Deep Learning and Automatic Differentiation

from Theano to PyTorch

Atılım Güneş Baydin
gunes@robots.ox.ac.uk

CSCS-ICS-DADSi Summer School
Swiss National Supercomputing Centre
September 5, 2017

UNIVERSITY OF
OXFORD

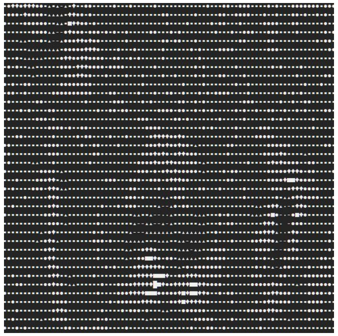
About me

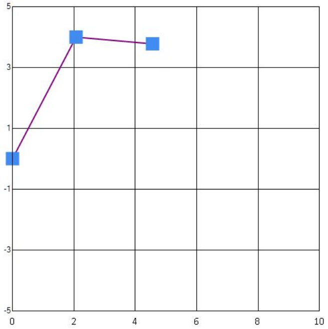
Postdoctoral researcher, University of Oxford

- Working with Frank Wood, Department of Engineering Science
- Working on probabilistic programming and its applications in science (high-energy physics and HPC)

Long-term interests:

- Automatic (algorithmic) differentiation (e.g. http://diffsharp.github.io)
- Evolutionary algorithms
- Computational physics





Deep learning

Deep learning

A reincarnation/rebranding of **artificial neural networks**, with roots in

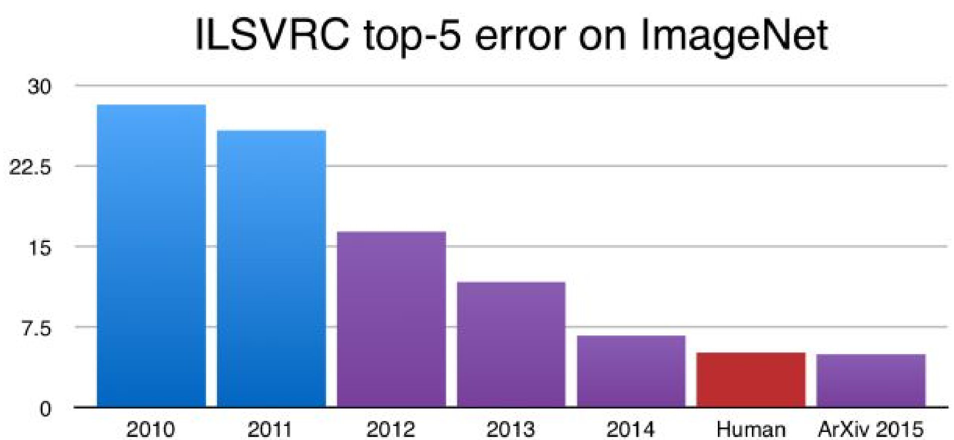
- Threshold logic (McCulloch & Pitts, 1943)
- Hebbian learning (Hebb, 1949)
- Perceptron (Rosenblatt, 1957)
- Backpropagation in NNs (Werbos, 1975; Rumelhart, Hinton, Williams, 1986)

Frank Rosenblatt with the Mark I Perceptron, holding a set of neural network weights

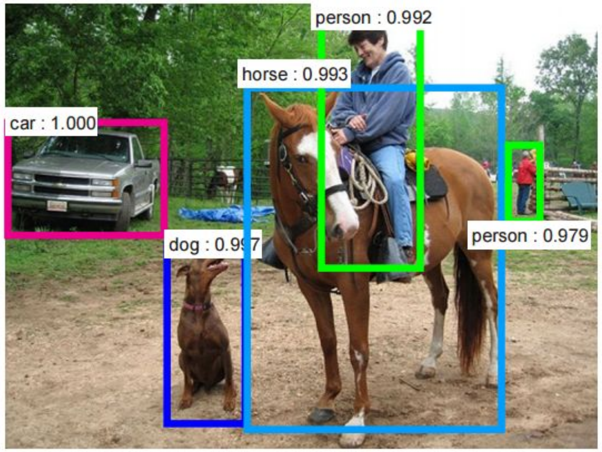
Deep learning

State of the art in **computer vision**

- ImageNet classification with deep convolutional neural networks (Krizhevsky et al., 2012)
  - Halved the error rate achieved with pre-deep-learning methods
- Replacing hand-engineered features
- Modern systems surpass human performance



Top-5 error rate for ImageNet
https://devblogs.nvidia.com/parallelforall/mocha-jl-deep-learning-julia/

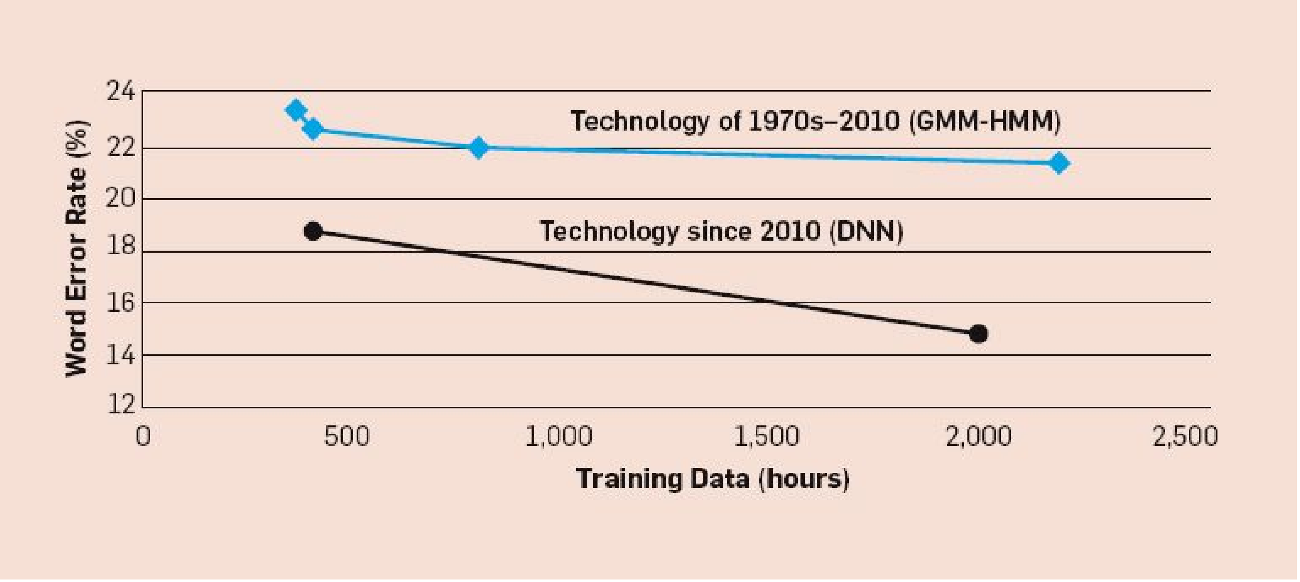


Faster R-CNN
(Ren et al., 2015)

Deep learning

State of the art in **speech recognition**

- Seminal work by Hinton et al. (2012)
  - First major industrial application of deep learning
- Replacing HMM-GMM-based models

