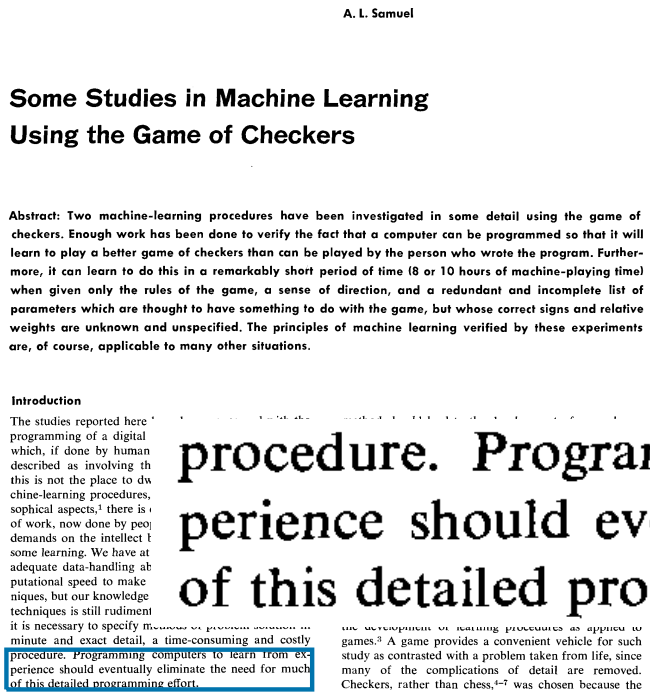
Introduction to Machine Learning

Yue Shi Lai

LBNL NSD

September 13, 2017

Machine learning: Origin



A. L. Samuel

Some Studies in Machine Learning Using the Game of Checkers

Abstract: Two machine-learning procedures have been investigated in some detail using the game of checkers. Enough work has been done to verify the fact that a computer can be programmed so that it will learn to play a better game of checkers than can be played by the person who wrote the program. Furthermore, it can learn to do this in a remarkably short period of time (8 or 10 hours of machine-playing time) when given only the rules of the game, a sense of direction, and a redundant and incomplete list of parameters which are thought to have something to do with the game, but whose correct signs and relative weights are unknown and unspecified. The principles of machine learning verified by these experiments are, of course, applicable to many other situations.

Introduction

The studies reported here ... programming of a digital ... which, if done by human ... described as involving th... this is not the place to dw... chine-learning procedures, ... sophical aspects,[1] there is ... of work, now done by peo... demands on the intellect ... some learning. We have at ... adequate data-handling ab... putational speed to make ... niques, but our knowledge ... techniques is still rudiment... it is necessary to specify ... minute and exact detail, a time-consuming and costly procedure. Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort.

games.[3] A game provides a convenient vehicle for such study as contrasted with a problem taken from life, since many of the complications of detail are removed. Checkers, rather than chess,[4–7] was chosen because the

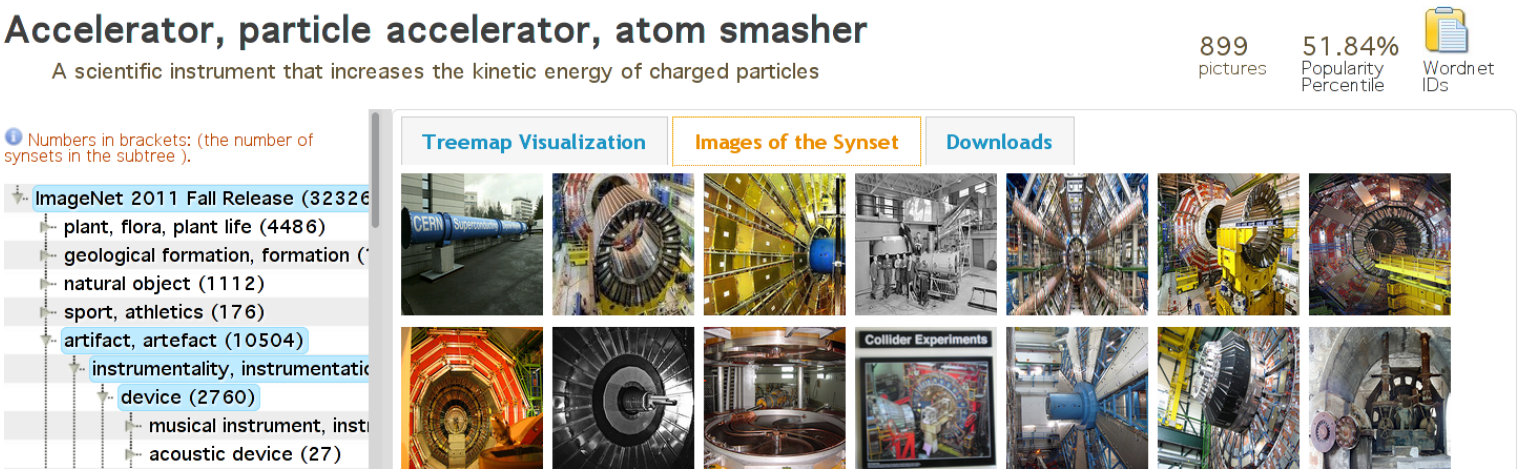
A. L. Samuel, Some Studies in Machine Learning Using the Game of Checkers, IBM J. Res. Dev. 3, 210 (1959)

