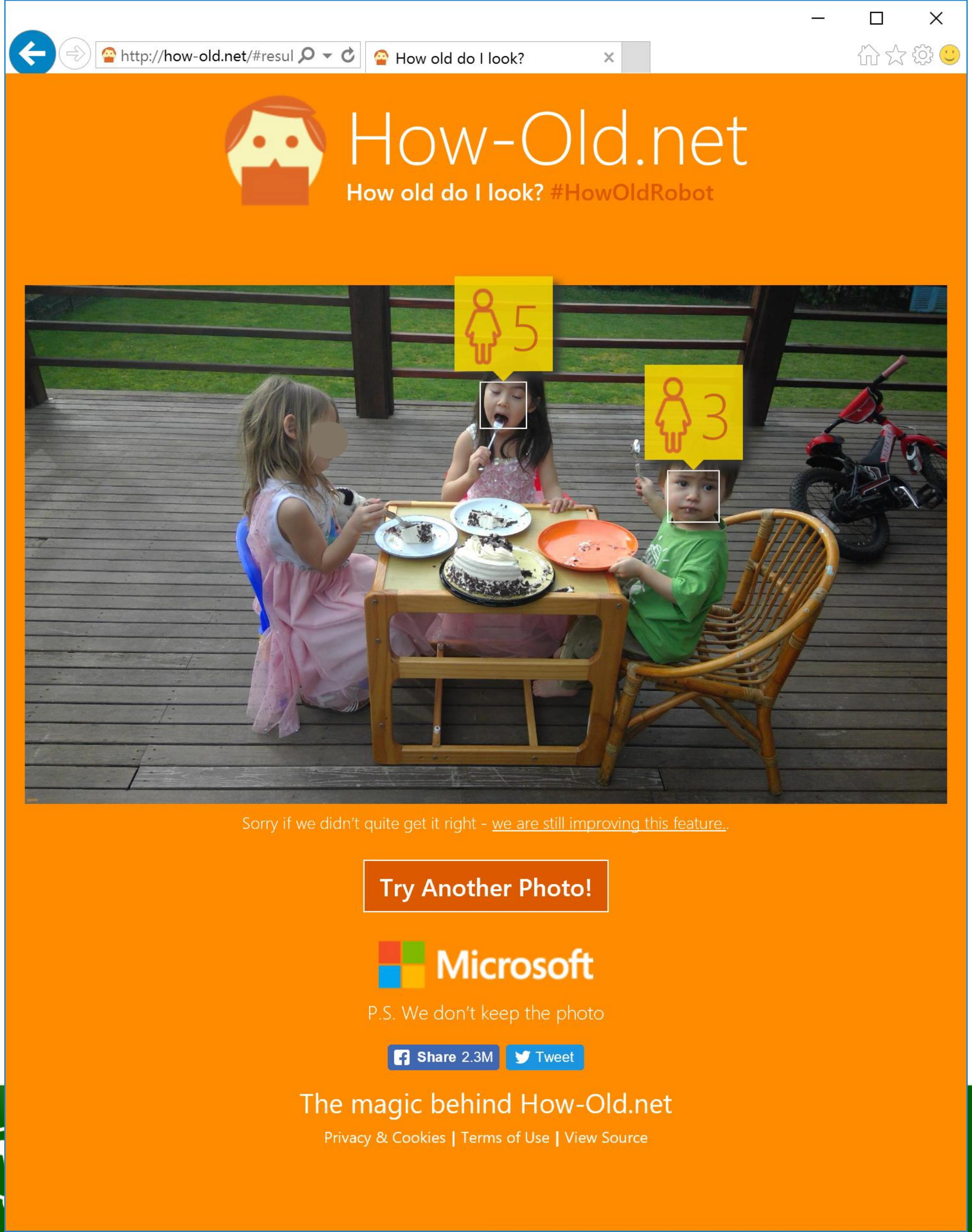
XBOX

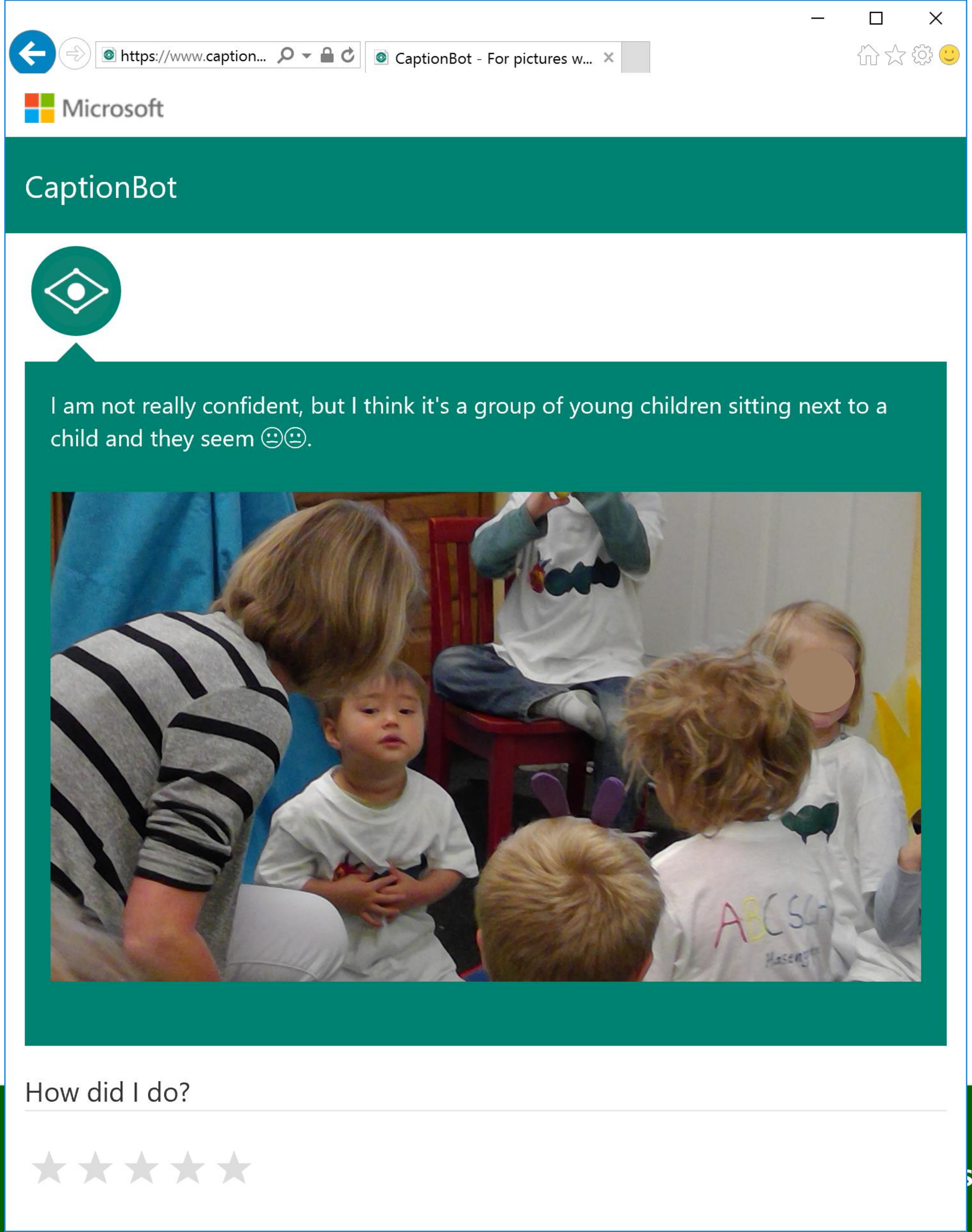
Microsoft Research

ImageNet: Microsoft 2015 ResNet



Microsoft had all **5 entries** being the 1-st places this year: ImageNet classification, ImageNet localization, ImageNet detection, COCO detection, and COCO segmentation


http://how-old.net/#resul
How old do I look?
How-Old.net
How old do I look? #HowOldRobot
5
3
Sorry if we didn't quite get it right - we are still improving this feature..
Try Another Photo!
Microsoft
P.S. We don't keep the photo
Share 2.3M
Tweet
The magic behind How-Old.net
Privacy & Cookies | Terms of Use | View Source


https://www.caption...
CaptionBot - For pictures w...
Microsoft
CaptionBot
I am not really confident, but I think it's a group of young children sitting next to a child and they seem 😐😐.
How did I do?

Uber to require selfie security check from drivers

Using Microsoft Cognitive Services, Uber hopes to make riders feel safer by verifying the ID of drivers before rides are given.

By Jake Smith for iGeneration | September 23, 2016 -- 19:59 GMT (03:59 GMT+08:00) | Topic: Innovation

Uber announced on Friday a new security feature called Real-Time ID Check that will require drivers to periodically take a selfie before starting their driving shift.

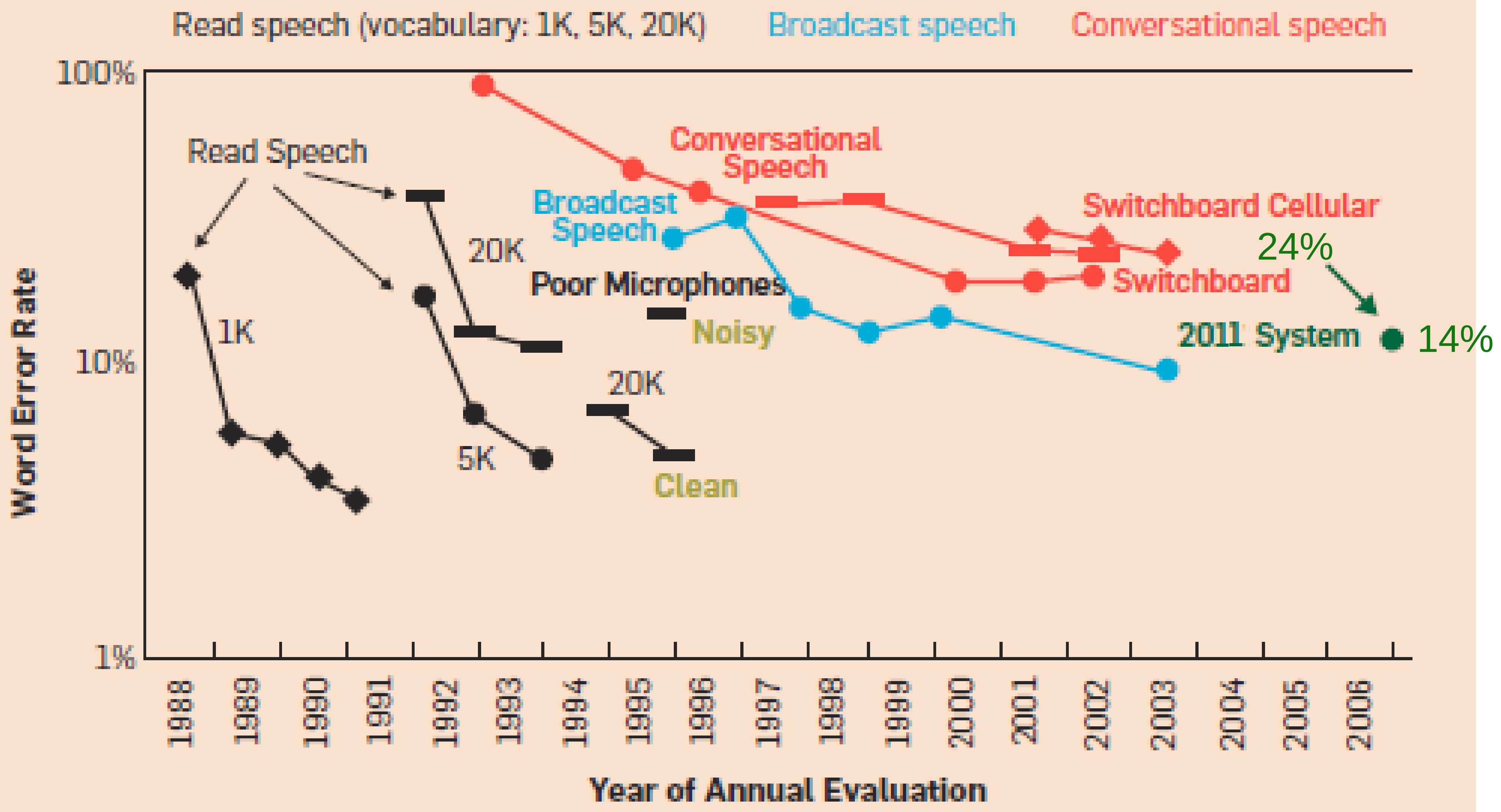
The feature, which begins rolling out to US cities on Friday, uses Microsoft Cognitive Services to reduce fraud and give riders an extra sense of security.

Uber says Microsoft's feature instantly compares the selfie to the one corresponding with the account on file. If the two

RELATED STORIES

Innovation
Victoria partners with Bosch for self-driving vehicle development

Figure 1. Historical progress of speech recognition word error rate on more and more difficult tasks.[10] The latest system for the switchboard task is marked with the green dot.



Microsoft's historic speech breakthrough

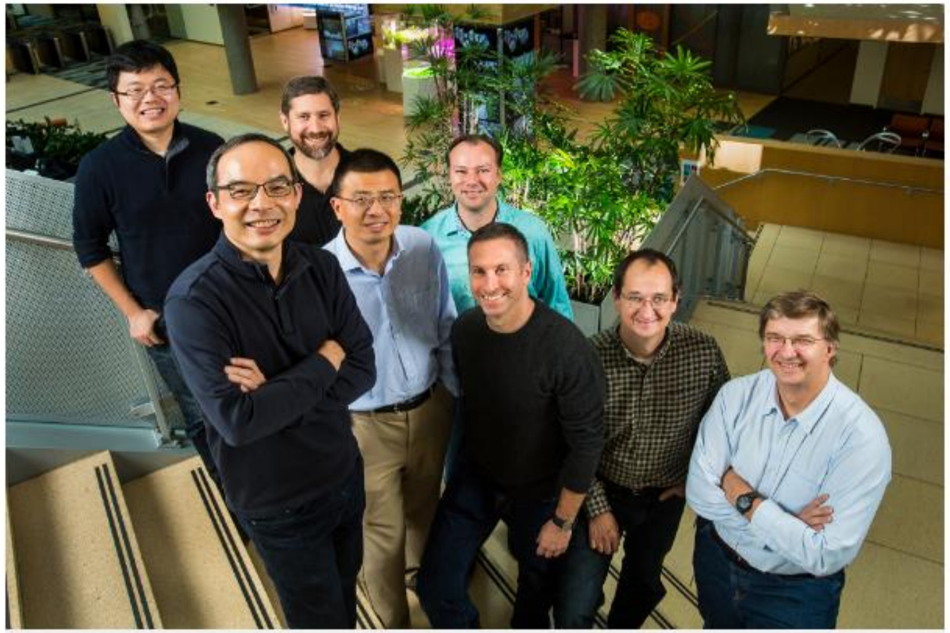
- Microsoft 2016 research system for conversational speech recognition
- 5.9% word-error rate
- enabled by CNTK's multi-server scalability

[W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, G. Zweig: "Achieving Human Parity in Conversational Speech Recognition," https://arxiv.org/abs/1610.05256]

Historic Achievement: Mi

https://blogs.microsoft.com/next/2016/10/18/historic-achievement-micrc

Historic Achievement: Microsoft researchers reach human parity in conversational speech recognition