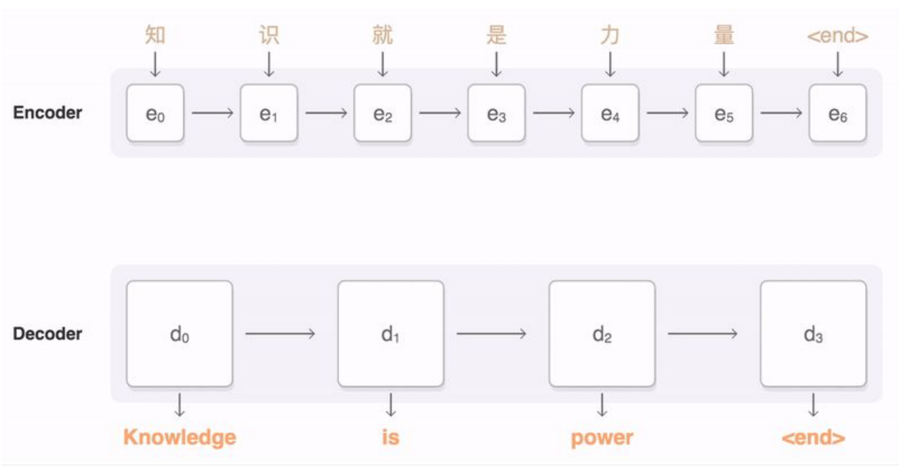


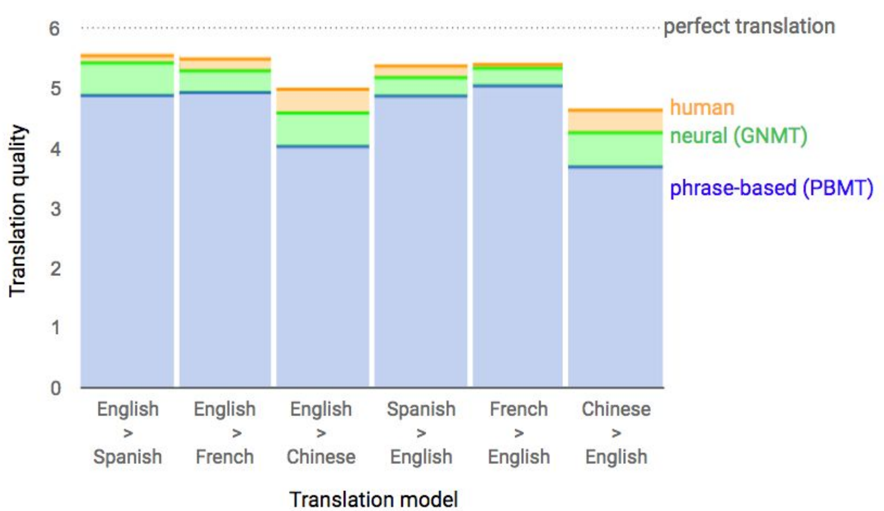
Recognition word error rates (Huang et al., 2014)

Deep learning

State of the art in **machine translation**

- Based on RNNs and CNNs (Bahdanu et al., 2015; Wu et al., 2016)
- Replacing statistical translation with engineered features
- Google (Sep 2016), Microsoft (Nov 2016), Facebook (Aug 2017) moved to neural machine translation
  https://techcrunch.com/2017/08/03/facebook-finishes-its-move-to-neural-machine-translation/





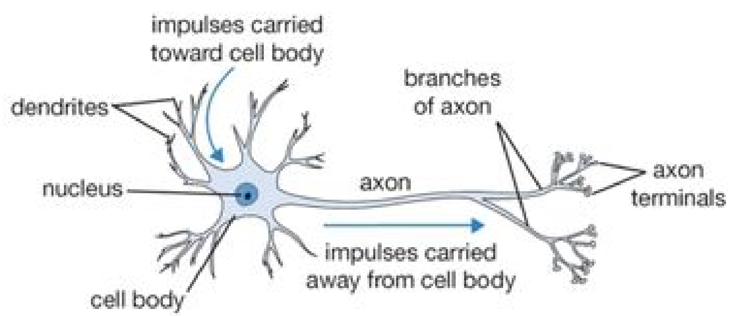
Google Neural Machine Translation System (GNMT)

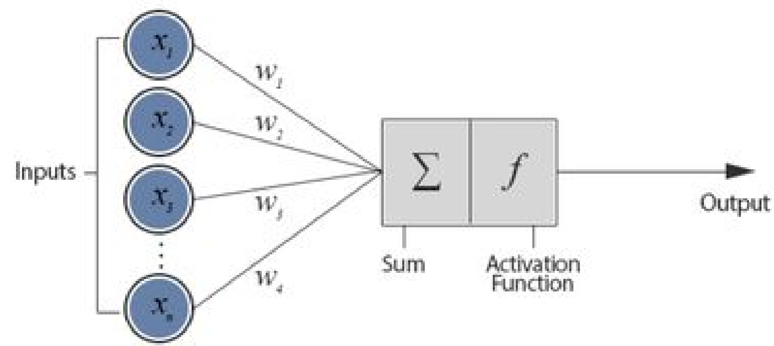
What makes deep learning tick?

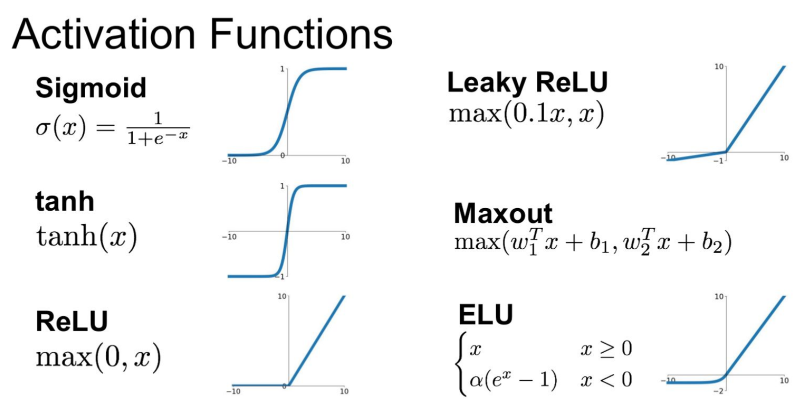
Deep neural networks

An artificial “neuron" is loosely based on the biological one

- Receive a set of weighted inputs (dendrites)
- Integrating and transforming (cell body)
- Passing the output further (axon)



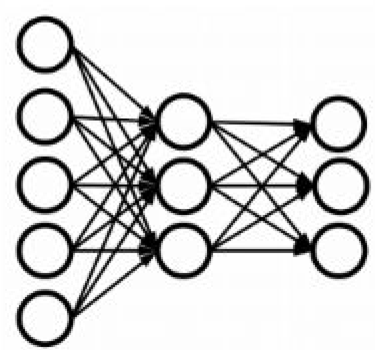


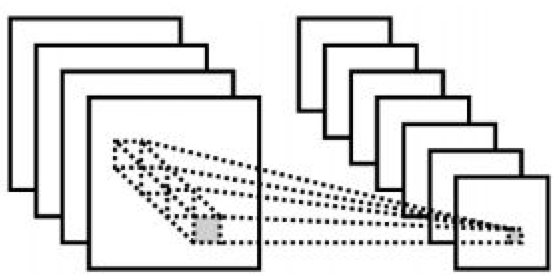


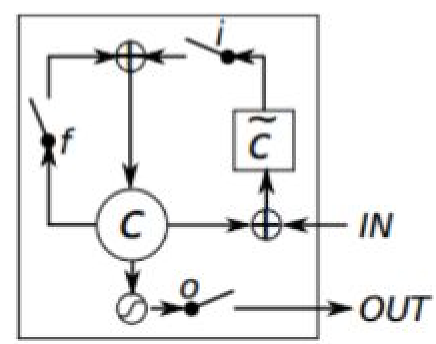
Deep neural networks

Three **main building blocks**

- Feedforward (Rosenblatt, 1957)
- Convolutional (LeCun et al., 1989)
- Recurrent (Hopfield, 1982; Hochreiter & Schmidhuber, 1997)