procedure. Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort.

- Long time competitor to “symbolic AI”
- Began with Arthur Samuels getting an IBM 701 to play checkers

Why modern machine learning works

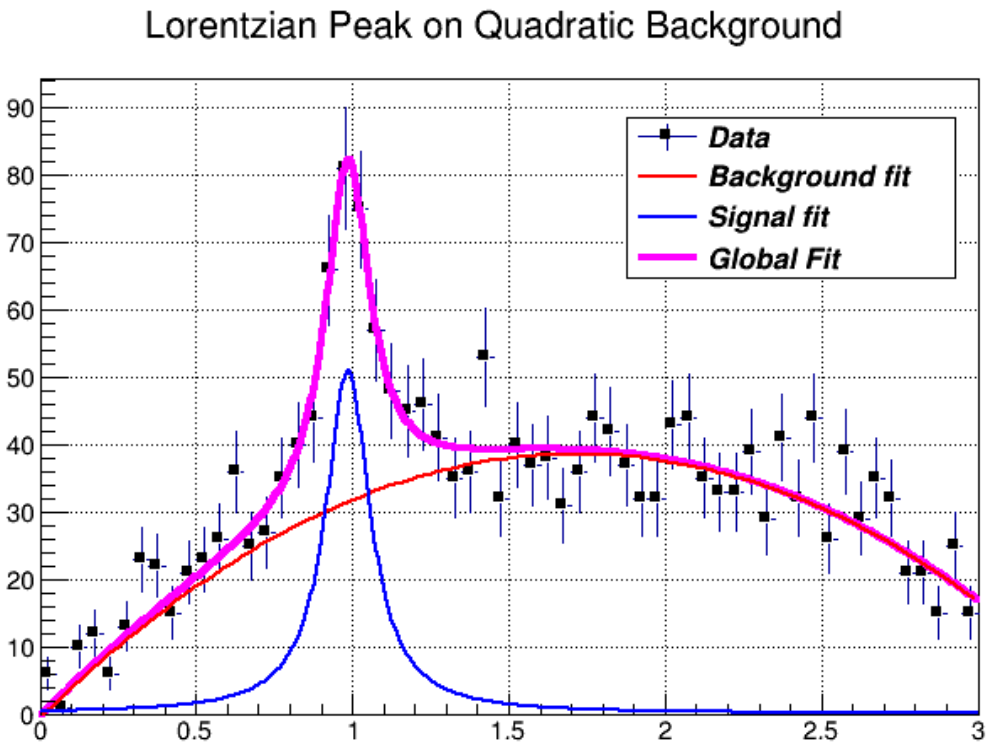
- Large (labeled) datasets – not a problem for high energy physics (HEP)/nuclear physics (NP)
  - ImageNet: 10^7 images (Flickr and other sources) with labels



http://image-net.org/

  - Compare typical sample size in CS around year 2000: 6×10^4 (MNIST, modified NIST handwriting)
- Fast (and cheap) matrix-multiply hardware: Graphics processing unit (GPU), vector/multicore CPU

Machine learning: You probably have done it

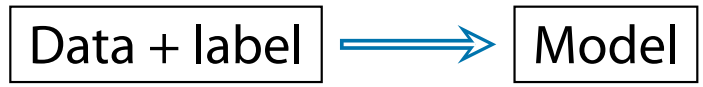


- Classical regression analysis is a form of machine learning
- Learning the parameters of a function without “detailed programming effort”
- Algorithms: linear least squares, damped least squares (Levenberg-Marquardt)