Microsoft researchers from the Speech & Dialog research group include, from back left, Wayne Xiong, Geoffrey Zweig, Xuedong Huang, Dong Yu, Frank Seide, Mike Seltzer, Jasha Droppo and Andreas Stolcke. (Photo by Dan DeLong)

Posted October 18, 2016

By **Allison Linn**

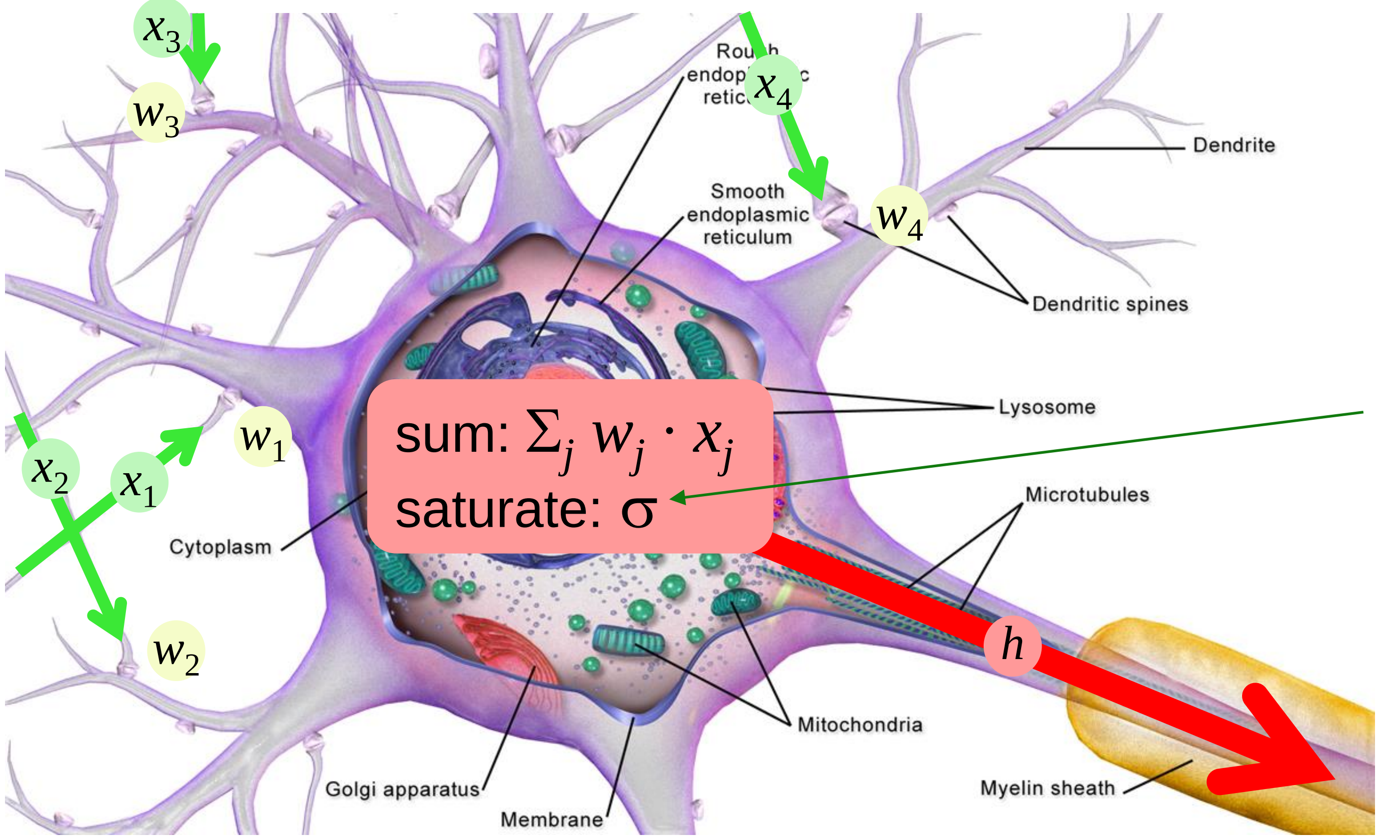
Microsoft has made a major breakthrough in speech recognition, creating a technology that recognizes the words in a conversation as well as a person does.

Youtube Link

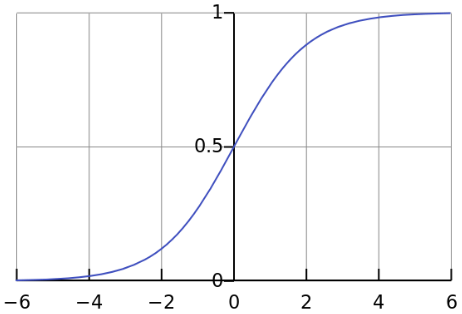
I. deep neural networks crash course
II. Microsoft Cognitive Toolkit (CNTK)
III. authoring neural networks
IV. executing neural networks
   -- GPU execution
   -- optimization
   -- parallelization
V. conclusion

deep neural networks in a single slide

- neurons are simple pattern detectors, measure how well inputs x_j **correlate** with synaptic weights w [Perceptron, Rosenblatt 1957]



example saturation: sigmoid function



[images from Wikipedia]

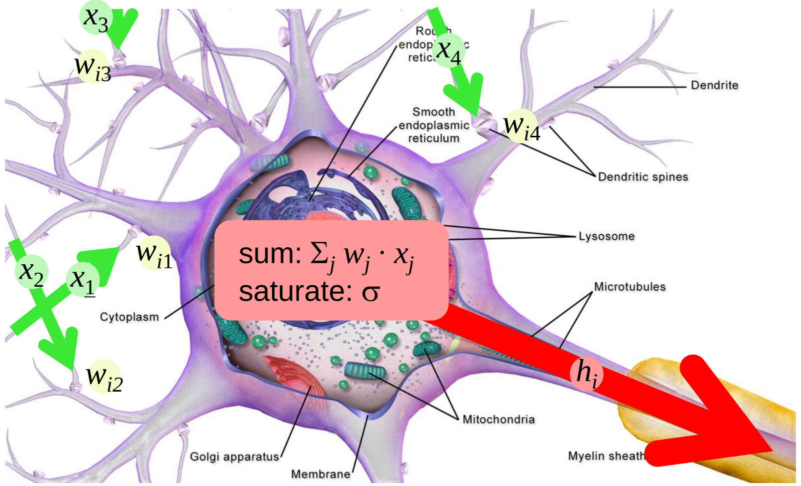
deep neural networks in a single slide

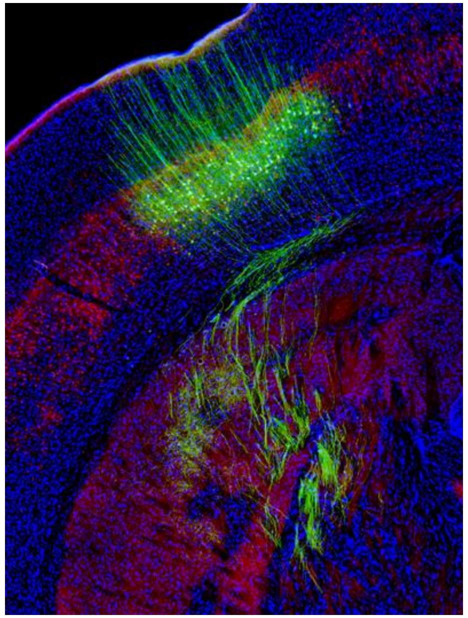
- neurons are simple pattern detectors, measure how well inputs x_j **correlate** with synaptic weights w

$$h_i = \sigma(\Sigma_j \, w_{ij} \cdot x_j + b_i)$$

- operate as **collections**, or vectors

$$h = \sigma(\mathbf{W}\, x + b)$$





correlate
with patterns

$$\mathbf{W}\, x = \begin{pmatrix} w_{11}, w_{12}, ..w_{1N} \\ w_{21}, w_{22}, ..w_{2N} \\ \ldots \\ w_{M1}, w_{M2}, ..w_{MN} \end{pmatrix} \mathrm{x} = \begin{pmatrix} \text{pattern}_1 \\ \text{pattern}_2 \\ \ldots \\ \text{pattern}_N \end{pmatrix} \mathrm{x}$$

e.g. 2000
neurons

deep neural networks in a single slide

- neurons are simple pattern detectors, measure how well inputs x_j **correlate** with synaptic weights w

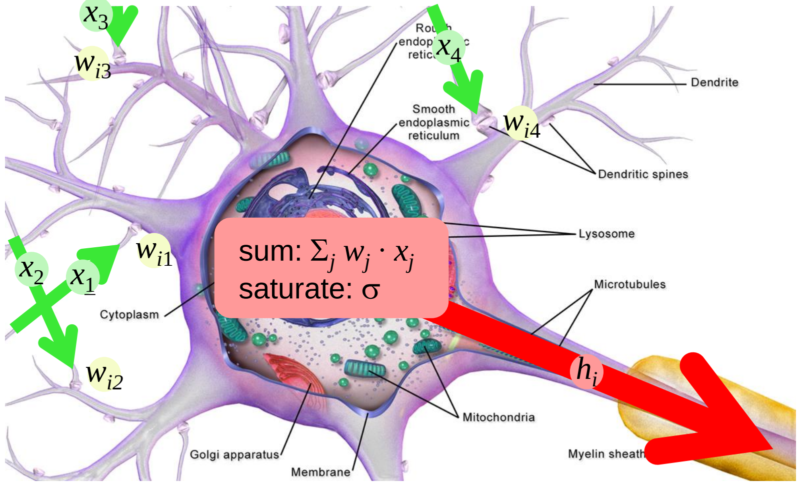
$$h_i = \sigma(\Sigma_j \, w_{ij} \cdot x_j + b_i)$$

- operate as **collections**, or vectors

$$h = \sigma(\mathbf{W} \, x + b)$$

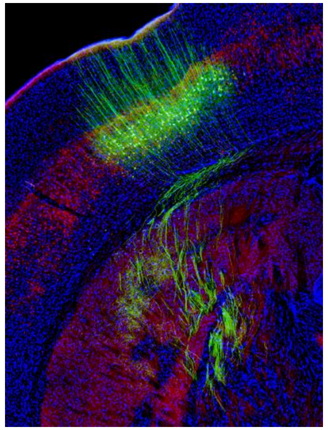
- arranged in **layers** → increasingly abstract representation

$$h^{(1)} = \sigma(\mathbf{W}^{(1)} \, x \quad + b^{(1)})$$
$$h^{(2)} = \sigma(\mathbf{W}^{(2)} \, h^{(1)} + b^{(2)})$$



$$\mathbf{W}^{(1)} = \begin{pmatrix} \boxed{\text{pattern}_1} \\ \boxed{\text{pattern}_2} \\ \dots \\ \boxed{\text{pattern}_N} \end{pmatrix}$$

$$\mathbf{W}^{(2)} = \begin{pmatrix} \boxed{\text{abstract pattern}_1} \\ \boxed{\text{abstract pattern}_2} \\ \dots \\ \boxed{\text{abstract pattern}_N} \end{pmatrix}$$



deep neural networks in a single slide

- neurons are simple pattern detectors, measure how well inputs x_j **correlate** with synaptic weights w

$$h_i = \sigma(\Sigma_j\, w_{ij} \cdot x_j + b_i)$$

- operate as **collections**, or vectors

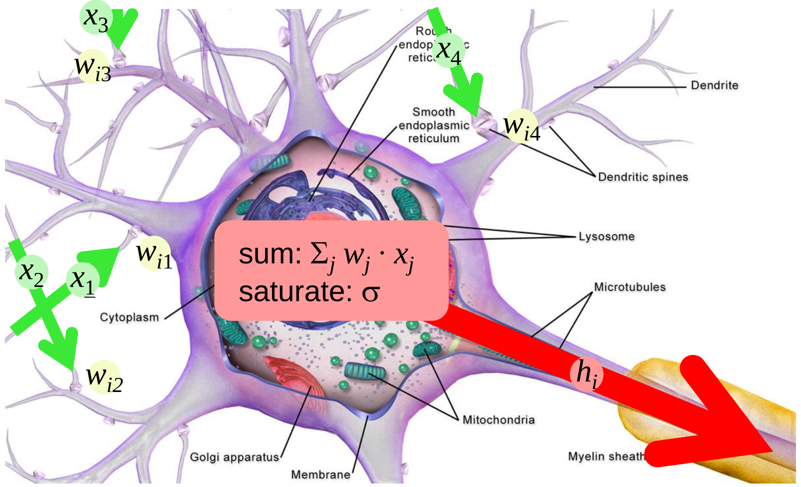
$$h = \sigma(\mathbf{W}\, x + b)$$

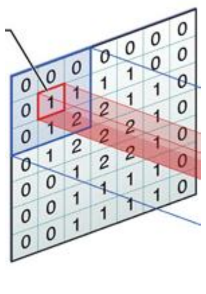
- arranged in **layers** → increasingly abstract representation

$$h^{(1)} = \sigma(\mathbf{W}^{(1)}\, x \quad + b^{(1)})$$
$$h^{(2)} = \sigma(\mathbf{W}^{(2)}\, h^{(1)} + b^{(2)})$$

- connectivity **can be local** (spatial receptive fields)

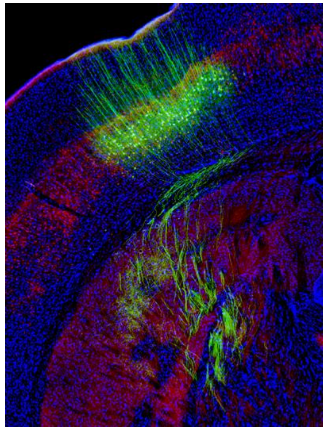
$$h(c,r) = \sigma(\mathbf{W}\, x(c\text{-}\Delta c..c+\Delta c, r\text{-}\Delta r..r+\Delta r) + b)$$





$$\mathbf{W}^{(1)} = \begin{pmatrix} \text{pattern}_1 \\ \text{pattern}_2 \\ \dots \\ \text{pattern}_N \end{pmatrix}$$