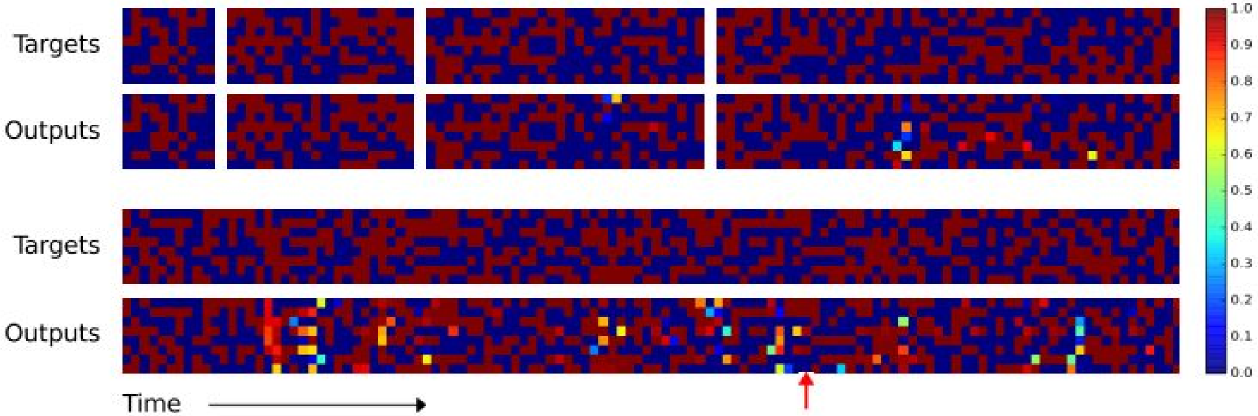




Deep neural networks

Newer additions introduce **(differentiable) algorithmic elements**

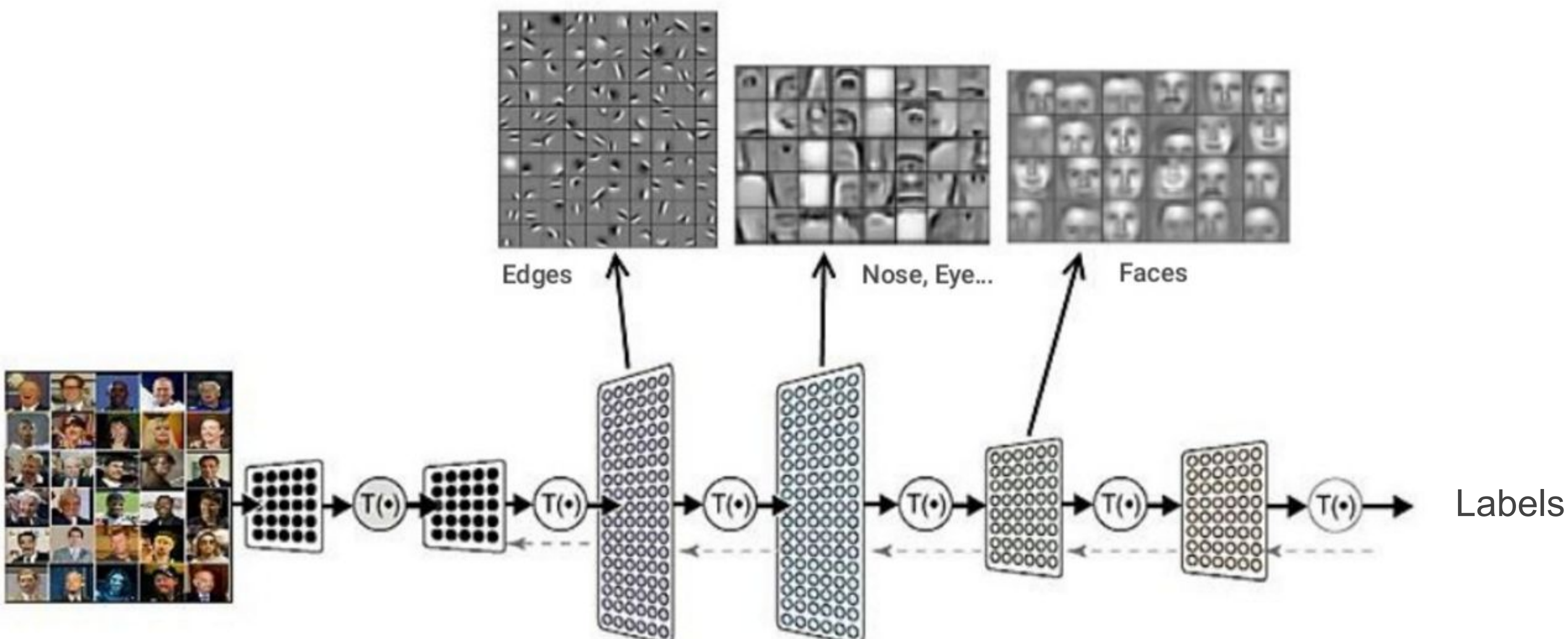
- Neural Turing Machine (Graves et al., 2014)
  - Can infer algorithms: copy, sort, recall
- Stack-augmented RNN (Joulin & Mikolov, 2015)
- End-to-end memory network (Sukhbaatar et al., 2015)
- Stack, queue, deque (Grefenstette et al., 2015)
- Discrete interfaces (Zaremba & Sutskever, 2015)



Neural Turing Machine on copy task (Graves et al., 2014)

Deep neural networks

- Deep, distributed representations learned end-to-end (also called "representation learning")
  - From **raw pixels** to **object labels**
  - From **raw waveforms of speech** to **text**

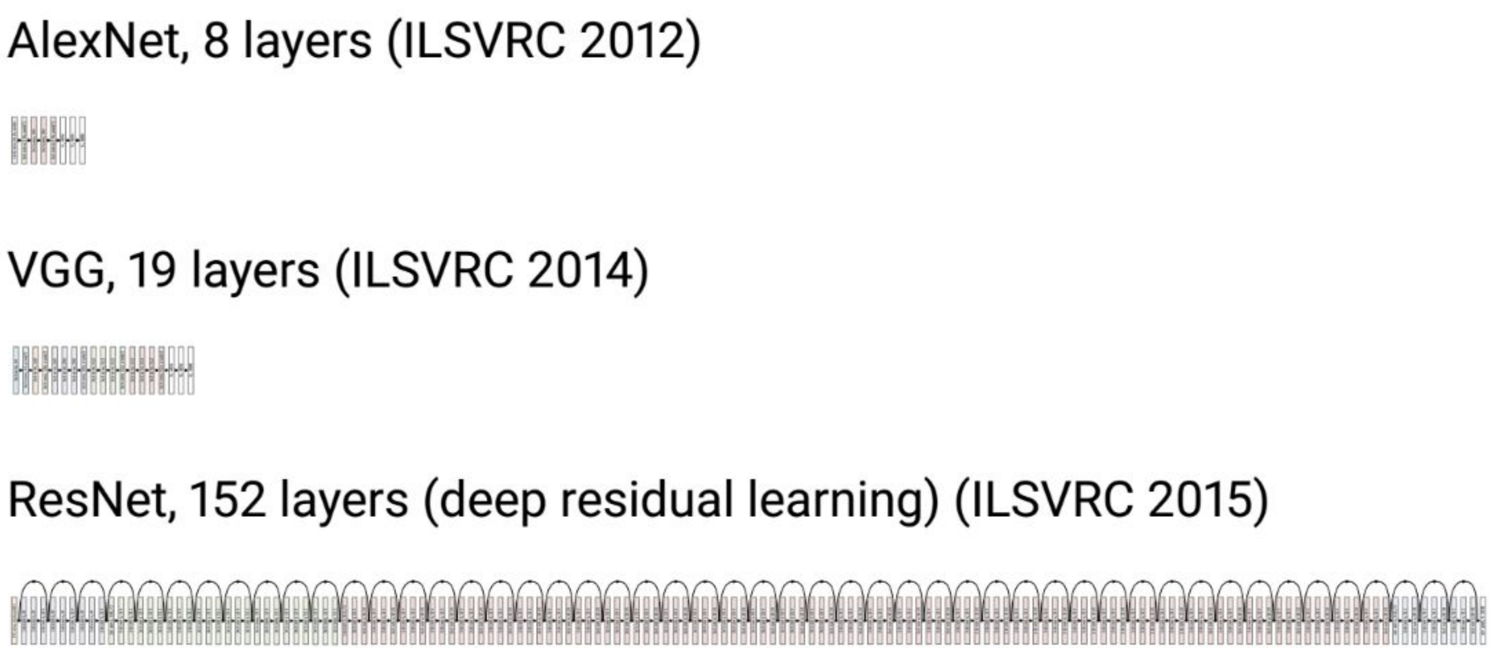


(Masuch, 2015)