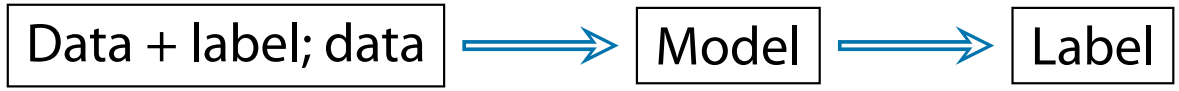
Machine learning

- **Supervised**
  - Classification (e.g. particles)
  - Regression (e.g. energy/momentum meausre)
  - Ranking (e.g. web pages, movies)

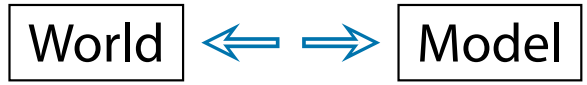
- **Unsupervised**
  - Clustering (e.g. detector noise modeling, triggering)



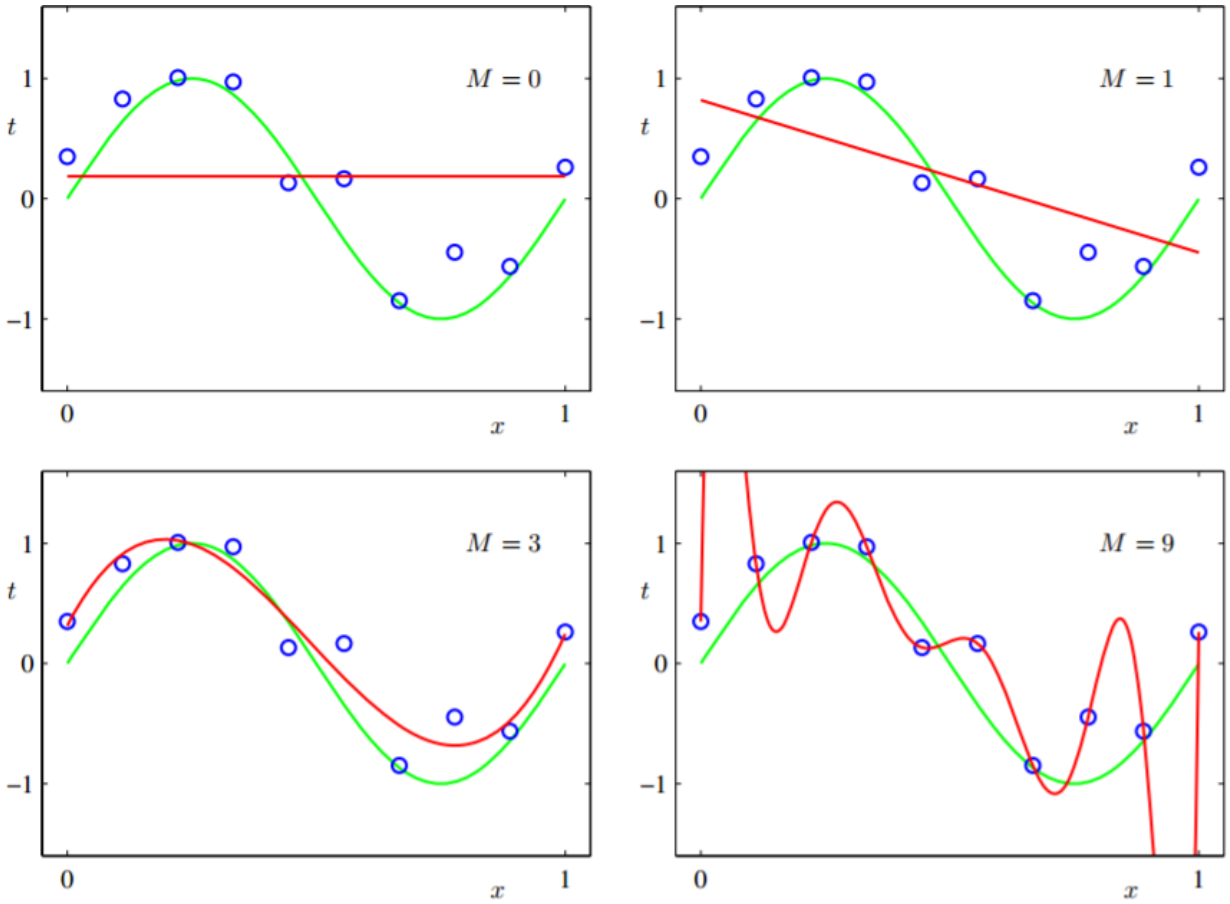
- **Semi-supervised**
  - Datasets too large to label (e.g. large scale translation, image labeling)