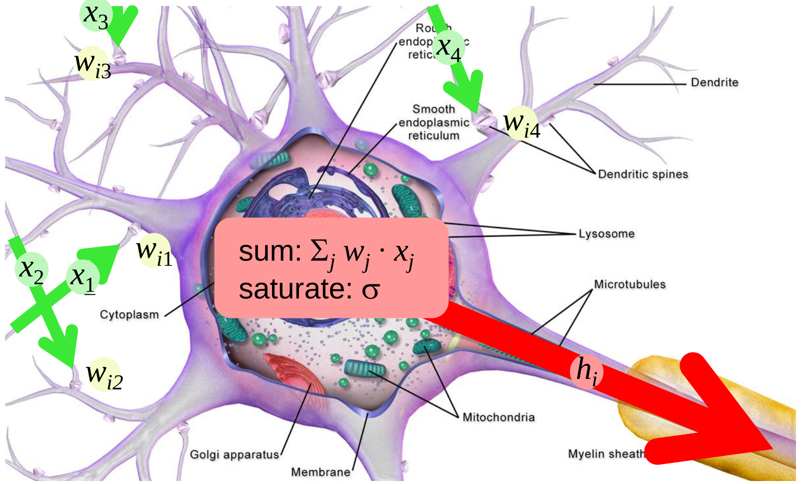
$$\mathbf{W}^{(2)} = \begin{pmatrix} \text{abstract pattern}_1 \\ \text{abstract pattern}_2 \\ \dots \\ \text{abstract pattern}_N \end{pmatrix}$$



deep neural networks in a single slide

- neurons are simple pattern detectors, measure how well inputs x_j **correlate** with synaptic weights w

$$h_i = \sigma(\Sigma_j\, w_{ij} \cdot x_j + b_i)$$

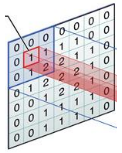


- operate as **collections**, or vectors

$$h = \sigma(\mathbf{W}\, x + b)$$

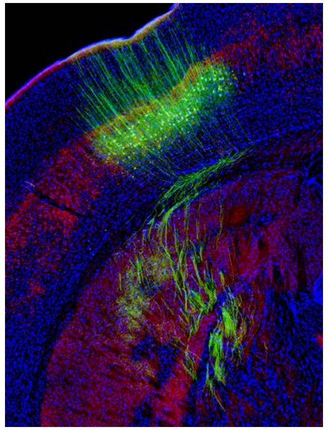
- arranged in **layers** → increasingly abstract representation

$$h^{(1)} = \sigma(\mathbf{W}^{(1)}\, x \quad + b^{(1)})$$
$$h^{(2)} = \sigma(\mathbf{W}^{(2)}\, h^{(1)} + b^{(2)})$$



$$\mathbf{W}^{(1)} = \begin{pmatrix} \text{pattern}_1 \\ \text{pattern}_2 \\ \dots \\ \text{pattern}_N \end{pmatrix}$$

$$\mathbf{W}^{(2)} = \begin{pmatrix} \text{abstract pattern}_1 \\ \text{abstract pattern}_2 \\ \dots \\ \text{abstract pattern}_N \end{pmatrix}$$

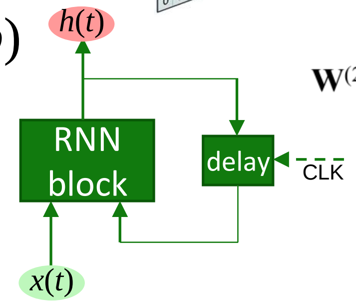


- connectivity **can be local** (spatial receptive fields)

$$h(c,r) = \sigma(\mathbf{W}\, x(c\text{-}\Delta c..c+\Delta c, r\text{-}\Delta r..r+\Delta r) + b)$$

- can form **feedback loops**

$$h(t) = \sigma(\mathbf{W}\, x(t) + \mathbf{R}\, h(t\text{-}1) + b)$$



deep neural networks in a single slide

- neurons are simple pattern detectors, measure how well inputs x_j **correlate** with synaptic weights w

neural $\qquad h_i = \sigma(\Sigma_j\, w_{ij} \cdot x_j + b_i)$

- operate as **collections**, or vectors

network $\qquad$ fully connected $\qquad h = \sigma(\mathbf{W}\, x + b)$

- arranged in **layers** → increasingly abstract representation

deep

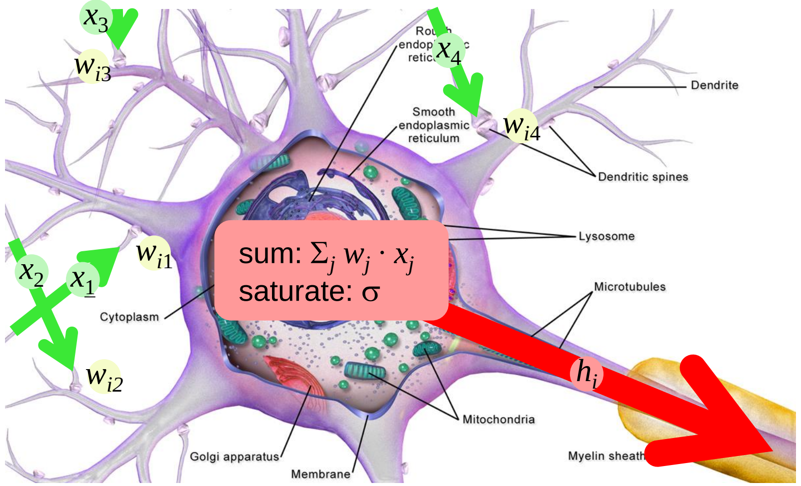
$$h^{(1)} = \sigma(\mathbf{W}^{(1)}\, x \quad + b^{(1)})$$
$$h^{(2)} = \sigma(\mathbf{W}^{(2)}\, h^{(1)} + b^{(2)})$$

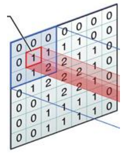
- connectivity **can be local** (spatial receptive fields)

convolutional $\qquad h(c,r) = \sigma(\mathbf{W}\, x(c\text{-}\Delta c..c+\Delta c, r\text{-}\Delta r..r+\Delta r) + b)$

- can form **feedback loops**

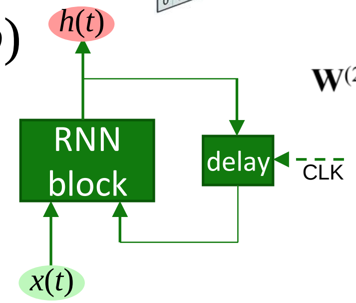
recurrent $\qquad h(t) = \sigma(\mathbf{W}\, x(t) + \mathbf{R}\, h(t\text{-}1) + b)$

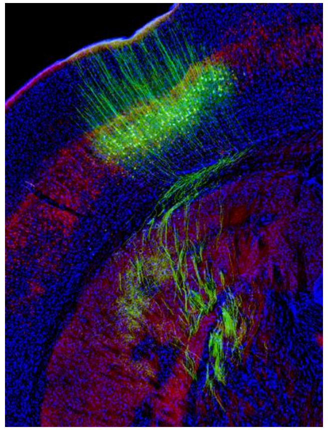




$$\mathbf{W}^{(1)} = \begin{pmatrix} \text{pattern}_1 \\ \text{pattern}_2 \\ \cdots \\ \text{pattern}_N \end{pmatrix}$$

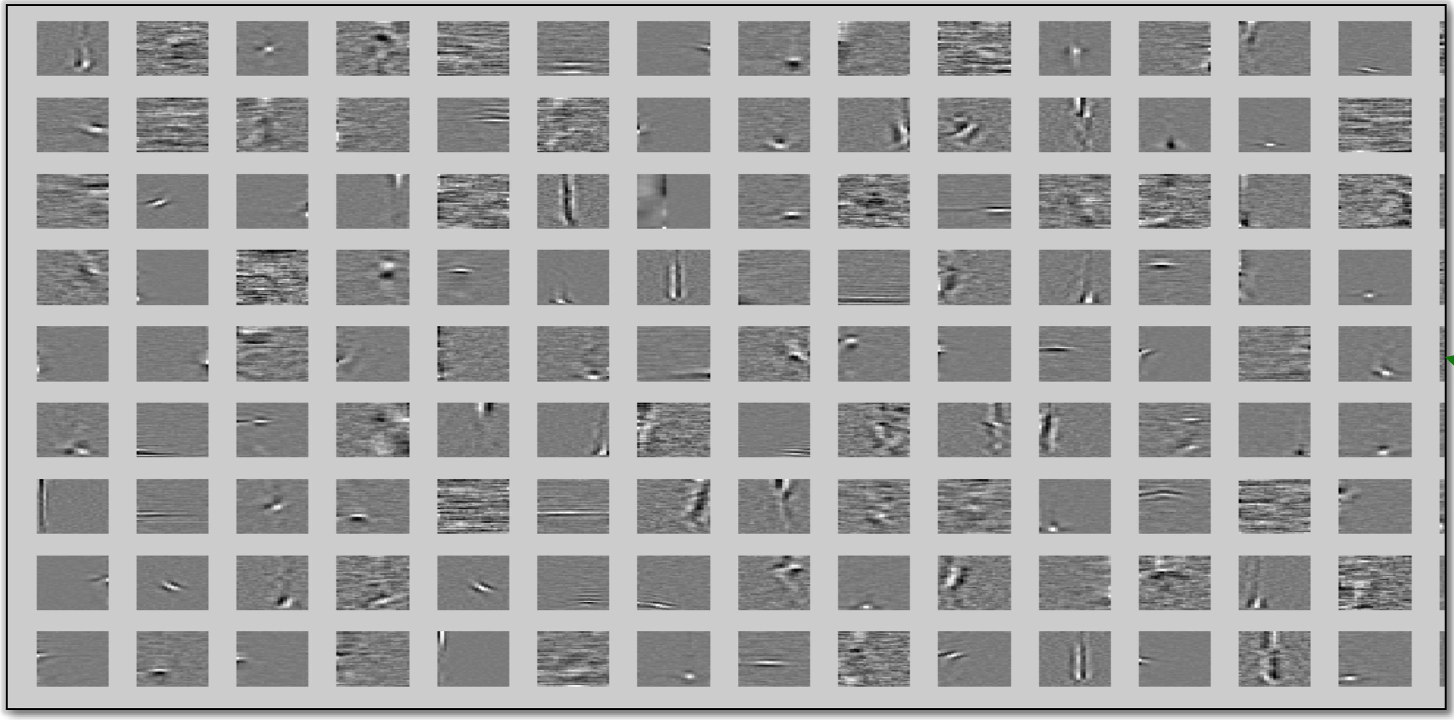
$$\mathbf{W}^{(2)} = \begin{pmatrix} \text{abstract pattern}_1 \\ \text{abstract pattern}_2 \\ \cdots \\ \text{abstract pattern}_N \end{pmatrix}$$

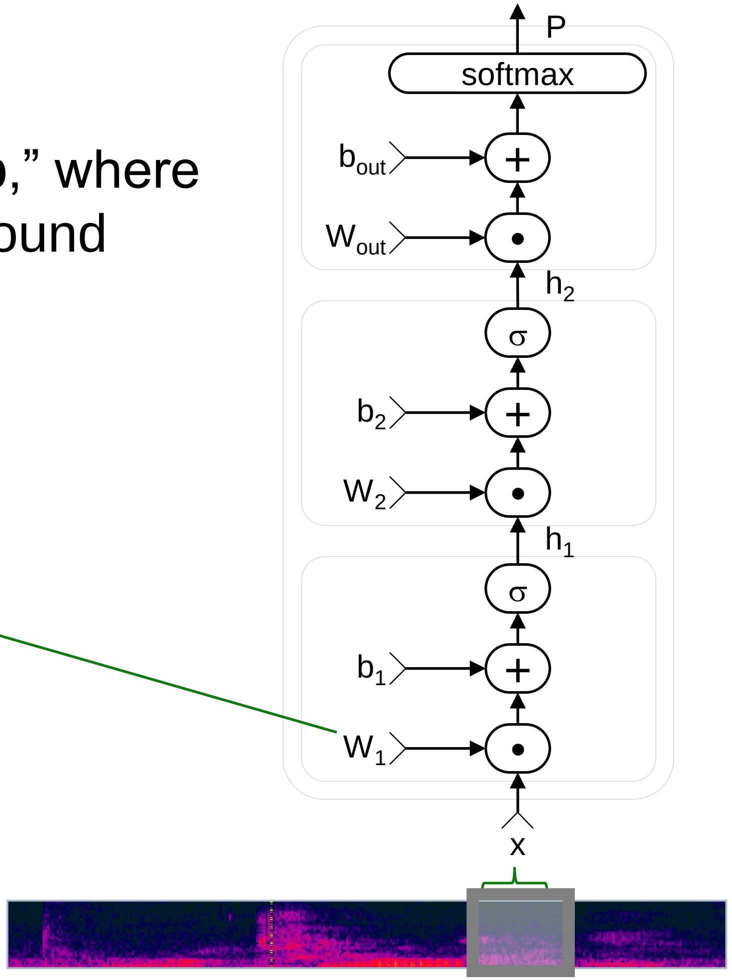




fully connected (FCN), convolutional (CNN), recurrent (RNN)

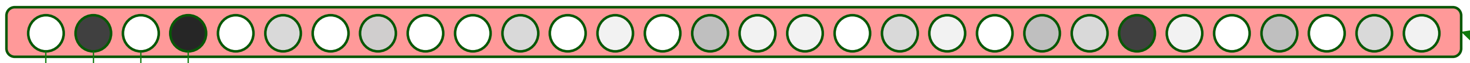
- fully connected (FCN) $h = \sigma(\mathbf{W}\, x + b)$
  - describes objects through probabilities of "**class membership**," where the N classes overlap and are whatever the training process found



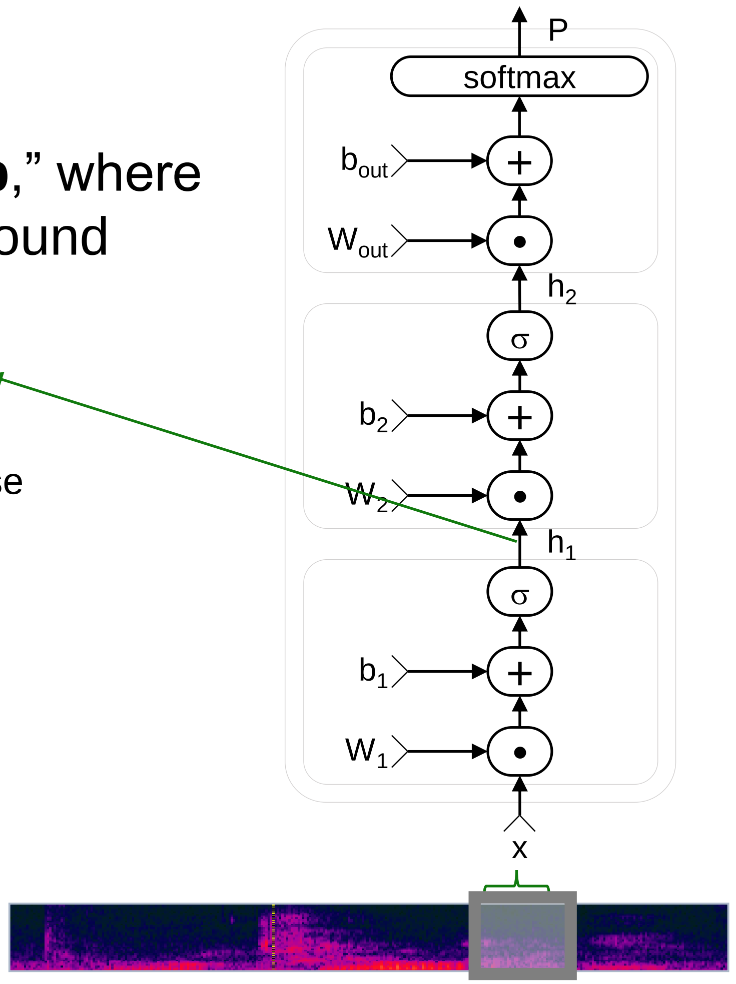


fully connected (FCN), convolutional (CNN), recurrent (RNN)

- fully connected (FCN) $h = \sigma(\mathbf{W}\, x + b)$
  - describes objects through probabilities of “**class membership**,” where the N classes overlap and are whatever the training process found