Deep neural networks

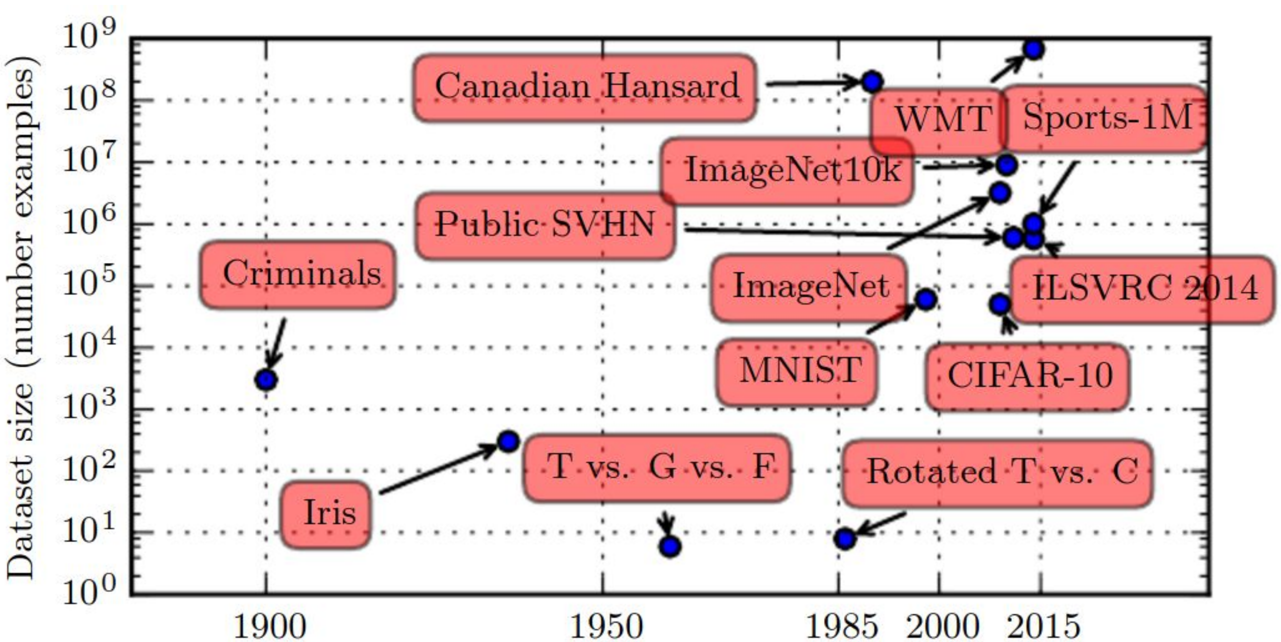
Deeper is better (Bengio, 2009)

- Deep architectures consistently beat shallow ones
- Shallow architectures exponentially inefficient for same performance
- Hierarchical, distributed representations achieve non-local generalization



Data

- Deep learning needs massive amounts of data
  - need to be labeled if performing supervised learning
- A rough rule of thumb: *"deep learning will generally match or exceed human performance with a dataset containing at least 10 million labeled examples."* (The Deep Learning book, Goodfellow et al., 2016)

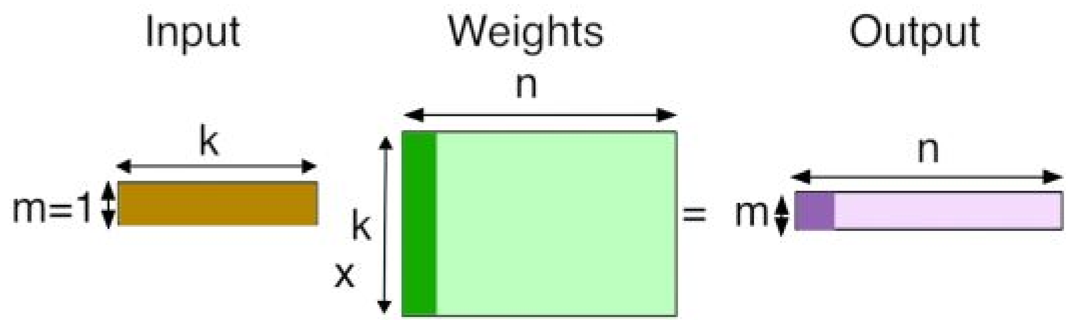


Dataset sizes (Goodfellow et al., 2016)

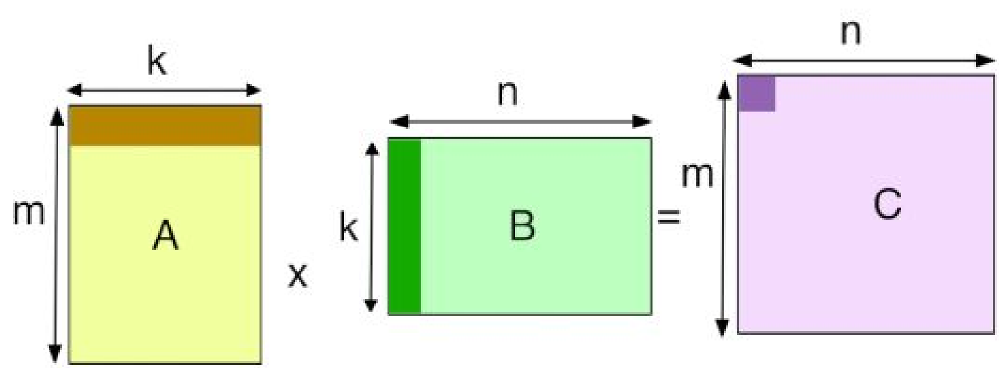
GPUs

Layers of NNs are conveniently expressed as series of **matrix multiplications**

One input vector,
n neurons of *k* inputs



A batch of *m* input vectors



GPUs

- BLAS (mainly GEMM) is at the hearth of mainstream deep learning, commonly running on off-the-shelf graphics processing units
- Rapid adoption after
  - Nvidia released CUDA (2007)
  - Raina et al. (2009) and Ciresan et al. (2010)
- ASICs such as tensor processing units (TPUs) are being introduced
  - As low as 8-bit floating point precision, better power efficiency

Nvidia Titan Xp (2017)

Google Cloud TPU server

Deep learning frameworks

- Modern tools make it **extremely easy to implement / reuse models**
- Off-the-shelf components
  - Simple: linear, convolution, recurrent layers
  - Complex: compositions of complex models (e.g., CNN + RNN)
- Base frameworks: Torch (2002), Theano (2011), Caffe (2014), TensorFlow (2015), PyTorch (2016)
- Higher-level model-building libraries: Keras (Theano & TensorFlow), Lasagne (Theano)
- High-performance low-level bindings: BLAS, Intel MKL, CUDA, Magma

Learning: gradient-based optimization

Loss function

$$Q(\boldsymbol{w}) = \sum_{i=1}^{N} Q_i(\boldsymbol{w})$$

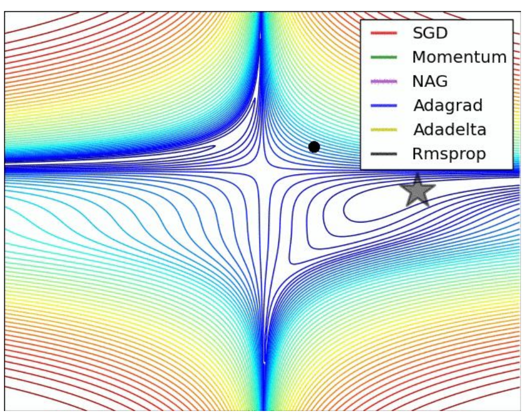
Parameter update

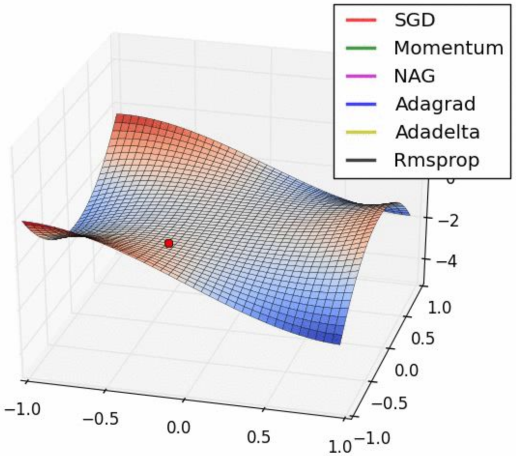
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \nabla_w Q(\boldsymbol{w})$$

Stochastic gradient descent (SGD)

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \sum_{i=1}^{d} \nabla_w Q_i(\boldsymbol{w})$$

In practice we use SGD or adaptive-learning-rate varieties such as Adam, RMSProp





(Ruder, 2017)

http://ruder.io/optimizing-gradient-descent/

Deep learning

Neural networks

\+ Data

\+ Gradient-based optimization

Deep learning

Neural networks

\+ Data

\+ Gradient-based optimization

We need derivatives

How do we compute derivatives?