- **Reinforced**
  - Robotics, gaming



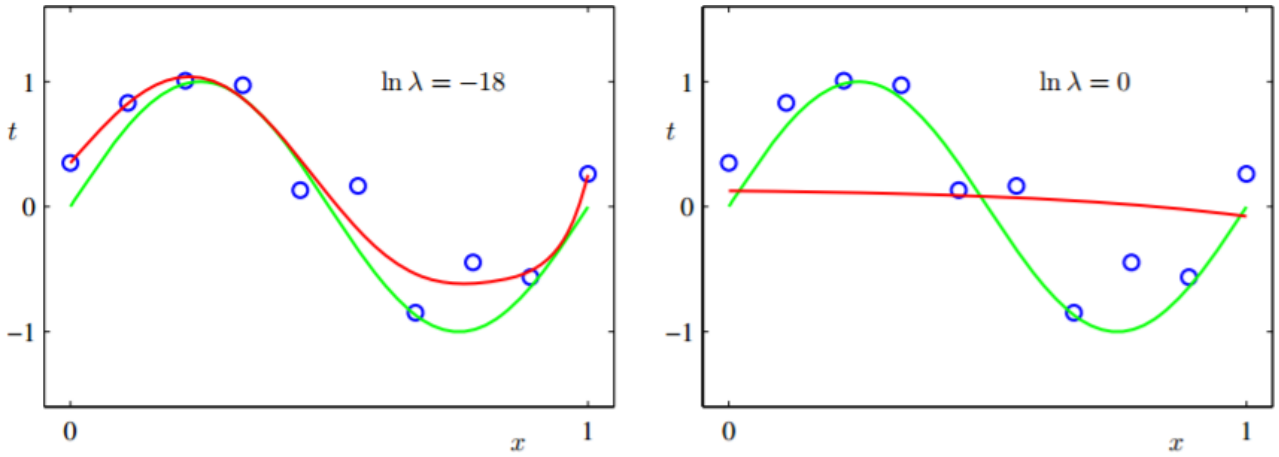
Overfitting, regularization



C. Bishop, Pattern Recognition and Machine Learning

- Machine “learning by heart” including fluctuations, sampling effects
- E.g. classical polynomial fit with too high order