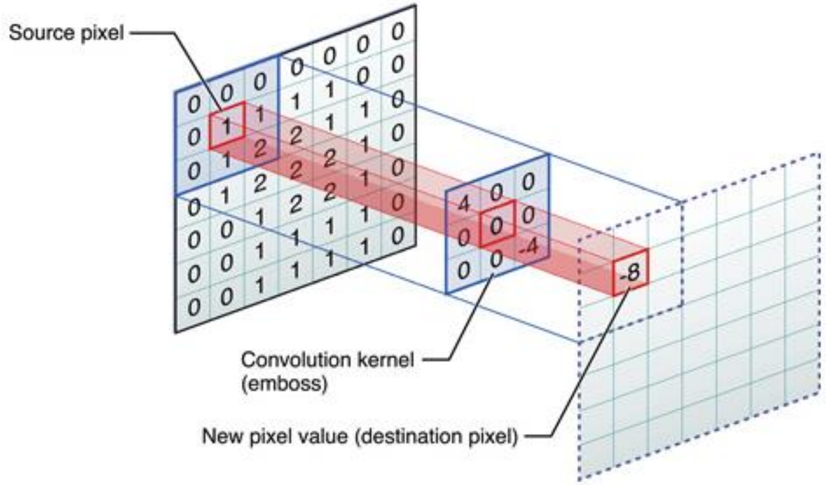


- is something with a sharp increase of higher-frequency broadband noise
- is something with a broad spectral peak at 820 Hz
- is something with broadly low energy
- is something with a spectral peak at 1 kHz moving up
- …



fully connected (FCN), convolutional (CNN), recurrent (RNN)

- fully connected (FCN) $h = \sigma(\mathbf{W}\ x + b)$
  - describes objects through probabilities of "**class membership.**"
- convolutional (CNN) $h(c,r) = \sigma(\mathbf{W}\ x(c\text{-}\Delta c..c+\Delta c, r\text{-}\Delta r..r+\Delta r) + b)$
  - repeatedly applies a little FCN over images or other **repetitive structures**



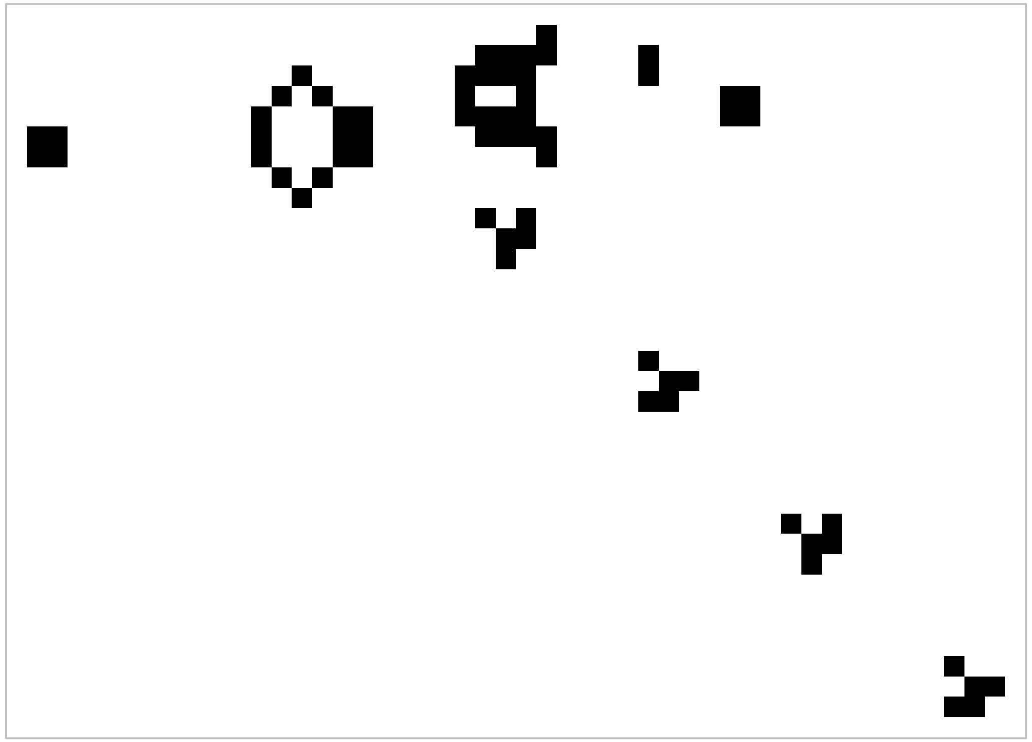
fully connected (FCN), convolutional (CNN), recurrent (RNN)

- fully connected (FCN) $h = \sigma(\mathbf{W}\, x + b)$
  - describes objects through probabilities of "**class membership.**"
- convolutional (CNN) $h(c,r) = \sigma(\mathbf{W}\, x(c\text{-}\Delta c..c+\Delta c, r\text{-}\Delta r..r+\Delta r) + b)$
  - repeatedly applies a little FCN over images or other **repetitive structures**
- recurrent (RNN) $h(t) = \sigma(\mathbf{W}\, x(t) + \mathbf{R}\, h(t\text{-}1) + b)$
  - repeatedly applies a FCN over a **sequence**, using its **own previous output**

fully connected (FCN), convolutional (CNN), recurrent (RNN)

- fully connected (FCN) `map`
  - describes objects through probabilities of "**class membership.**"
- convolutional (CNN) `windowed >> map` FIR filter
  - repeatedly applies a little FCN over images or other **repetitive structures**
- recurrent (RNN) `scanl, foldl, unfold` IIR filter
  - repeatedly applies a FCN over a **sequence**, using its **own previous output**
- most interesting applications are composite functions of these, e.g.:
  - **translation**: RNN encoder (fold) + RNN decoder (unfold) + beam search [Sutskever *et al.*, 2014]
  - **image captioning**: CNN stack + FCN classifier + text generator [Fang *et al.*, 2015]
  - **Generative Adversarial Nets**: inverted CNN stack trying to fool a CNN stack [Goodfellow *et al.*, 2015]
  - **Neural Turing machines**: multiple RNNs learn algorithms from data [Graves *et al.*, 2015]

illustration: Conway’s game of life, hand-crafted



[https://en.wikipedia.org/wiki/Conway's_Game_of_Life]

illustration: Conway's game of life, hand-crafted

$$\begin{pmatrix} -1,-1,-1,-1,-1,-1,-1,-1,-1 \\ -1,-1,-1,-1,-1,-1,-1,-1,+1 \\ -1,-1,-1,-1,-1,-1,-1,+1,-1 \\ \ldots \\ +1,+1,+1,+1,+1,+1,+1,+1,+1 \end{pmatrix} \begin{pmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ +1 \\ +1 \\ -1 \\ +1 \end{pmatrix}$$

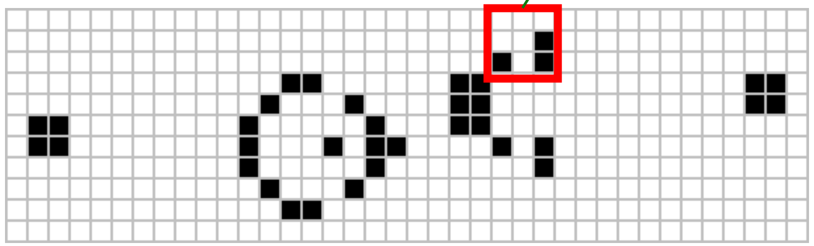
- create a 2-layer FCN that implements the propagation rules
  - Layer 1:
    - represent the 9 bits as a 9-dim vector of {-1, +1}
    - $\mathbf{W}^{(1)}$ enumerates all 512 input patterns, threshold for "all match"

$$h^{(1)} = r(\mathbf{W}^{(1)}\, x + b) \;;\; b = -8;\; r(z) = \max(z,0)$$

  - Layer 2:
    - $\mathbf{W}^{(2)}$ enumerates output values of truth table: (0, 0, 0, 1, ..., 1, ..., 0)

$$h^{(2)} = \mathrm{s}(\mathbf{W}^{(2)}\, h^{(1)} + \mathrm{b}) \;;\; \mathrm{b} = -\,0.5 \;;\; \mathrm{s}(z) = \mathrm{sgn}(z)$$

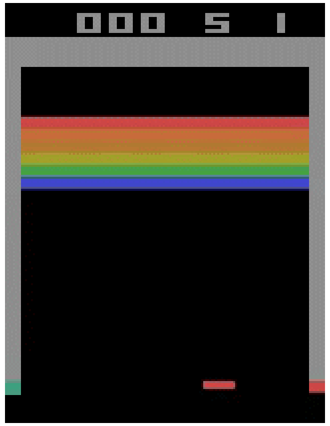
- apply independently to every pixel position to perform one step → CNN
- unfold this over time → RNN
- game of life is a universal Turing machine
  → **so are deep recurrent networks**



[https://en.wikipedia.org/wiki/Conway's_Game_of_Life]

training deep neural networks with SGD

- training
  - find weight parameters ($\mathbf{W}^{(n)}, b^{(n)}$) as to match some **criterion function**
  - **supervised learning** → **classify** an input
  - **unsupervised learning** → **discover** hidden structure in data
  - **reinforcement learning** → interact with an environment to **maximize reward**



CNTK / OpenAI Gym
[Morgan Funtowicz]

- stochastic gradient descent (SGD)
  - feed input sample, compare to desired output
  - iteratively take a step in the direction of the **gradient** of the criterion function w.r.t. a weight parameter
  - gradients are computed through **automatic differentiation**
- SGD training is VERY expensive
  - speech: 100M MACs/sample, 3.6B samples, 3 passes, fw+bw → **10^{18} MACs**
  - image: 4.4B MACs/sample, 1.2M samples, 120 passes , fw+bw → **10^{19} MACs**
  - Titan X GPU (3840 CUDA cores): peak 3.5 10^{12} MACs/s → 1+ weeks

deep-learning toolkits must address two questions:

- **how to author neural networks?** ← user's job
- **how to execute them efficiently? (training/test)** ← *tool*'s job*!!*

I. deep neural networks crash course
II. Microsoft Cognitive Toolkit (CNTK)
III. authoring neural networks
IV. executing neural networks
    -- GPU execution
    -- optimization
    -- parallelization
V. conclusion

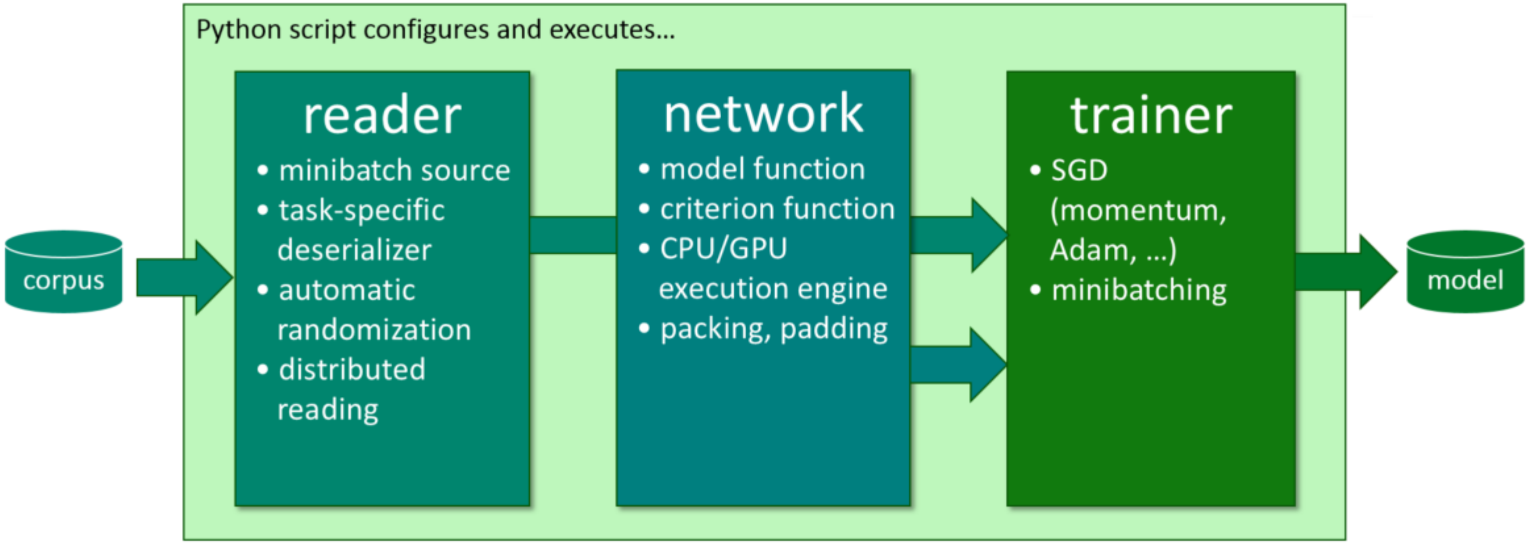
Microsoft Cognitive Toolkit, CNTK

Microsoft Cognitive Toolkit
aka.ms/CognitiveToolkit
Microsoft

- CNTK is a library for deep neural networks
  - model definition
  - scalable training
  - efficient I/O
- makes it easy to author, train, and use neural networks
  - think “what” not “how”
  - focus on composability
- functional-style EDSL on top of Python on top of C++ API/library
- open source since 2015 https://github.com/Microsoft/CNTK Star 9,456 Fork 2,289
  - created by Microsoft Speech researchers (Dong Yu et al.) in 2012, “Computational Network Toolkit”
  - contributions from MS product groups and external (e.g. MIT, Stanford), development is visible on Github
  - Linux, Windows, docker, cudnn5, CUDA 8

Microsoft Cognitive Toolkit, CNTK

```
from cntk import *

# reader
def create_reader(path, is_training):
    ...

# network
def create_model_function():
    ...
def create_criterion_function(model):
    ...

# trainer (and evaluator)
def train(reader, model):
    ...
def evaluate(reader, model):
    ...

# main function
model = create_model_function()

reader = create_reader(..., is_training=True)
train(reader, model)

reader = create_reader(..., is_training=False)
evaluate(reader, model)
```



Microsoft Cognitive Toolkit, CNTK

TABLE 7. COMPARATIVE EXPERIMENT RESULTS (TIME PER MINI-BATCH IN SECOND)