Manual

- Calculus 101, rules of differentiation

General Formulas

1. $$\frac{d}{dx}c = 0$$

2. $$\frac{d}{dx}[f(x) \mp g(x)] = f'(x) \mp g'(x)$$

3. $$\frac{d}{dx}[f(x)g(x)] = f'(x)g(x) + g'(x) + f(x)$$

4. $$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x) - g'(x)f(x)}{(g(x))^2}$$

5. $$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$$

6. $$\frac{d}{dx}x^n = nx^{n-1}$$

Exponential and Logarithmic Functions

7. $$\frac{d}{dx}e^{f(x)} = f'(x)e^{f(x)}$$

8. $$\frac{d}{dx}a^x = a^x\ln(a)$$

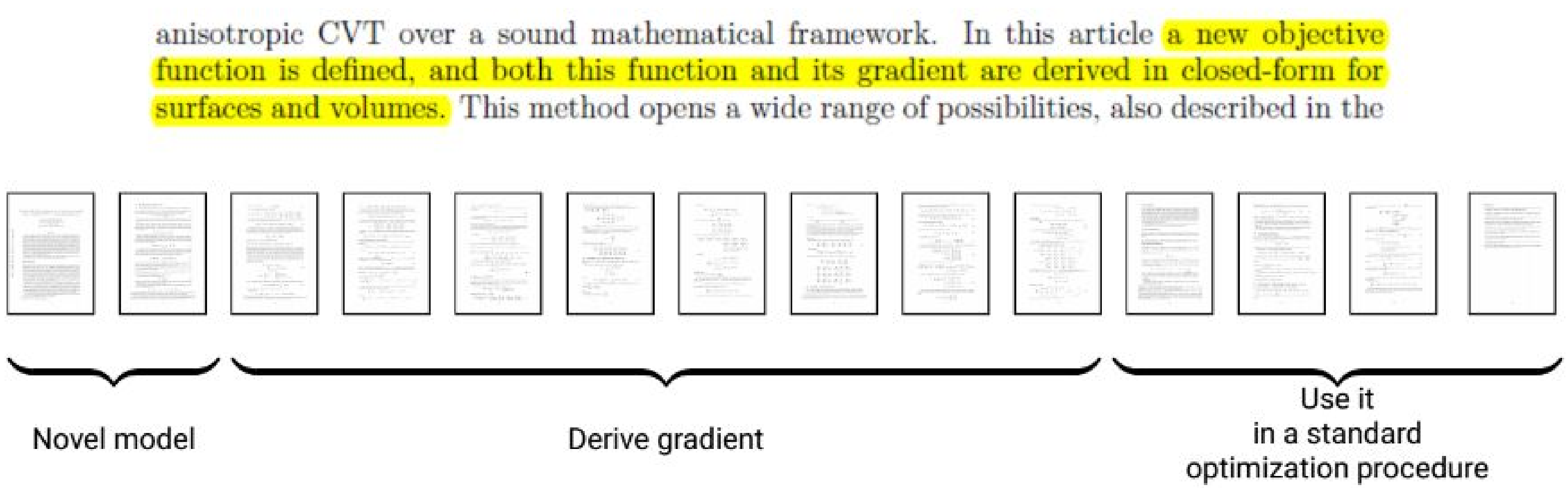
9. $$\frac{d}{dx}\ln(C|f(x)|) = \frac{d}{dx}[\ln(C) + \ln(f(x))] = \frac{f'(x)}{f(x)}$$

10. $$\frac{d}{dx}log_a(f(x)) = \frac{f'(x)}{xln(a)}$$

...

Manual

- Analytical derivatives are needed for theoretical insight
  - analytic solutions, proofs
  - mathematical analysis, e.g., stability of fixed points
- They are **unnecessary when we just need numerical derivatives** for optimization
- Until very recently, machine learning looked like this:

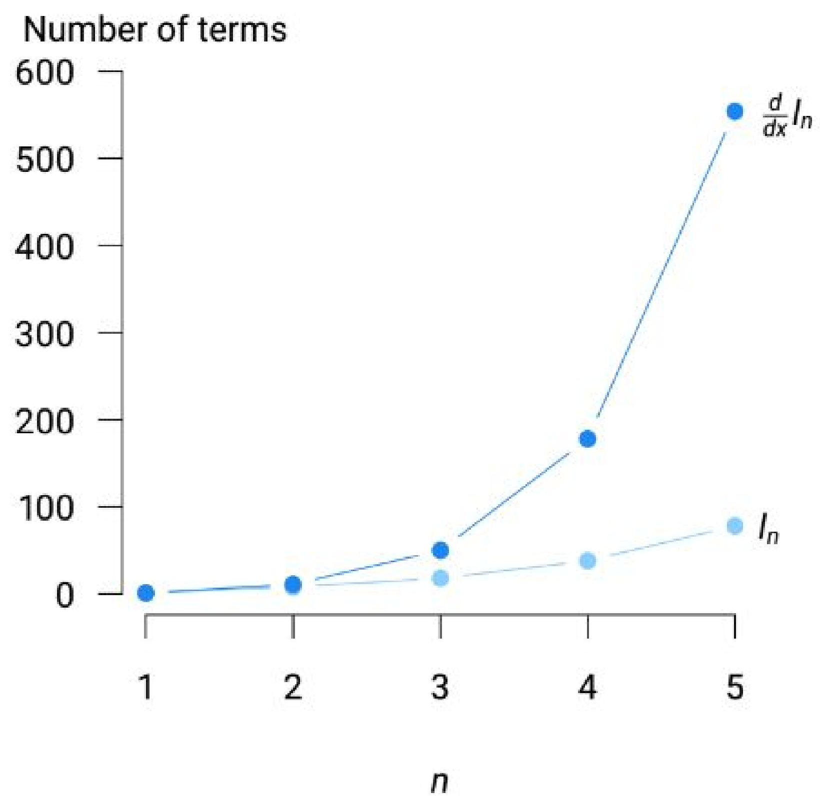


Symbolic derivatives

- Symbolic computation with Mathematica, Maple, Maxima, also deep learning frameworks such as Theano
- Main issue: **expression swell**

Logistic map $l_{n+1} = 4l_n(1 - l_n)$, $l_1 = x$

n	l_n	$\frac{d}{dx} l_n$
1	x	1
2	$4x(1-x)$	$4(1-x) - 4x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2 - 16x(1-2x)^2 - 64x(1-x)(1-2x)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2) + 64(1-x)(1-2x)^2(1-8x+8x^2)^2 - 64x(1-2x)^2(1-8x+8x^2)^2 - 256x(1-x)(1-2x)(1-8x+8x^2)^2$



Symbolic derivatives

- Symbolic computation with, e.g., Mathematica, Maple, Maxima
- Main limitation: **only applicable to closed-form expressions**

You can find the derivative of:

```
In [1]: def f(x):
            return 64 *(1-x) *(1-2*x)^2 *(1-8*x+8*x*x)^2
```

But not of:

```
In [2]: def f(x,n):
            if n == 1:
                return x
            else:
                v = x
                for i in range(1,n):
                    v = 4*v*(1-v)
                return v
```