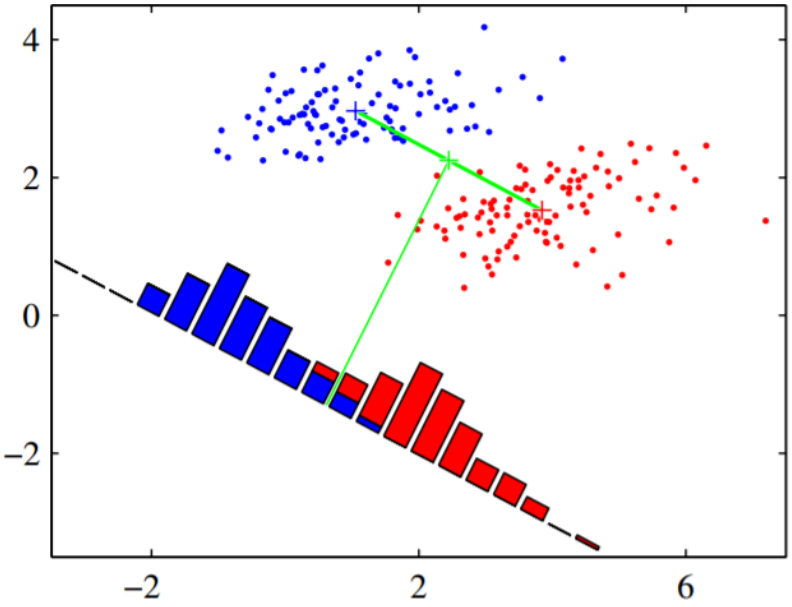
Overfitting, regularization

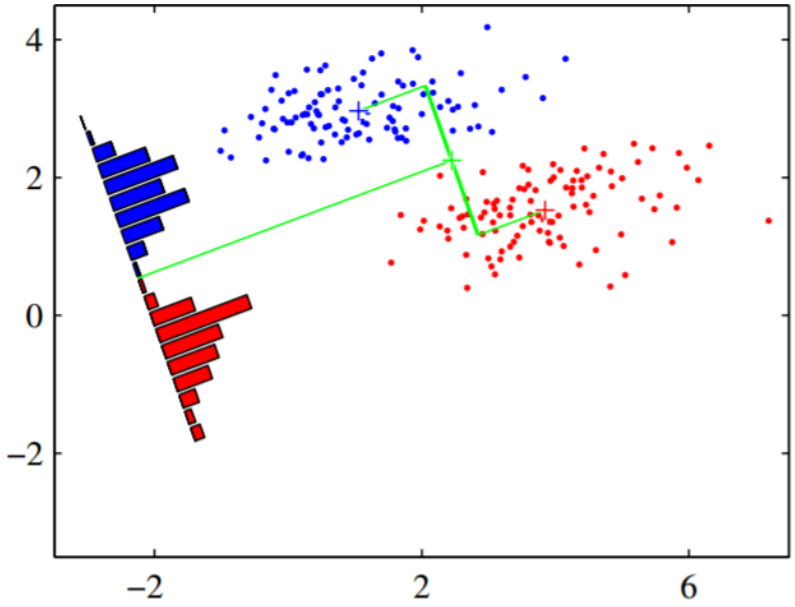


C. Bishop, Pattern Recognition and Machine Learning

- Old trick (Tikhonov, 1943): Add to $\sum \chi^2$ a "soft constraint" $\lambda \|w\|$ that penalizes too large coefficients/weights w
- Overregularization ($\lambda = 1$ "kills off" the ability of the function to fit)
- Useful everywhere if model can have the same or more degrees of freedom as data (you will see this in HEP/NP unfolding analyses, seismic exploration, etc.)

Simple ML: Fisher LDA





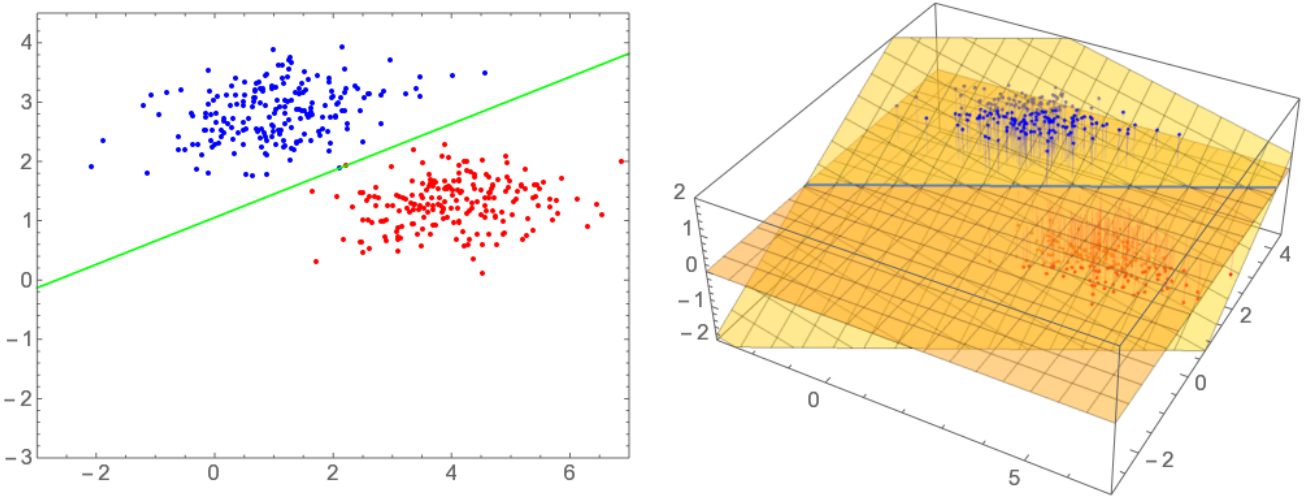
C. Bishop, Pattern Recognition and Machine Learning