		Desktop CPU (Threads used)				Server CPU (Threads used)						Single GPU		
		1	2	4	8	1	2	4	8	16	32	G980	G1080	K80
FCN-S	Caffe	1.324	0.790	**0.578**	-	1.355	0.997	0.745	**0.573**	0.608	1.130	0.041	**0.030**	0.071
	CNTK	1.227	0.660	**0.435**	-	1.340	0.909	0.634	0.488	**0.441**	1.000	0.045	**0.033**	0.074
	TF	7.062	4.789	2.648	**1.938**	9.571	6.569	3.399	1.710	0.946	**0.630**	0.060	**0.048**	0.109
	MXNet	4.621	2.607	2.162	**1.831**	5.824	3.356	2.395	2.040	**1.945**	2.670	-	**0.106**	0.216
	Torch	1.329	0.710	**0.423**	-	1.279	1.131	0.595	0.433	**0.382**	1.034	0.040	**0.031**	0.070
AlexNet-S	Caffe	1.606	0.999	**0.719**	-	1.533	1.045	**0.797**	0.850	0.903	1.124	0.034	**0.021**	0.073
	CNTK	3.761	1.974	**1.276**	-	3.852	2.600	1.567	1.347	**1.168**	1.579	0.045	**0.032**	0.091
	TF	6.525	2.936	1.749	**1.535**	5.741	4.216	2.202	1.160	**0.701**	0.962	0.059	**0.042**	0.130
	MXNet	2.977	2.340	2.250	**2.163**	3.518	3.203	2.926	2.828	**2.827**	2.887	0.020	**0.014**	0.042
	Torch	4.645	2.429	**1.424**	-	4.336	2.468	1.543	1.248	**1.090**	1.214	0.033	**0.023**	0.070
RenNet-50	Caffe	11.554	7.671	**5.652**	-	10.643	8.600	6.723	**6.019**	6.654	8.220	-	**0.254**	0.766
	CNTK	-	-	-	-	-	-	-	-	-	-	0.240	**0.168**	0.638
	TF	23.905	16.435	10.206	**7.816**	29.960	21.846	11.512	6.294	**4.130**	4.351	0.327	**0.227**	0.702
	MXNet	**14.035**	16.053	16.162	15.000	**17.910**	19.277	19.123	18.898	19.048	19.355	0.059	**0.041**	0.132
	Torch	13.178	7.500	**4.736**	4.948	12.807	8.391	5.471	4.164	**3.683**	4.422	0.208	**0.144**	0.523
FCN-R	Caffe	2.476	1.499	**1.149**	-	2.282	1.748	1.403	1.211	1.127	**1.127**	0.025	**0.017**	0.055
	CNTK	1.845	0.970	0.661	**0.571**	1.592	0.857	0.501	0.323	**0.252**	0.280	0.025	**0.017**	0.053
	TF	2.647	1.913	1.157	**0.919**	3.410	2.541	1.297	0.661	0.361	**0.325**	0.033	**0.020**	0.063
	MXNet	1.914	1.072	0.719	**0.702**	1.609	1.065	0.731	0.534	0.451	**0.447**	0.029	**0.019**	0.060
	Torch	1.670	0.926	**0.565**	0.611	1.379	0.915	0.662	0.440	0.402	**0.366**	0.025	**0.016**	0.051
AlexNet-R	Caffe	3.558	2.587	**2.157**	2.963	4.270	3.514	3.381	**3.364**	4.139	4.930	0.041	**0.027**	0.137
	CNTK	9.956	7.263	**5.519**	6.015	9.381	6.078	4.984	**4.765**	6.256	6.199	0.045	**0.031**	0.108
	TF	4.535	3.225	1.911	**1.565**	6.124	4.229	2.200	1.396	1.036	**0.971**	**0.227**	0.317	0.385
	MXNet	13.401	12.305	12.278	**11.950**	17.994	17.128	16.764	**16.471**	17.471	17.770	0.060	**0.032**	0.122
	Torch	5.352	3.866	**3.162**	3.259	6.554	5.288	4.365	**3.940**	4.157	4.165	0.069	**0.043**	0.141
RenNet-56	Caffe	6.741	5.451	**4.989**	6.691	7.513	**6.119**	6.232	6.689	7.313	9.302	-	**0.116**	0.378
	CNTK	-	-	-	-	-	-	-	-	-	-	0.206	**0.138**	0.562
	TF	-	-	-	-	-	-	-	-	-	-	0.225	**0.152**	0.523
	MXNet	34.409	31.255	**30.069**	31.388	44.878	43.775	**42.299**	42.965	43.854	44.367	0.105	**0.074**	0.270
	Torch	5.758	3.222	**2.368**	2.475	8.691	4.965	3.040	**2.560**	2.575	2.811	0.150	**0.101**	0.301
LSTM	Caffe	-	-	-	-	-	-	-	-	-	-	-	-	-
	CNTK	0.186	0.120	**0.090**	0.118	0.211	0.139	0.117	0.114	**0.114**	0.198	0.018	**0.017**	0.043
	TF	4.662	3.385	1.935	**1.532**	6.449	4.351	2.238	1.183	0.702	**0.598**	0.133	**0.065**	0.140
	MXNet	-	-	-	-	-	-	-	-	-	-	0.089	**0.079**	0.149
	Torch	6.921	3.831	**2.682**	3.127	7.471	4.641	3.580	**3.260**	5.148	5.851	0.399	**0.324**	0.560

[“Benchmarking State-of-the-Art Deep Learning Software Tools,” HKBU, https://arxiv.org/pdf/1608.07249v6.pd]

I. deep neural networks crash course
II. Microsoft Cognitive Toolkit (CNTK)
III. authoring neural networks
IV. executing neural networks
   -- GPU execution
   -- optimization
   -- parallelization
V. conclusion

expression-graph representation of neural networks

example: 2-hidden layer feed-forward NN

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

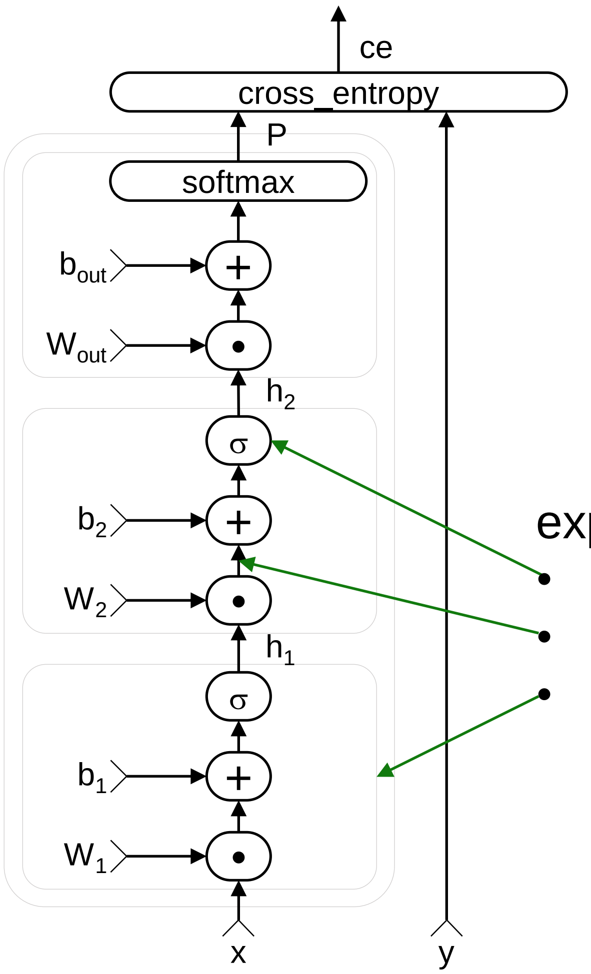
```
h1 = sigmoid (x  @ W1   + b1)
h2 = sigmoid (h1 @ W2   + b2)
P  = softmax (h2 @ Wout + bout)
```

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)
```

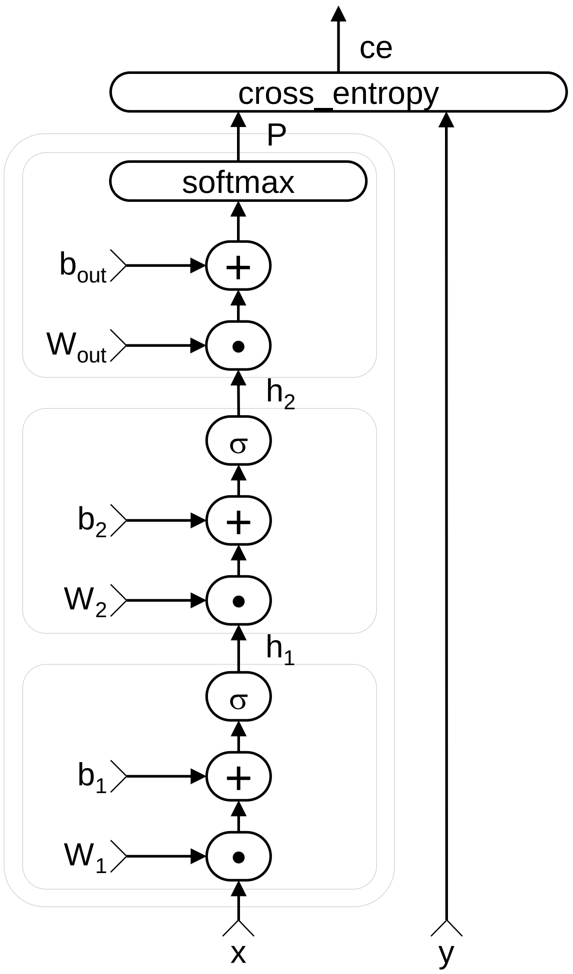
expression-graph representation of neural networks



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)
```

expression-graph representation of neural networks



why the expression-graph detour?

- automatic differentiation*!!*
  - chain rule: $\partial\mathcal{F} / \partial in = \partial\mathcal{F} / \partial out \cdot \partial out / \partial in$
  - run graph backwards

→ **“back propagation”**

expression graphs vs. imperative Python code

- **Theano, TensorFlow**:
  - **expression graph is user surface**; user builds the graph
  - Python functions are just code-generation macros; one "**programs at a distance**" [John Launchbury]
  - referential transparency problem, e.g. what does this really mean? [Bruno Bozza]

```
y = tf.contrib.layers.fully_connected(x, num_outputs=512, scope=variable_scope)
```

→ graph is too low an abstraction level, implementation detail

- **Chainer, DyNet**:
  - **imperative computation**, also builds a **graph for back prop only**
  - re-done for each input → **can implement arbitrary algorithms** in any coding style
  - but **control flow opaque** to toolkit → parallelization (batching) left to user

→ imperative execution is too high an abstraction level for optimization

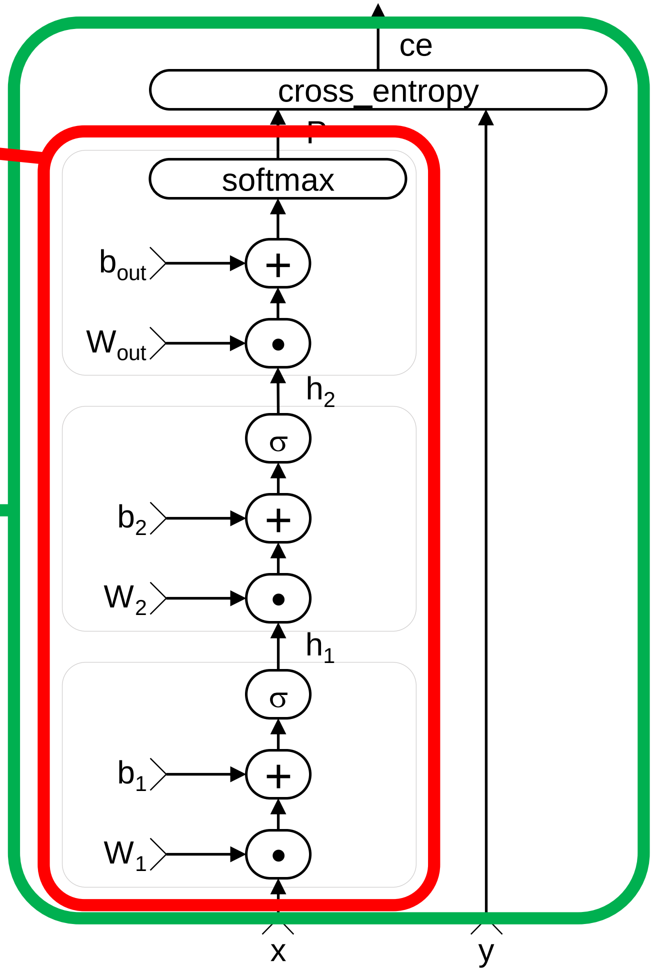
authoring networks as functions

- **CNTK** model: neural networks are functions, and reasoned about as such
  - pure functions (cannot modify state)
  - with “special powers”:
    - can compute a gradient w.r.t. any of its nodes
    - external deity can update model parameters (but think creating a new function from an old one)
- user specifies network as function objects:
  - formula as a Python function (low level, e.g. LSTM)
  - function composition of smaller sub-networks (layering)
  - higher-order functions (equiv. of scan, fold, unfold)
  - model parameters owned (closed over) by the function objects → solves referential transparency
- “compiled” into the static execution graph under the hood

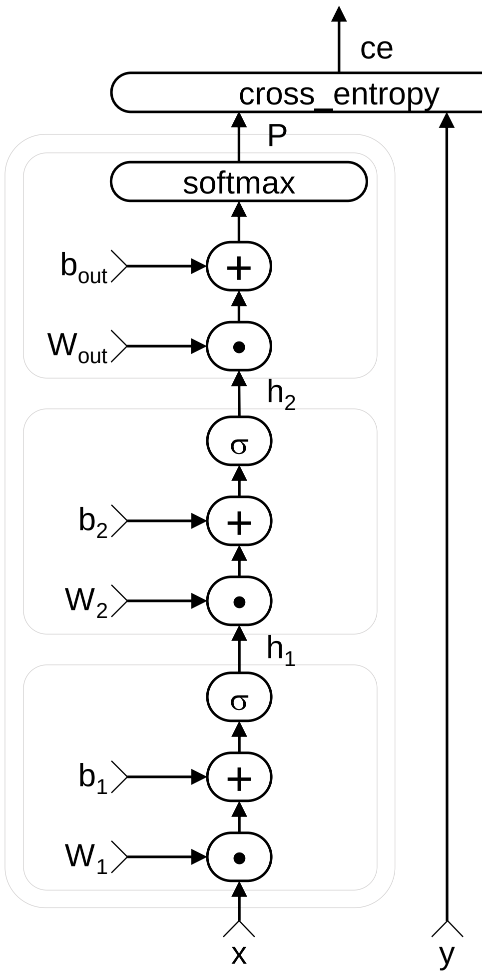
authoring networks as functions

- “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

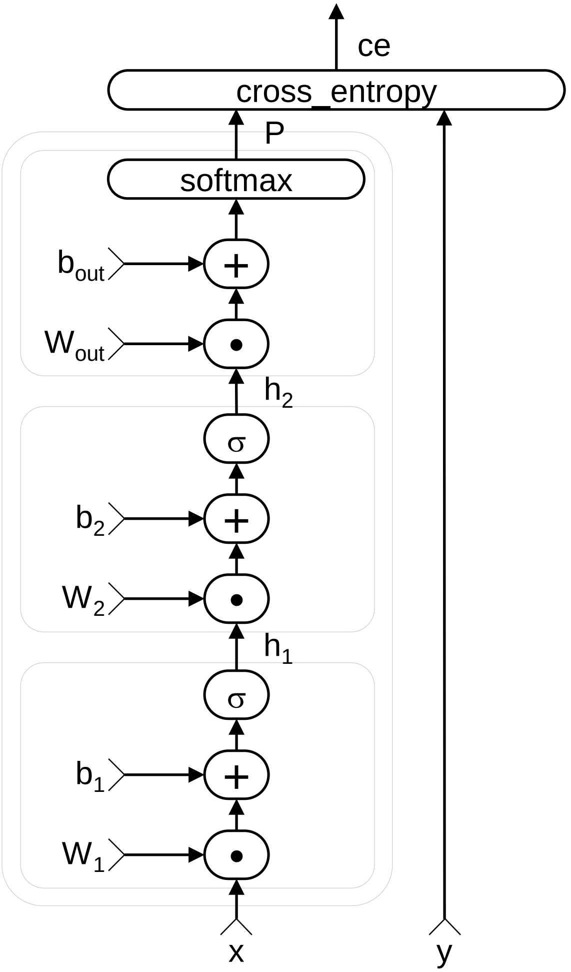


authoring networks as functions



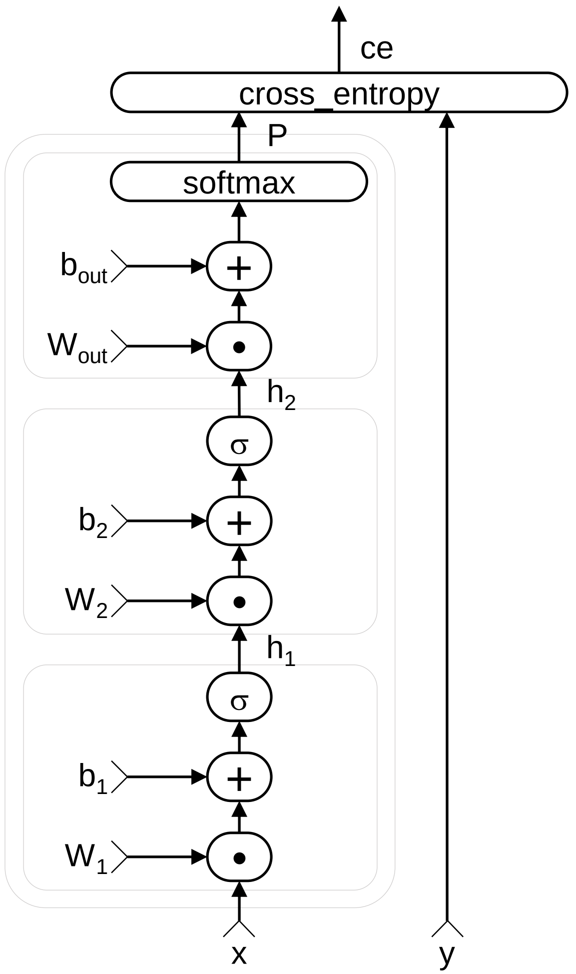

```
# --- graph building ---
M = 40 ; H = 512 ; J = 9000  # feat/hid/out dim
# define learnable parameters
W1   = Parameter((M,H)); b1   = Parameter(H)
W2   = Parameter((H,H)); b2   = Parameter(H)
Wout = Parameter((H,J)); bout = Parameter(J)
# build the graph
x = Input(M) ; y = Input(J)  # feat/labels
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)
```

authoring networks as functions




```
M = 40 ; H = 512 ; J = 9000  # feat/hid/out dim
# function composition
model = (Dense(H, activation=sigmoid) >>
         Dense(H, activation=sigmoid) >>
         Dense(J, activation=softmax))
# criterion as function, invokes model function
@Function
def crit(x: Tensor[M], y: Tensor[J]):
    P = model(x)
    return cross_entropy(P, y)
# function is passed to trainer
tr = Trainer(crit, Learner(model.parameters), …)
```

authoring networks as functions

enables higher-order functions:

- forward composition:

```
model = Dense(H, activation=sigmoid) >> Dense(H, activation=sigmoid) >> Dense(J)
```

- recurrence (scanl/foldl):

```
model = (Embedding(emb_dim, name='embed') >>
         Recurrence(RNNUnit(hidden_dim)) >>  # == scanl over recurrent block
         Dense(num_labels, name='out_projection'))
```

- unfold:

```
model = UnfoldFrom(lambda history: s2smodel(history, input) >> hardmax,
                   until_predicate=lambda w: w[...,sentence_end_index],
                   length_increase=length_increase)
output = model(START_SYMBOL)
```

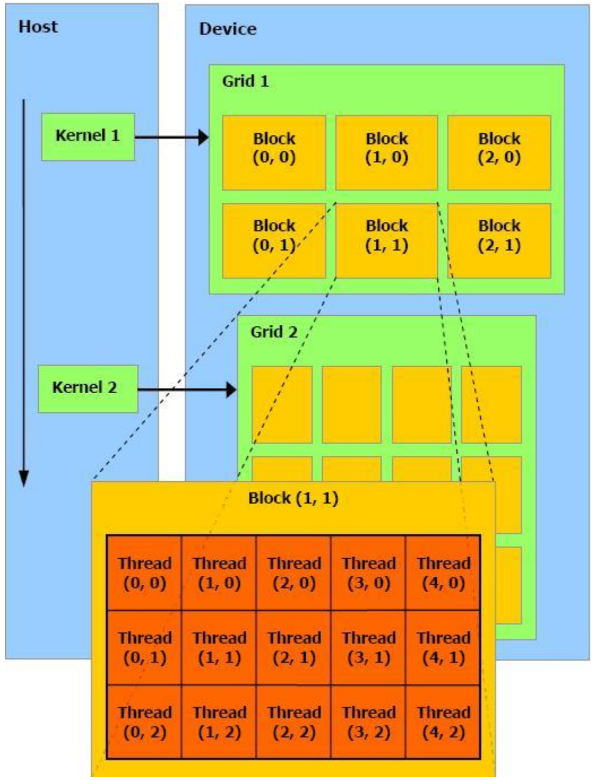
Layers lib: full list of layers/blocks

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

I. deep neural networks crash course
II. Microsoft Cognitive Toolkit (CNTK)
III. authoring neural networks
IV. executing neural networks
   -- GPU execution
   -- optimization
   -- parallelization
V. conclusion

high performance with GPUs

- after being stuck for decades, GPUs made NN research and experimentation productive
- must turn DNNs into **parallel programs**
- two main priorities in GPU computing:
  1. make sure all CUDA cores are always busy
     - Titan X: 3072 parallel processors, so single threaded code would only get 1/3072 = 0.03% of peak performance
  2. read from GPU RAM as little as possible
     - reading a float and using it once = 4 bytes for 1 operation = 288 GB/sec * ¼ GFLOP/GB = 72 GFLOP/sec peak = 72/7000 **= 1% utilization**
     - even if you use all CUDA cores!



[Jacob Devlin, NLPCC 2016 Tutorial]

parallel programs through minibatching

- **minibatching :=** batch N samples, e.g. N=256; execute in lockstep
  - turns N matrix-vector products into one matrix-matrix product
  - cuBLAS gets close to peak performance
  - benefits element-wise ops and reductions, too
  - key enabler
- limits:
  - convergence
  - cross-sample dependencies (recurrent nets)
  - memory size
- difficult to get right
  → CNTK makes batching fully transparent

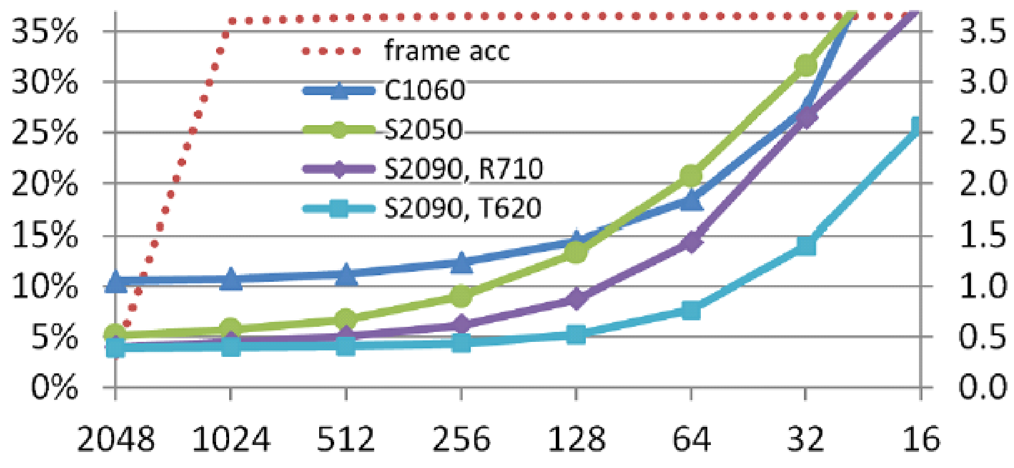


Figure 1: *Relative runtime for different minibatch sizes and GPU/server model types, and corresponding frame accuracy measured after seeing 12 hours of data.*[7]

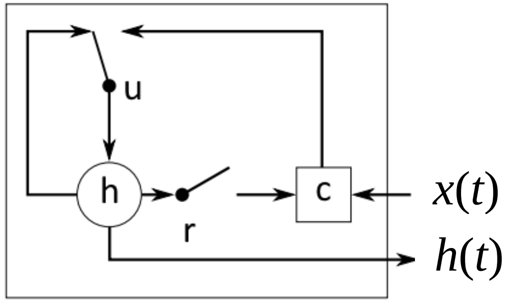
improved parallelism through operation fusion

[Jacob Devlin, Efficient Training and Deployment of Large Scale Deep Learning Systems for NLP, NLPCC 2016]

- example “gated recurrent unit” (GRU), a popular recurrent unit

$$u(t) = \sigma(\mathbf{W}_u\, x(t) + \mathbf{R}_u\, h(t\text{-}1) + b_u)$$
$$r(t) = \sigma(\mathbf{W}_r\, x(t) + \mathbf{R}_r\, h(t\text{-}1) + b_r)$$
$$c(t) = \tanh(\mathbf{W}_c\, x(t) + r_i \odot (\mathbf{R}_c\, h(t\text{-}1)) + b_c)$$
$$h(t) = (1\text{-}u(t)) \odot h(t\text{-}1) + u(t) \odot c(t)$$

6 matrix-vector multiplications
14 element-wise functions



[Chung et al., 2015]

- **operation fusion**:
  - combine matrix products: stack ($\mathbf{R}_u$, $\mathbf{R}_r$, $\mathbf{R}_c$) and ($\mathbf{W}_u$, $\mathbf{W}_r$, $\mathbf{W}_c$) → better core use
  - pull products with $x(t)$ outside the loop → single launch
  - combine element-wise operations → avoid memory round trips

Model	$N = 64, H = 512$ [GFLOP/s]	$N = 128, H = 512$ [GFLOP/s]	$N = 128, H = 1024$ [GFLOP/s]
Basic Implementation	350	700	1,550
+ GEMM Weight Fusion	650	1,200	2,900
+ GEMM Timestep Fusion	800	1,450	3,150
+ Element-Wise Fusion	1,600	2,600	4,250

NVIDIA Titan X
Peak: 5,500 GFLOP/s
N = batch size
H = hidden dim

I. deep neural networks crash course
II. Microsoft Cognitive Toolkit (CNTK)
III. authoring neural networks
IV. executing neural networks
   -- GPU execution
   -- optimization
   -- parallelization
V. conclusion

symbolic loops over sequential data

extend our example to a recurrent network (RNN)

$$h_1 = \sigma(\mathbf{W}_1\, x + b_1)$$

$$h_2 = \sigma(\mathbf{W}_2\, h_1 + b_2)$$

$$P = \mathrm{softmax}(\mathbf{W}_{\mathrm{out}}\, h_2 + b_{\mathrm{out}})$$

$$ce = L^{\mathrm{T}} \log P$$

$$\textstyle\sum_{\mathrm{corpus}} ce = \max$$

symbolic loops over sequential data

extend our example to a recurrent network (RNN)

$$h_1(t) = \sigma(\mathbf{W}_1\, x(t) + \mathbf{R}_1\, h_1(t\text{-}1) + b_1)$$

$$h_2(t) = \sigma(\mathbf{W}_2\, h_1(t) + \mathbf{R}_2\, h_2(t\text{-}1) + b_2)$$

$$P(t) = \mathrm{softmax}(\mathbf{W}_{\mathrm{out}}\, h_2(t) + b_{\mathrm{out}})$$

$$ce(t) = L^{\mathrm{T}}(t) \log P(t)$$

$$\sum\nolimits_{\mathrm{corpus}} ce(t) = \max$$

```
h1 = sigmoid(x  @ W1 + past_value(h1) @ R1 + b1)
h2 = sigmoid(h1 @ W2 + past_value(h2) @ R2 + b2)
P  = softmax(h2 @ Wout + bout)
ce = cross_entropy(P, L)
```

symbolic loops over sequential data

extend our example to a recurrent network (RNN)

$$h_1(t) = \sigma(\mathbf{W}_1\, x(t) + \mathbf{R}_1\, h_1(t\text{-}1) + b_1)$$
$$h_2(t) = \sigma(\mathbf{W}_2\, h_1(t) + \mathbf{R}_2\, h_2(t\text{-}1) + b_2)$$
$$P(t) = \mathrm{softmax}(\mathbf{W}_{\mathrm{out}}\, h_2(t) + b_{\mathrm{out}})$$
$$ce(t) = L^{\mathrm{T}}(t) \log P(t)$$
$$\textstyle\sum_{\mathrm{corpus}} ce(t) = \max$$

```
h1 = sigmoid(x  @ W1 + past_value(h1) @ R1 + b1)
h2 = sigmoid(h1 @ W2 + past_value(h2) @ R2 + b2)
P  = softmax(h2 @ Wout + bout)
ce = cross_entropy(P, L)
```

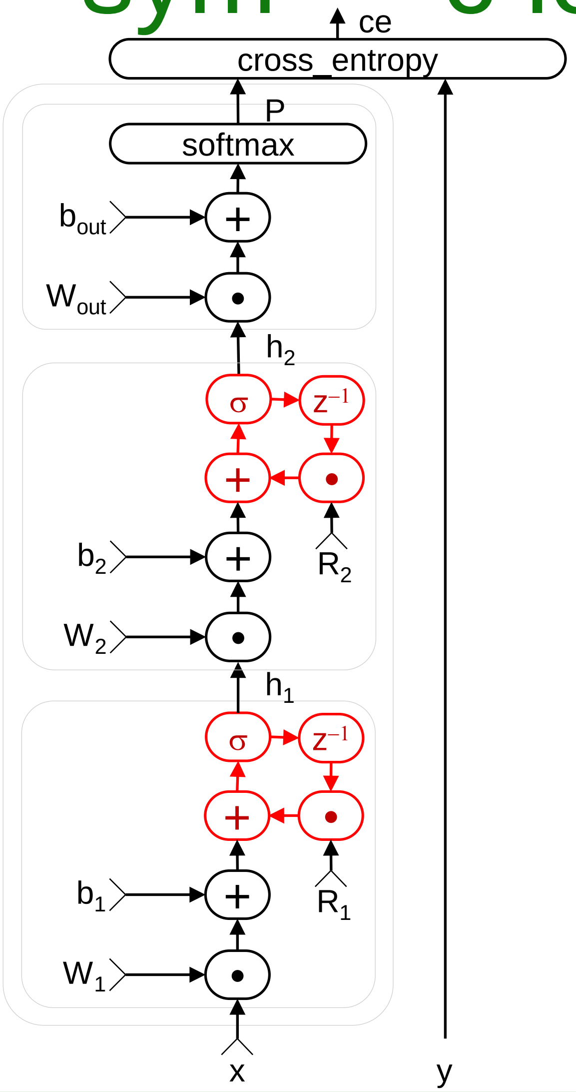
compare to id (Irvine Dataflow):