In deep learning, symbolic graph builders such as Theano and TensorFlow face issues with control flow, loops, recursion

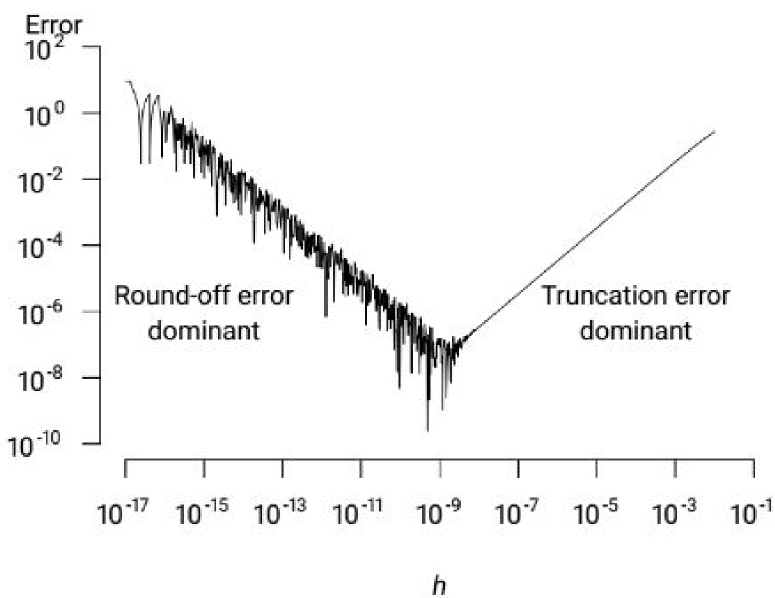
Numerical differentiation

Finite differences, for example:

$f : \mathbb{R}^n \to \mathbb{R}$, approximate the gradient $\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$ using

$$\frac{\partial f(\mathbf{x})}{\partial x_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}, \; 0 < h \ll 1$$

But we have to select h and we face **approximation errors**



Computed using

$$E(h, x^*) = \left| \frac{f(x^* + h) - f(x^*)}{h} - \frac{d}{dx} f(x)\Big|_{x^*} \right|$$

$$f(x) = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$

$$x^* = 0.2$$

Numerical differentiation

Better approximations exist

- Higher-order finite differences
  E.g.
  $$\frac{\partial f(\mathbf{x})}{\partial x_i} = \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h} + O(h^2) \ ,$$
- Richardson extrapolation
- Differential quadrature

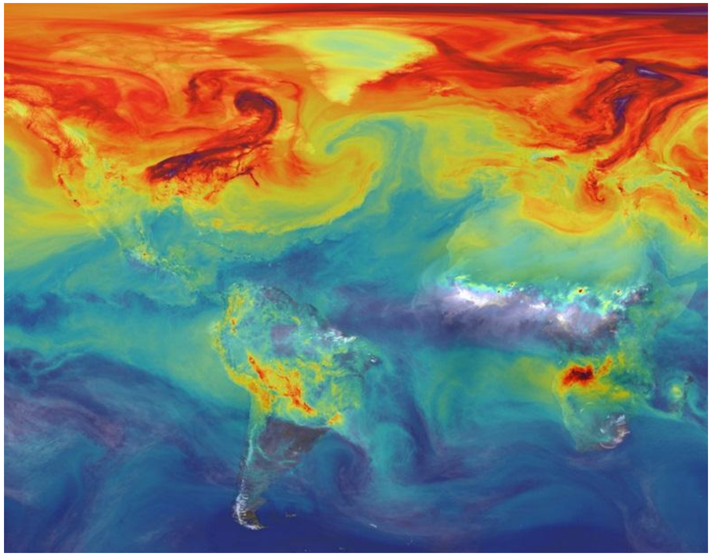
but they increase rapidly in complexity and never completely eliminate the error

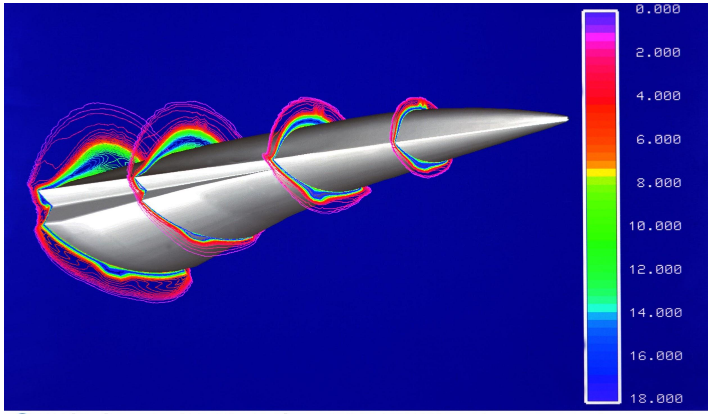
Automatic differentiation

- Small but established subfield of scientific computing
  http://www.autodiff.org/

- Traditional application domains:
  - Computational fluid dynamics
  - Atmospheric sciences
  - Engineering design optimization
  - Computational finance

AD has shared roots with the backpropagation algorithm for neural networks, but it is more general





NASP–Computer Aided Design Computational Fluid Dynamics
NASA Langley Research Center 6/6/1988 Image # EL–1996–00191

Automatic differentiation

Given an algorithm **A**,

- build an augmented algorithm **A'**
- for each value, keep a primal and a derivative component (dual numbers)
- compute the derivatives along with the original values

All algorithms are compositions of a finite set of elementary operations (with known derivatives)

```
f(a, b):
    c = a * b
    d = sin c
    return d
```

```
f'(a, a', b, b'):
    (c, c') = (a*b, a'*b + a*b')
    (d, d') = (sin c, c' * cos c)
    return (d, d')
```

Automatic differentiation

Given an algorithm **A**,

- build an augmented algorithm **A’**
- for each value, keep a primal and a derivative component (dual numbers)
- compute the derivatives along with the original values

All algorithms are compositions of a finite set of elementary operations (with known derivatives)

```
f(a, b):
   c = a * b
   d = sin c
   return d
```

→

```
f’(a, a’, b, b’):
   (c, c’) = (a*b, a’*b + a*b’)
   (d, d’) = (sin c, c’ * cos c)
   return (d, d’)
```

Exact derivatives, not an approximation

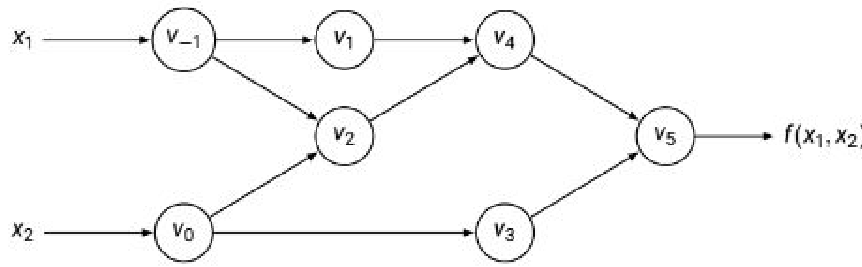
Automatic differentiation

AD has two main modes:

- **Forward mode:** straightforward
- **Reverse mode**: slightly more difficult, when you see it for the first time

Forward mode

Let's take $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$

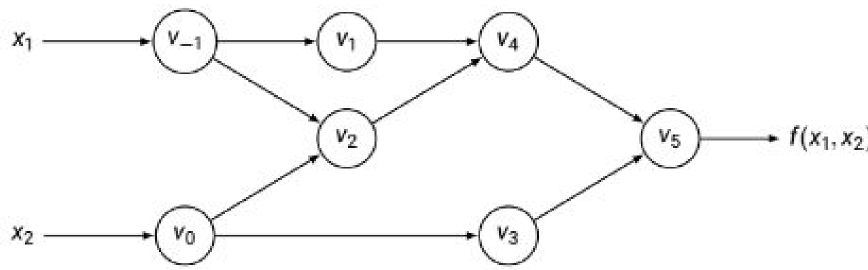


Derivatives propagate: independent $\rightarrow$ dependent

- select a var. of differentiation x_i
- augment each intermediate value v_j with $\dot{v}_j = \frac{\partial v_j}{\partial x_i}$
- set $\dot{x}_i = 1$
- run the algorithm forward