Naïve class mean

Fisher LDA

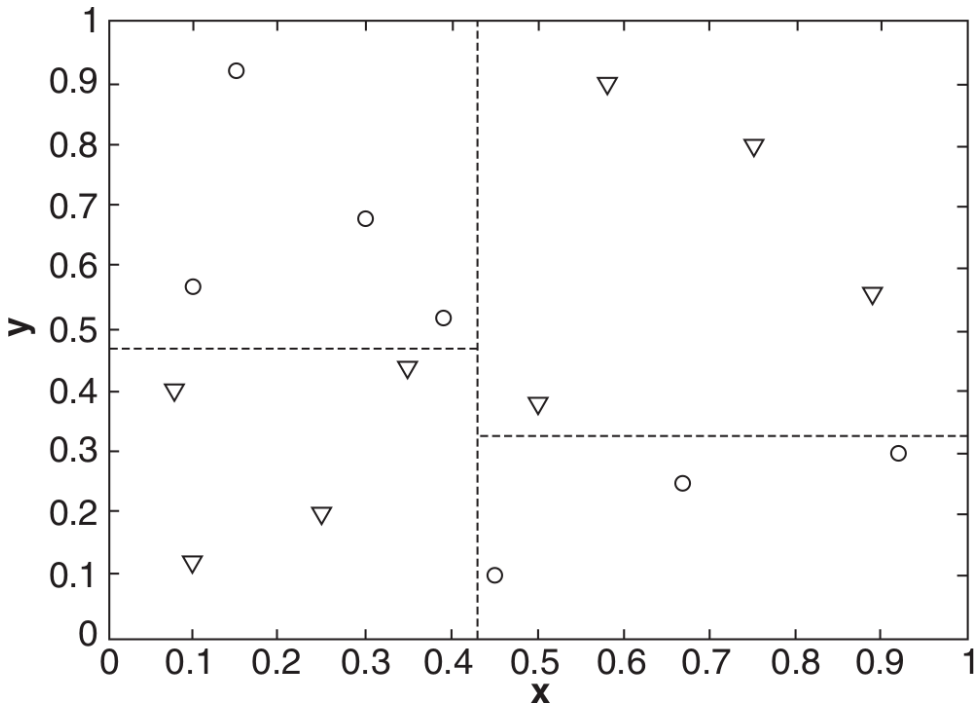
- Fisher linear discriminant analysis (LDA) calculates the hyperplane that optimally separates two Gaussian distributions
- "Separation" $S = \dfrac{\sigma^2_{\text{inter-class}}}{\sigma^2_{\text{intra-class}}} = \dfrac{(w \cdot (\mu_2 - \mu_1))^2}{w^T(\Sigma_1 + \Sigma_2)w}$
- Not equal the naïve plane orthogonal to the difference of class means
- Optimal only if the two classes are Gaussian

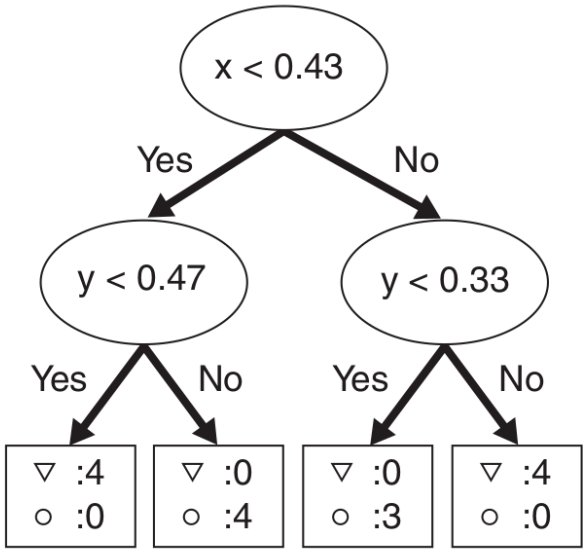
Supervised learning is function approximation



- Take the previous distribution, map blue into $(x, y, 1)$ and red into $(x, y, -1)$, least square fit to a plane
- The separating hyperplane is given by $f(x, y) = 0$
- Called logistic regression, behaves like the Fisher LDA (more robust for non-Gaussian cases)
- Classification is function approximation with categorical values