[Arvind et al., TR114a, Dept ISC, UC Irvine, Dec **1978**; "Executing a Program on the MIT Tagged-Token Dataflow Architecture", 1988]

```
Def ip A B = {  s = 0
             In
               {For j From 1 To n Do
                  Next s = s + A[j] * B[j]
                Finally s }} ;
```

```
@Function
def ip(a: Sequence[tensor], b: Sequence[tensor]):
    s0 = 0
    s_ = ForwardDeclaration()
    s = past_value(s_, initial_value=s0) + a * b
    s_.resolve_to(s)
    s = last(s)
    return s
```

symbolic loops over sequential data



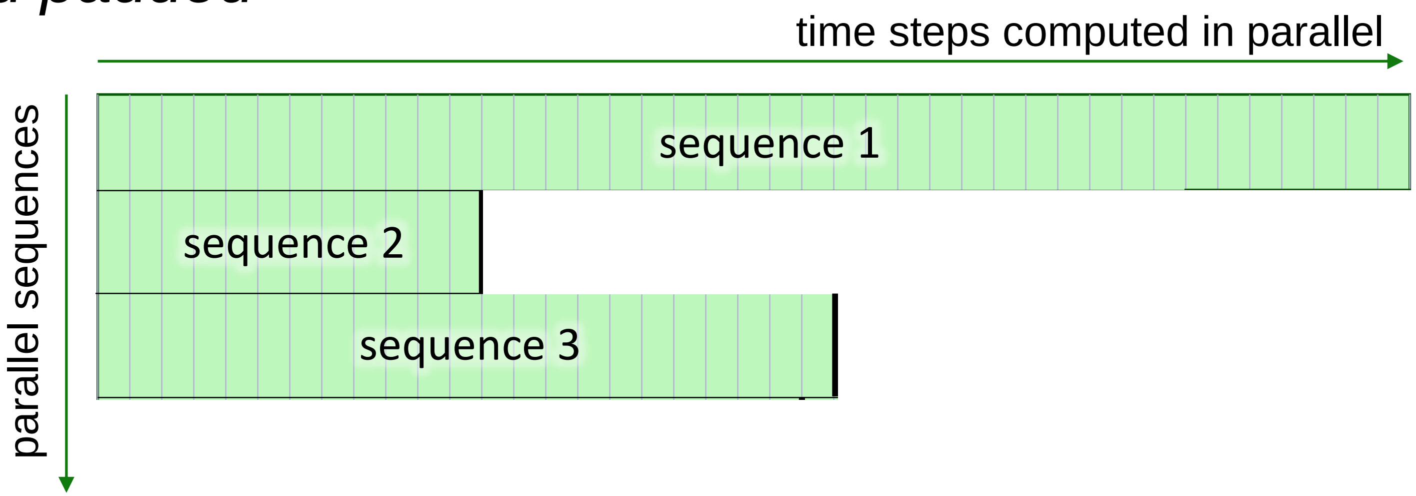

```
h1 = sigmoid(x  @ W1 + past_value(h1) @ R1 + b1)
h2 = sigmoid(h1 @ W2 + past_value(h2) @ R2 + b2)
P  = softmax(h2 @ Wout + bout)
ce = cross_entropy(P, L)
```

- CNTK automatically unrolls cycles at *execution time*
  - cycles are detected with Tarjan's algorithm
  - only nodes in cycles
- efficient and composable
  - cf. TensorFlow: [https://www.tensorflow.org/versions/r1.0/tutorials/recurrent/index.html]

```
lstm = rnn_cell.BasicLSTMCell(lstm_size)
state = tf.zeros([batch_size, lstm.state_size]
probabilities = []
loss = 0.0
for current_batch_of_words in words_in_dataset:
    output, state = lstm(current_batch_of_words, state)
    logits = tf.matmul(output, softmax_w) + softmax_b
    probabilities.append(tf.nn.softmax(logits))
    loss += loss_function(probabilities, target_words)
```

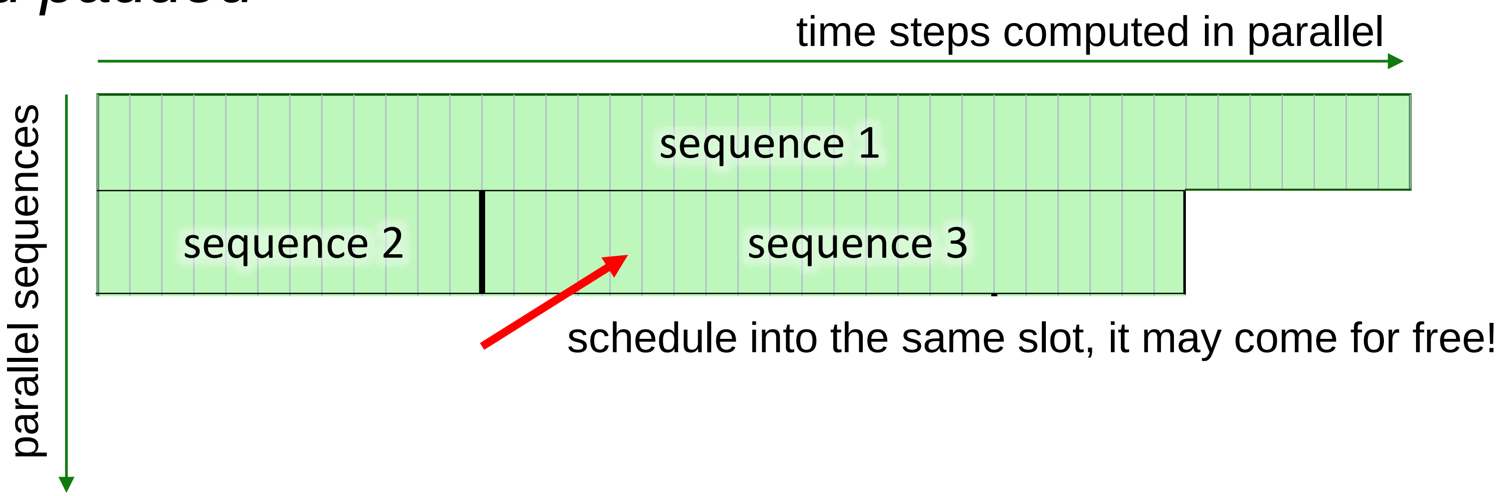
batch-scheduling of variable-length sequences

- minibatches containing sequences of different lengths are automatically packed *and padded*



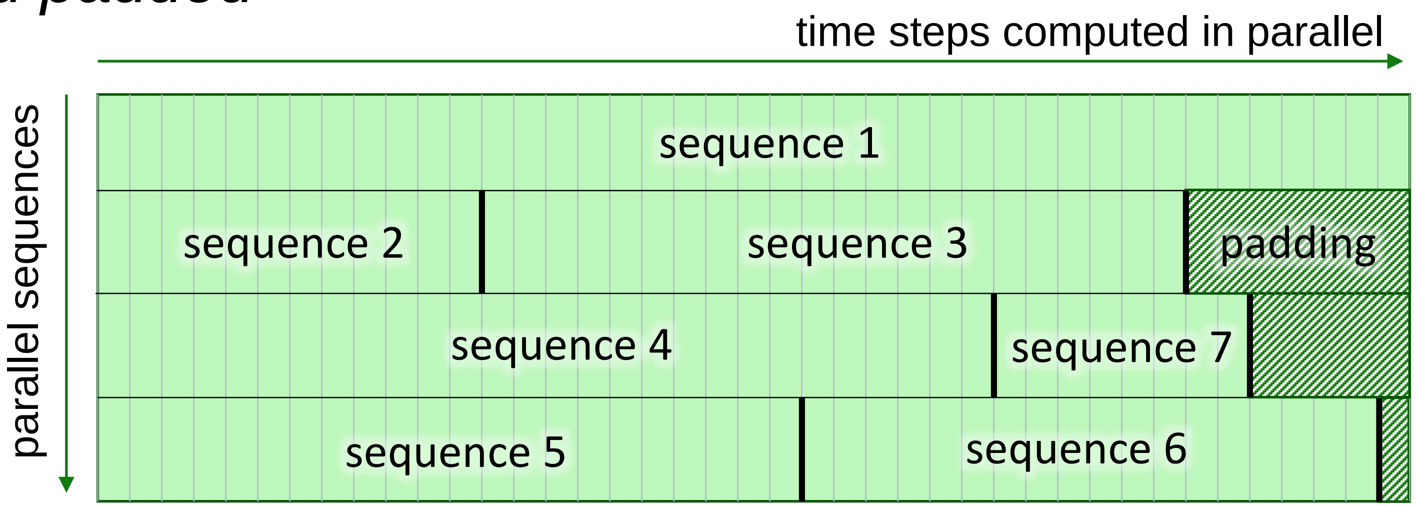
batch-scheduling of variable-length sequences

- minibatches containing sequences of different lengths are automatically packed *and padded*



batch-scheduling of variable-length sequences

- minibatches containing sequences of different lengths are automatically packed *and padded*



- fully transparent batching
  - recurrent → CNTK unrolls, handles sequence boundaries
  - non-recurrent operations → parallel
  - sequence reductions → mask

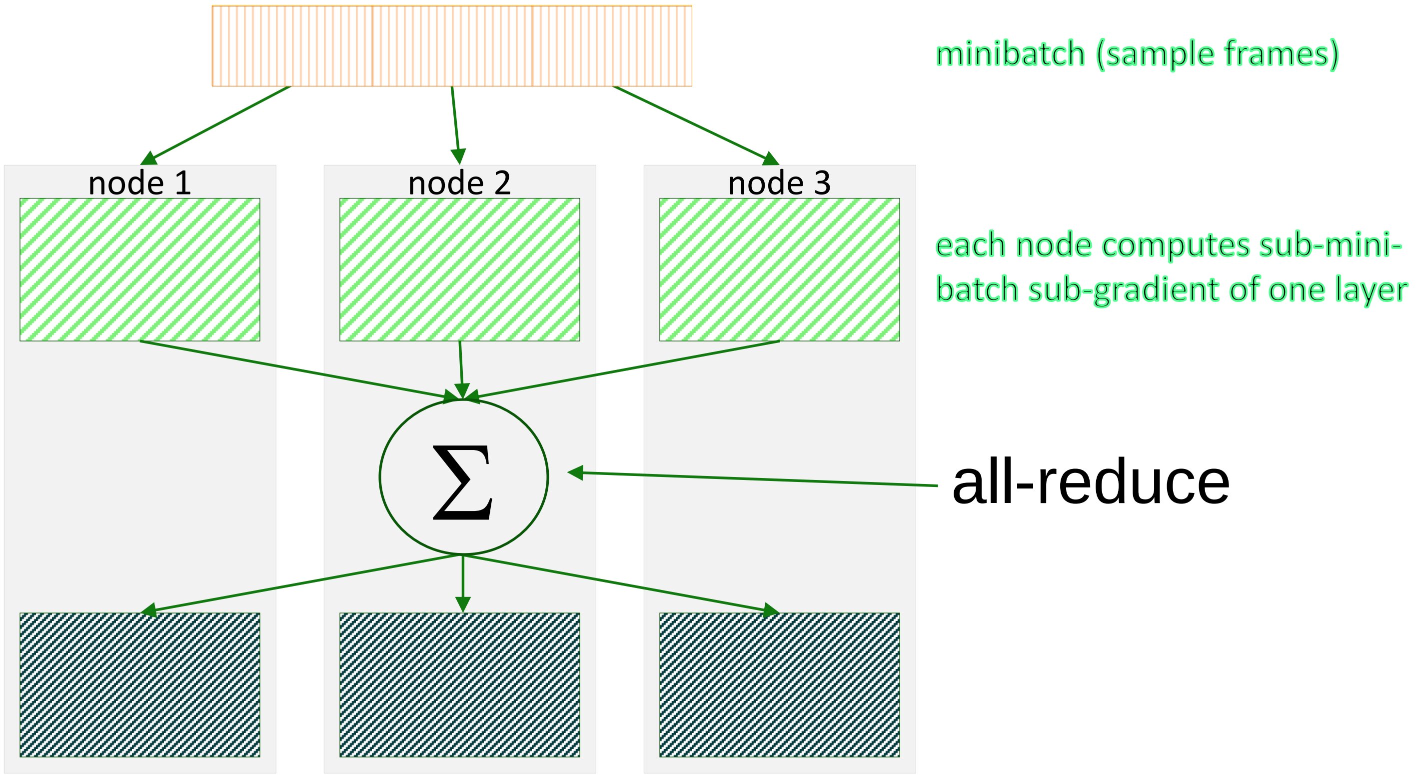
I. deep neural networks crash course
II. Microsoft Cognitive Toolkit (CNTK)
III. authoring neural networks
IV. executing neural networks
   -- GPU execution
   -- optimization
   -- parallelization
V. conclusion

data-parallel training

- degrees of parallelism:
  - within-vector parallelization: “vectorized”
  - across independent samples: “batching”
  - across GPUs: async PCIe device-to-device transfers
  - across servers: MPI etc., NVidia NCCL
- parallelization options:
  - **data-parallel**
  - model-parallel
  - layer-parallel

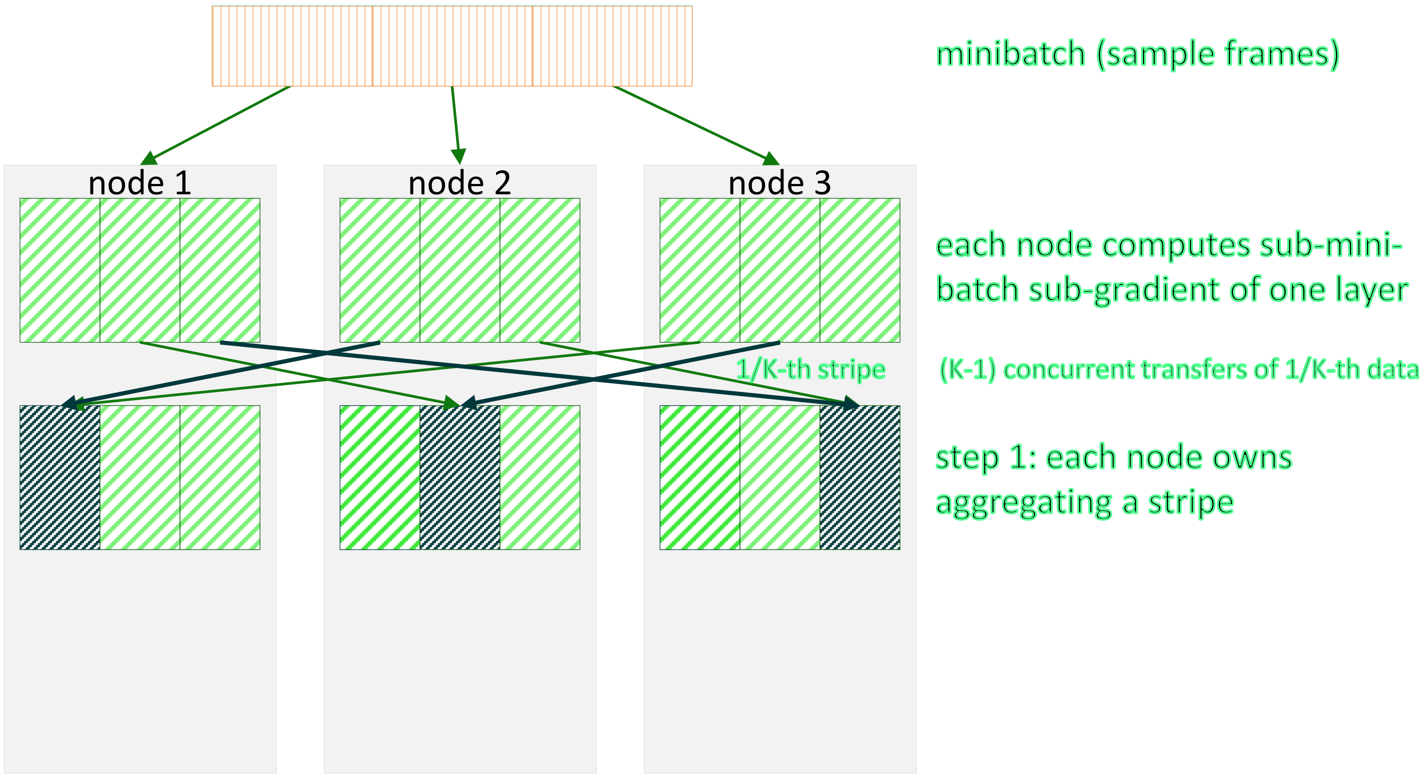
data-parallel training

- data-parallelism: distribute minibatch over workers, all-reduce partial gradients