Forward mode

Let's take $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



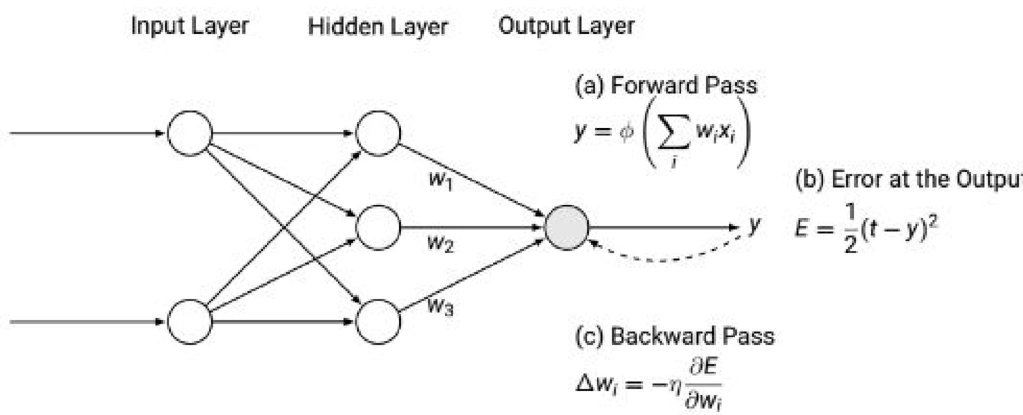
Forward Evaluation Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Forward Derivative Trace		
$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
$\dot{v}_0$	$= \dot{x}_2$	$= 0$
$\dot{v}_1$	$= \dot{v}_{-1} / v_{-1}$	$= 1/2$
$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\boldsymbol{\dot{y}}$	$\boldsymbol{= \dot{v}_5}$	$\mathbf{= 5.5}$

Reverse mode

If you know the maths behind backpropagation,
you know reverse mode AD

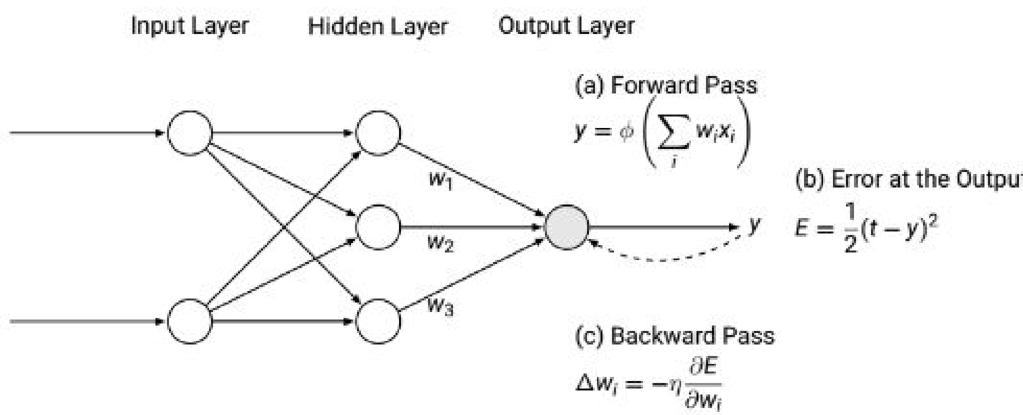
Backpropagation **is just a special case** of reverse mode AD



Reverse mode

If you know the maths behind backpropagation,
you know reverse mode AD

Backpropagation **is just a special case** of reverse mode AD

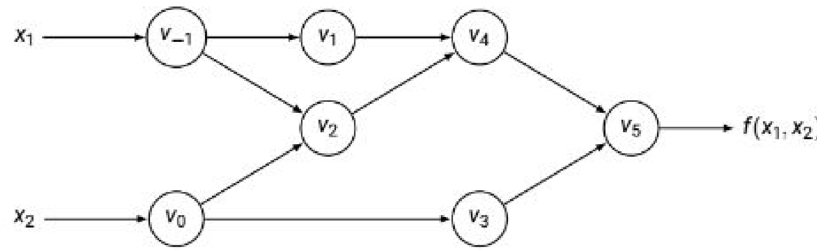


Origins in the same papers (Bryson and Ho, 1969; Werbos, 1974)
Backprop. brought to fame by Rumelhart et al. (1986)

AD and machine learning communities somehow managed to stay disconnected

Reverse mode

Again take $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



Derivatives propagate: dependent $\rightarrow$ independent
1st stage:

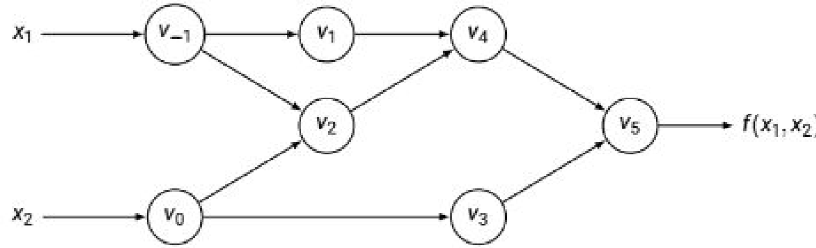
- run your original code forward

2nd stage:

- select a dependent var. y_j
- augment each intermediate value v_i with $\bar{v}_i = \frac{\partial y_j}{\partial v_i}$ (adjoint)
- set $\bar{y}_j = 1$
- run the algorithm backward

Reverse mode

Again take $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



Forward Evaluation Trace

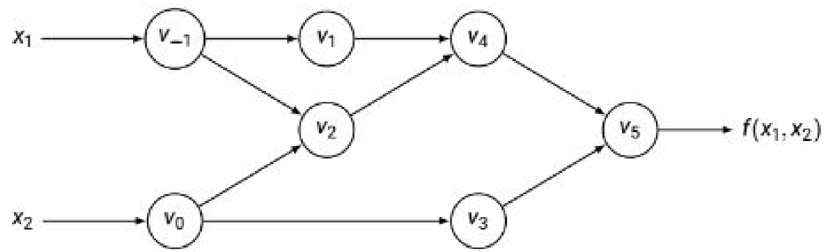
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Reverse Adjoint Trace

$\boldsymbol{\bar{x}_1}$	$\boldsymbol{= \bar{v}_{-1}}$		$\mathbf{= 5.5}$
$\boldsymbol{\bar{x}_2}$	$\boldsymbol{= \bar{v}_0}$		$\mathbf{= 1.716}$
$\bar{v}_{-1}$	$= \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0$	$= \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1}$	$= \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0$	$= \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5$	$= \bar{y}$	$= 1$	

Reverse mode

Again take $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



In [1]:
```
import torch
from torch import Tensor
from torch.autograd import Variable
```

In [2]:
```
x1 = Variable(Tensor([2]),requires_grad=True)
x2 = Variable(Tensor([5]),requires_grad=True)
```

In [3]:
```
v1 = torch.log(x1)
v2 = x1 * x2
v3 = torch.sin(x2)
v4 = v1 + v2
y = v4 - v3
```

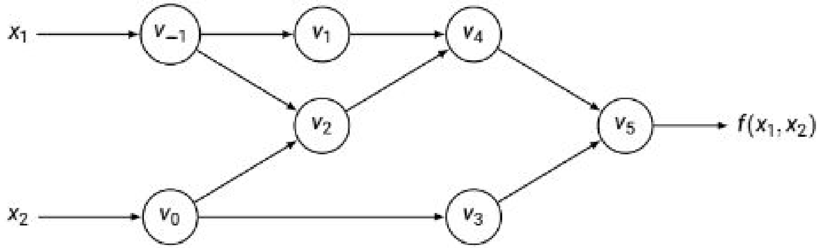
In [4]:
```
y.backward()
print(x1.grad.data)
print(x2.grad.data)
```

```
 5.5000
[torch.FloatTensor of size 1]


 1.7163
[torch.FloatTensor of size 1]
```