Decision trees

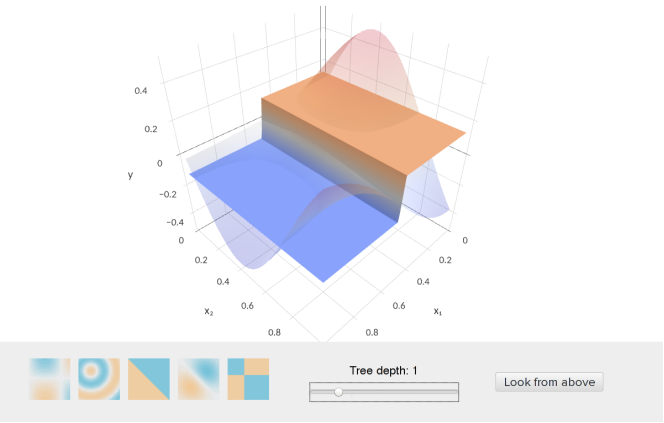


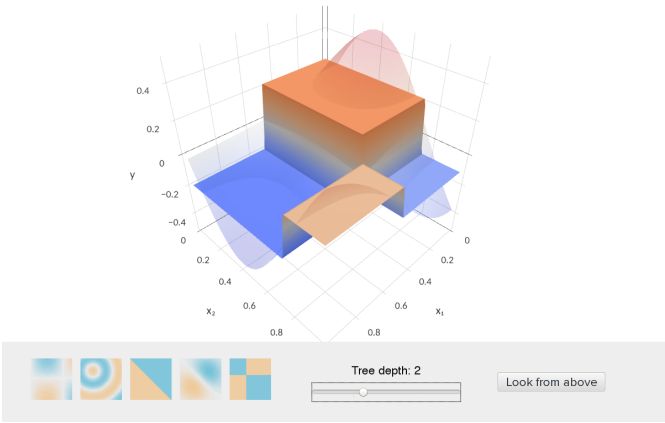


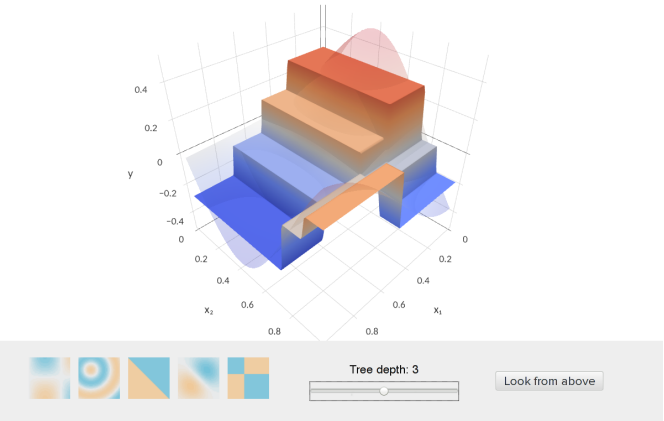
P.-n. Tan, M. Steinbach, V. Kumar, Introduction to Data Mining

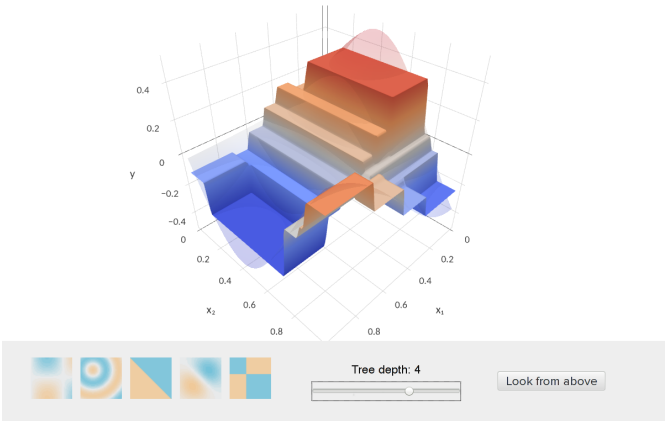
- So called weak classifier
- Regression/classification by binary decisions
- In the language of function approximation: Approximation by step functions
- Boosting/bagging allows trees/forests to have soft-decisions
- Splitting at each branch by information theory: Search parameter and take the axis/value giving maximum change to the distribution