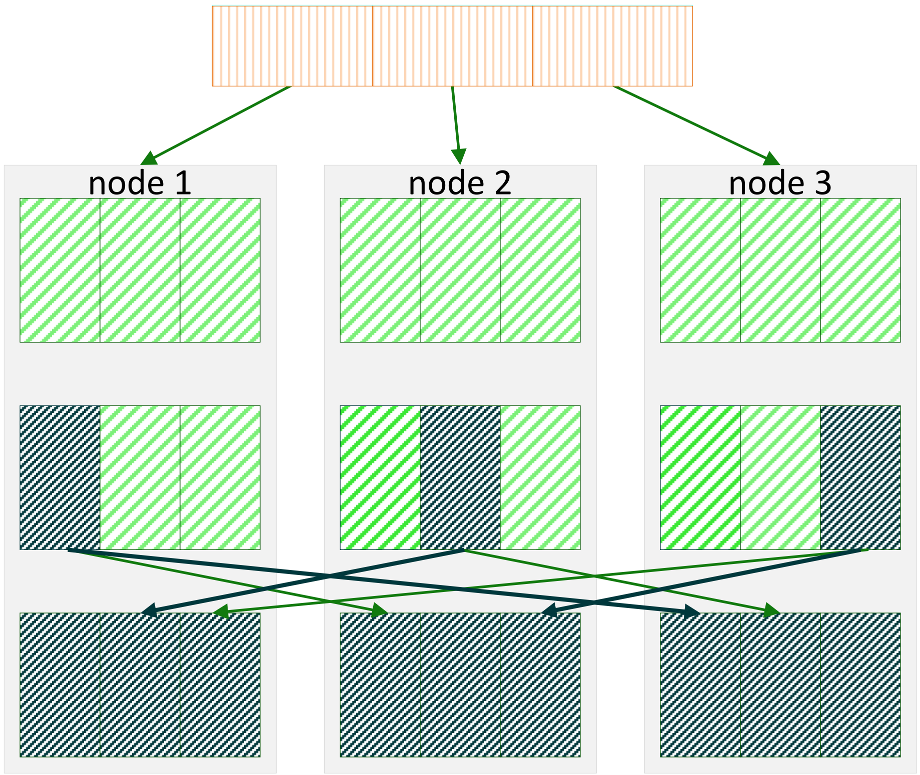
data-parallel training

- data-parallelism: distribute minibatch over workers, all-reduce partial gradients



data-parallel training

- data-parallelism: distribute minibatch over workers, all-reduce partial gradients



minibatch (sample frames)

each node computes sub-mini-batch sub-gradient of one layer

ring algorithm
O(2 (K-1)/K M)
→ O(1) w.r.t. K

step 1: each node owns aggregating a stripe

step 2: aggregated stripes are redistributed

data-parallel training

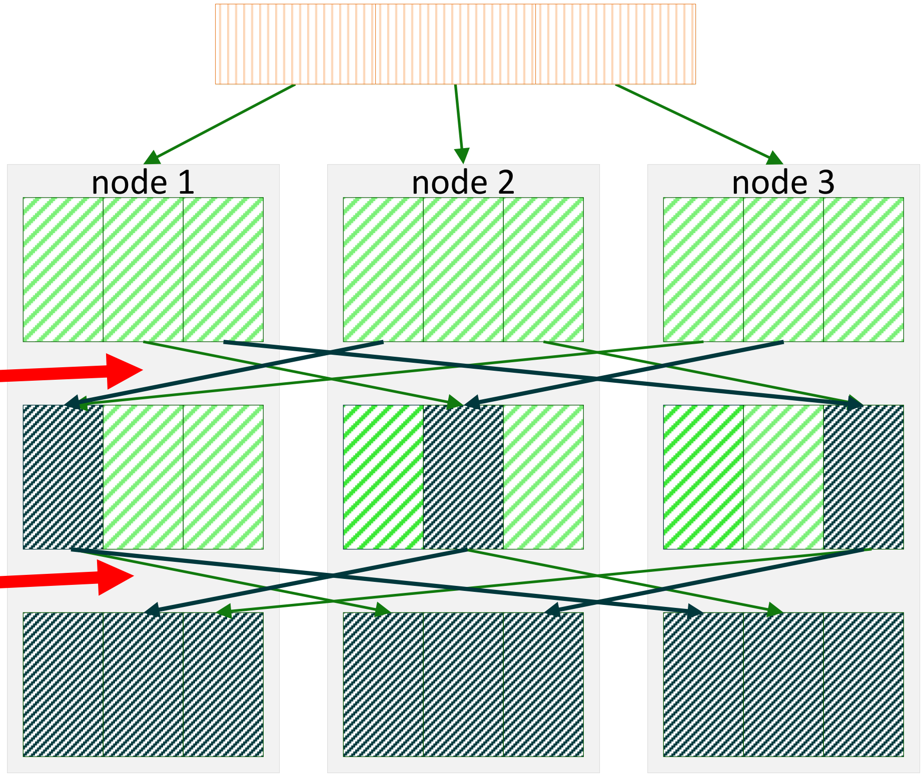
- data-parallelism: distribute minibatch over workers, all-reduce partial gradients
- O(1) — enough?
- example: DNN, MB size 1024, 160M model parameters
  - compute per MB: → 1/7 second
  - communication per MB: → 1/9 second (640M over 6 GB/s)
  - can't even parallelize to 2 GPUs: communication cost already dominates!
- how about doing it asynchronously?
  - HogWild! [-], DistBelief ASGD [Dean *et al.*, 2012]
  - does not change the problem fundamentally
  - (but helps with latency and jitter)

data-parallel training

how to reduce communication cost:

communicate less each time

- 1-bit SGD:
  [F. Seide, H. Fu, J. Droppo, G. Li, D. Yu: “1-Bit Stochastic Gradient Descent... Distributed Training of Speech DNNs”, Interspeech 2014]
  - quantize gradients to 1 bit per value
  - trick: carry over quantization error to next minibatch



data-parallel training

how to reduce communication cost:

communicate less each time

- 1-bit SGD:
  [F. Seide, H. Fu, J. Droppo, G. Li, D. Yu: “1-Bit Stochastic Gradient Descent... Distributed Training of Speech DNNs”, Interspeech 2014]
  - quantize gradients to 1 bit per value
  - trick: carry over quantization error to next minibatch
- alternative: 3-level quantization (with residual)
  [Nikko Ström: “Scalable Distributed DNN Training Using Commodity GPU Cloud Computing”, Interspeech 2015]
  - most gradients are close to 0
  - using 3 levels allows very good data compression
  - very sparse: all-reduce → all-to-all

data-parallel training

how to reduce communication cost:

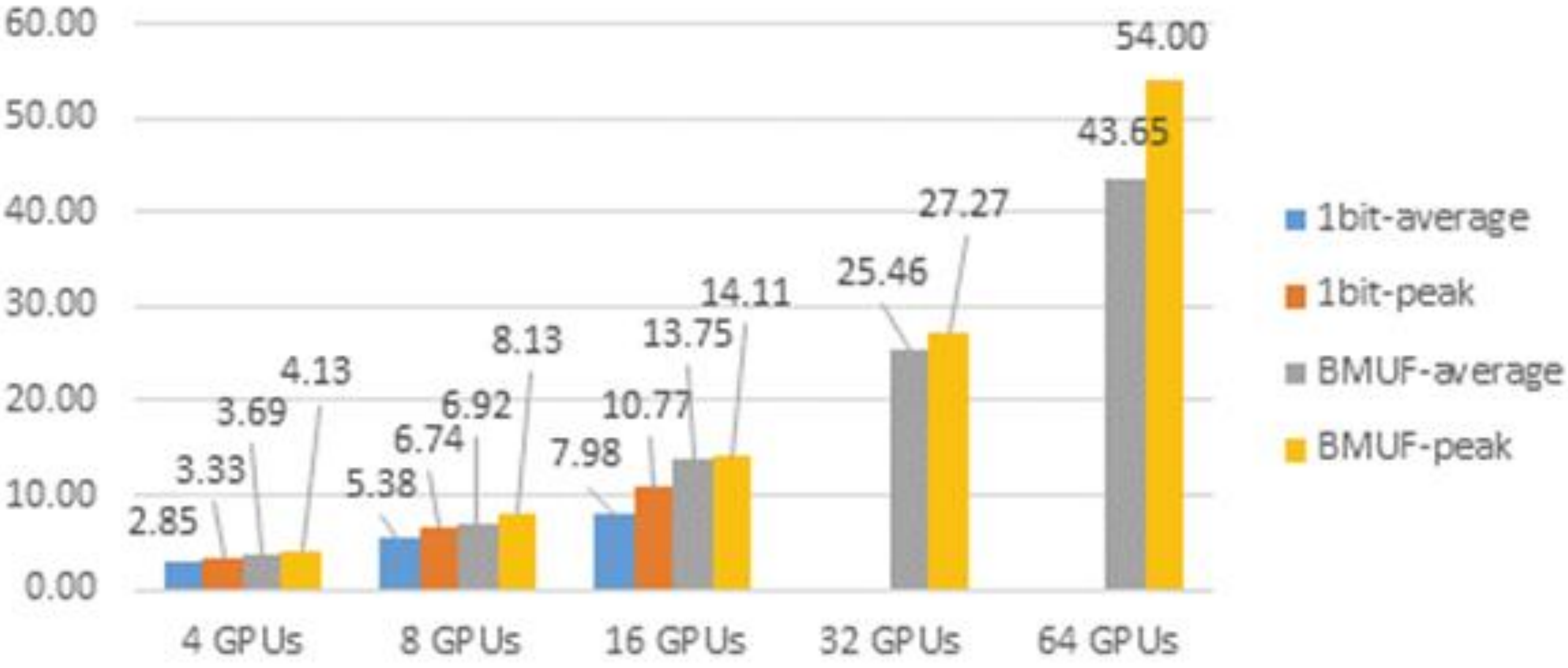
communicate less each time

- 1-bit SGD: [F. Seide, H. Fu, J. Droppo, G. Li, D. Yu: "1-Bit Stochastic Gradient Descent...Distributed Training of Speech DNNs", Interspeech 2014]
  - quantize gradients to 1 bit per value
  - trick: carry over quantization error to next minibatch

communicate less often

- automatic MB sizing [F. Seide, H. Fu, J. Droppo, G. Li, D. Yu: "ON Parallelizability of Stochastic Gradient Descent...", ICASSP 2014]
- block momentum [K. Chen, Q. Huo: "Scalable training of deep learning machines by incremental block training…," ICASSP 2016]
  - very recent, very effective parallelization method
  - combines model averaging with error-residual idea

data-parallel training



LSTM SGD baseline	11.08				
Parallel Algorithms	4-GPU	8-GPU	16-GPU	32-GPU	64-GPU
1bit	10.79	10.59	11.02		
BMUF	10.82	10.82	10.85	10.92	11.08

Table 2: WERs (%) of parallel training for LSTMs

[Yongqiang Wang, IPG; internal communication]

I. deep neural networks crash course
II. Microsoft Cognitive Toolkit (CNTK)
III. authoring neural networks
IV. executing neural networks
    -- GPU execution
    -- optimization
    -- parallelization
V. conclusion

how CNTK addresses the two key questions:

- **how to author neural networks?**
  - **functional programming paradigm**, well-matching the nature of DNNs
  - focus on **what, not how**
  - familiar syntax and flexibility through **EDSL on Python**
  - transparent automatic differentiation (expression graph: “implementation detail”)
- **how to execute them efficiently?**
  - turn graph into parallel program through **minibatching**
  - **symbolic loops** over sequences with dynamic scheduling
  - unique **parallel training algorithms** (1-bit SGD, Block Momentum)

Microsoft Cognitive Toolkit

aka.ms/CognitiveToolkit

Microsoft

challenges going forward

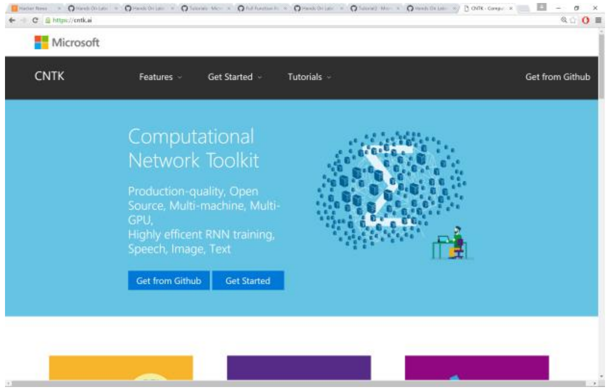
- flexibility vs. efficiency trade-off still not satisfactorily solved
- representational power of DNNs not complete
  - YES: logic & state machines
  - YES: simple data structures (tensors, sequences)
  - **NO**: structured data (composites/aggregates, references, symbolic knowledge, data bases)
- data scarcity → libraries
  - pre-trained neural networks
  - *world knowledge*

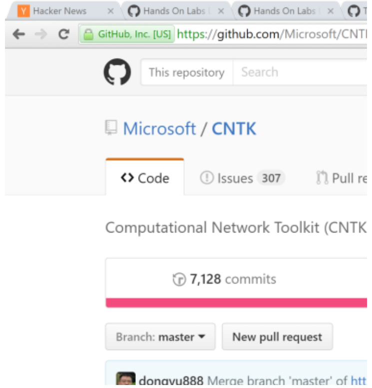
conclusion

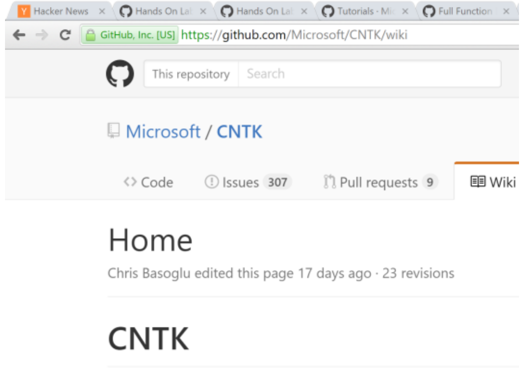
- deep neural networks are a **new paradigm of creating programs**
  - NNs and differentiable computing should be **1st-class citizens in PL and architectures**, maximizing expressiveness and efficiency
  - CNTK is guided by this
- deep neural networks touch upon many **classic CS problems**
  - auto-diff, PL, optimization, hybrid architectures, parallelization (GPU/farms), big data
  - often requires some change to DNN algorithms
- looking forward to many great contributions from these three communities!

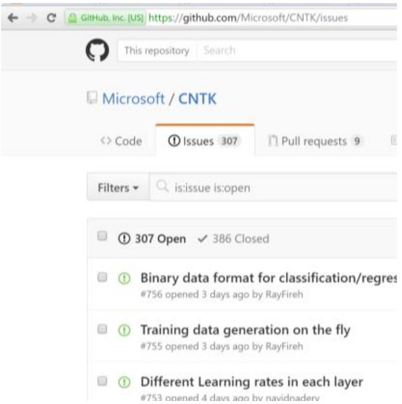
Cognitive Toolkit: democratizing the AI tool chain

- Web site: https://cntk.ai/
- Github: https://github.com/Microsoft/CNTK
- Wiki: https://github.com/Microsoft/CNTK/wiki
- Issues: https://github.com/Microsoft/CNTK/issues









mailto:fseide@microsoft.com

Microsoft