Reverse mode

Again take $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



In [1]:
```
import torch
from torch import Tensor
from torch.autograd import Variable
```

In [2]:
```
x1 = Variable(Tensor([2]),requires_grad=True)
x2 = Variable(Tensor([5]),requires_grad=True)
```

In [3]:
```
v1 = torch.log(x1)
v2 = x1 * x2
v3 = torch.sin(x2)
v4 = v1 + v2
y = v4 - v3
```

In [4]:
```
y.backward()
print(x1.grad.data)
print(x2.grad.data)
```

```
 5.5000
[torch.FloatTensor of size 1]


 1.7163
[torch.FloatTensor of size 1]
```

More on this in the afternoon exercise session

Forward vs reverse

In the extreme cases,

for $F : \mathbb{R} \to \mathbb{R}^m$, forward AD can compute all $\left(\frac{\partial F_1}{\partial x}, \ldots, \frac{\partial F_m}{\partial x} \right)$

for $f : \mathbb{R}^n \to \mathbb{R}$, reverse AD can compute $\nabla f = \left(\frac{\partial f}{\partial x_i}, \ldots, \frac{\partial f}{\partial x_n} \right)$

in **just one application**

Forward vs reverse

In the extreme cases,
for $F : \mathbb{R} \to \mathbb{R}^m$, forward AD can compute all $\left(\frac{\partial F_1}{\partial x}, \ldots, \frac{\partial F_m}{\partial x}\right)$
for $f : \mathbb{R}^n \to \mathbb{R}$, reverse AD can compute $\nabla f = \left(\frac{\partial f}{\partial x_i}, \ldots, \frac{\partial f}{\partial x_n}\right)$
in **just one application**

In general, for $f : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian $\mathbf{J} \in \mathbb{R}^{m \times n}$ takes

- $O(n \times \text{time}(f))$ with forward AD
- $O(m \times \text{time}(f))$ with reverse AD

Reverse mode performs better when $n \gg m$

Derivatives and deep learning frameworks

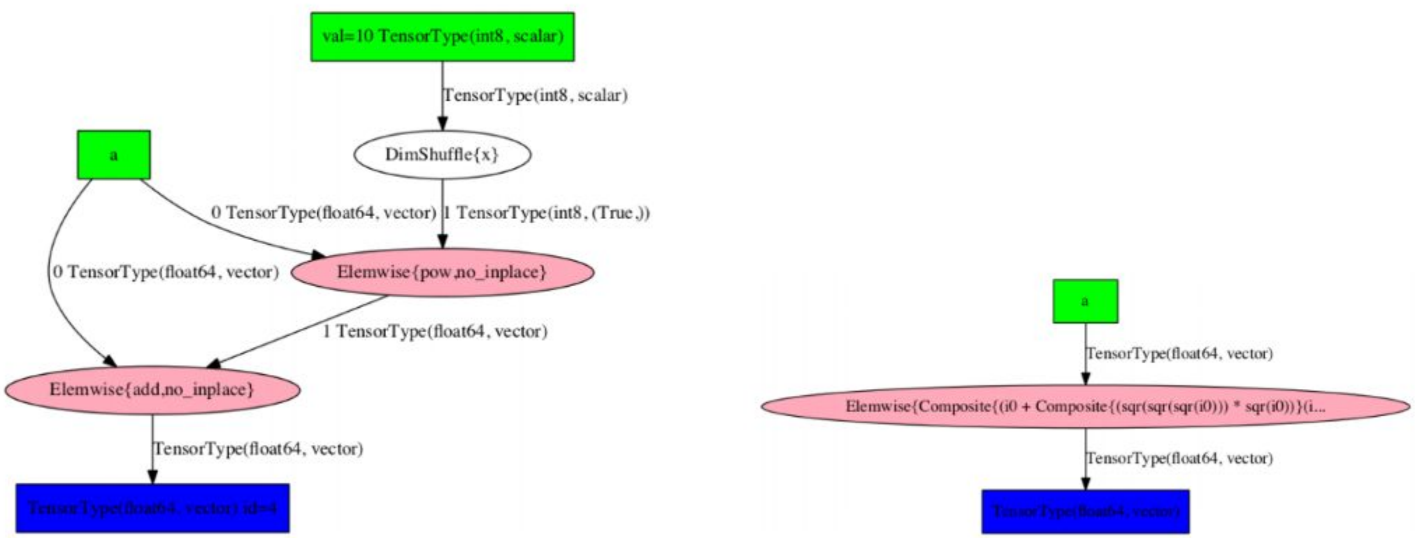
Deep learning frameworks

Two main families:

- Symbolic graph builders
- Dynamic graph builders (general-purpose AD)

Symbolic graph builders

- Implement models using symbolic placeholders, using a mini-language
- Severely limited (and unintuitive) control flow and expressivity
- The graph gets “compiled” to take care of expression swell



Graph compilation in Theano

Symbolic graph builders

Theano, TensorFlow, CNTK

You are limited to symbolic graph building, with the mini-language

For example, instead of this in pure Python (for A^k):

```
result = 1
for i in xrange(k):
  result = result * A
```

You build this symbolic graph:

```
import theano
import theano.tensor as T

k = T.iscalar("k")
A = T.vector("A")

# Symbolic description of a loop
result, updates = theano.scan(fn=lambda prior_result, A: prior_result * A,
                              outputs_info=T.ones_like(A),
                              non_sequences=A,
                              n_steps=k)
final_result = result[-1]

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

Dynamic graph builders (general-purpose AD)

- Use general-purpose **automatic differentiation** operator overloading
- Implement models as regular programs, with full support for control flow, branching, loops, recursion, procedure calls

Dynamic graph builders (general-purpose AD)

autograd (Python) by Harvard Intelligent Probabilistic Systems Group
https://github.com/HIPS/autograd

- torch-autograd by Twitter Cortex
https://github.com/twitter/torch-autograd

- PyTorch
http://pytorch.org/

a general-purpose AD system that allows you to **implement models as regular Python programs** (more on this in the exercise session)

```
result = Variable(Tensor([1]))
for i in range(k):
    result = result * A
```

```
result.backward()
print(A.grad.data)
```

Summary

Summary

- Neural networks + data + gradient descent = deep learning
- General-purpose AD is the future of deep learning
- More distance to cover:
  - Implementations better than operator overloading exist in AD literature (source transformation) but not in machine learning
  - Forward AD is not currently present in any mainstream machine learning framework
  - Nesting of forward and reverse AD enables efficient higher-order derivatives such as Hessian-vector products

Thank you!

CSCS-ICS-DADSi Summer School
Swiss National Supercomputing Centre
September 5, 2017

UNIVERSITY OF
OXFORD

References

Baydin, A.G., Pearlmutter, B.A., Radul, A.A. and Siskind, J.M., 2015. Automatic differentiation in machine learning: a survey. arXiv preprint arXiv:1502.05767.

Baydin, Atılım Güneş, Barak A. Pearlmutter, and Jeffrey Mark Siskind. 2016. “Tricks from Deep Learning.” In 7th International Conference on Algorithmic Differentiation, Christ Church Oxford, UK, September 12–15, 2016.

Baydin, Atılım Güneş, Barak A. Pearlmutter, and Jeffrey Mark Siskind. 2016. “DiffSharp: An AD Library for .NET Languages.” In 7th International Conference on Algorithmic Differentiation, Christ Church Oxford, UK, September 12–15, 2016.

Maclaurin, D., Duvenaud, D. and Adams, R., 2015, June. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning* (pp. 2113-2122).

Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, Jeffrey Dean. Technical Report, 2016.

Convolutional Sequence to Sequence Learning. Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, Yann N. Dauphin. arXiv, 2017

Neural Machine Translation by Jointly Learning to Align and Translate. Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio. International Conference on Learning Representations, 2015.

A Historical Perspective of Speech Recognition. Xuedong Huang, James Baker, Raj Reddy. Communications of the ACM, Vol. 57 No. 1, Pages 94-103 10.1145/2500887

Bengio, Yoshua. "Learning deep architectures for AI." Foundations and trends® in Machine Learning 2.1 (2009): 1-127.