Decision trees





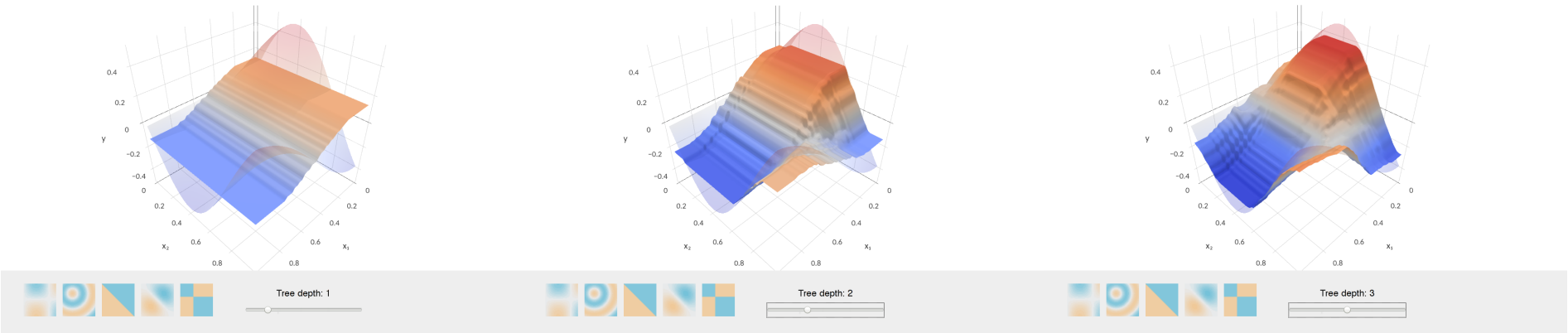




http://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html

- Result for 1 tree and 1–4 level of branches

Ensemble learning: Bagging/boosting

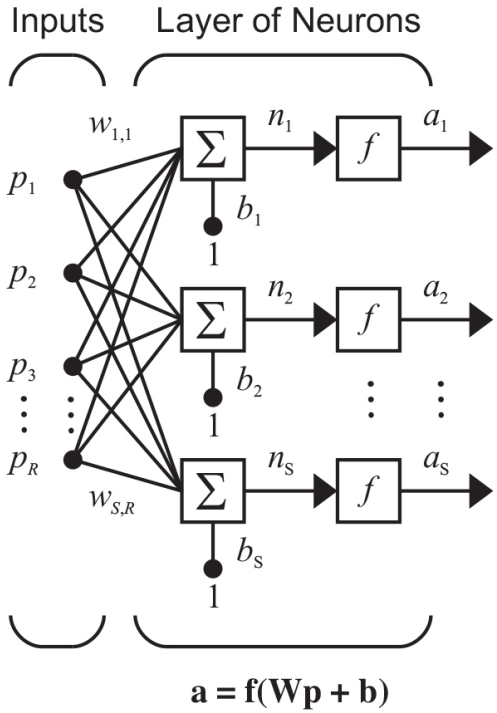


http://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html

- A "boosted" forest of 100 trees and 1–4 level of branches
- Bootstrap aggregating (bagging): Train using a random subset from the training samples, collect the ensemble of weak classifiers and average them
- A regularization technique
- Boosting: Train the *n*-th classifier with input weighted by misclassification of the previous steps
- Popular as "off-the-shelf" machine learning method: Not the best performance, but not much to tune
- Extremely popular in HEP/NP since MiniBooNE

Neural network

- Neural network (NN) or artificial NN is a computational graph consisting of:
  - Matrix multiplication with weights W
  - Adding biases b
  - Activation function f
- output $= f(W \text{ input} + b)$
- For each layer (signal-flow graph):



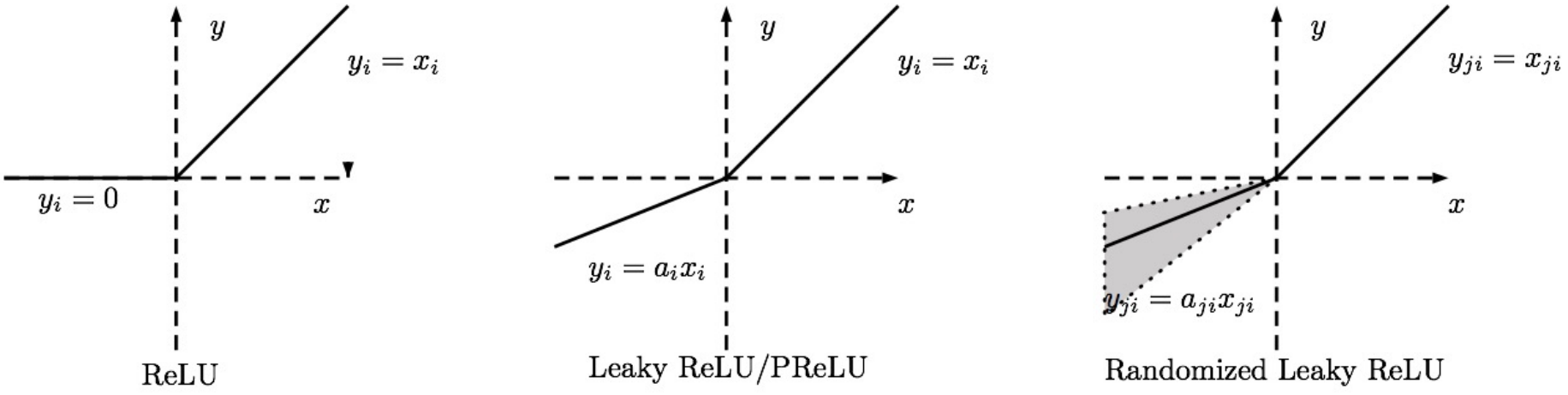
Where

R = number of elements in input vector

S = number of neurons in layer

https://www.mathworks.com/help/nnet/

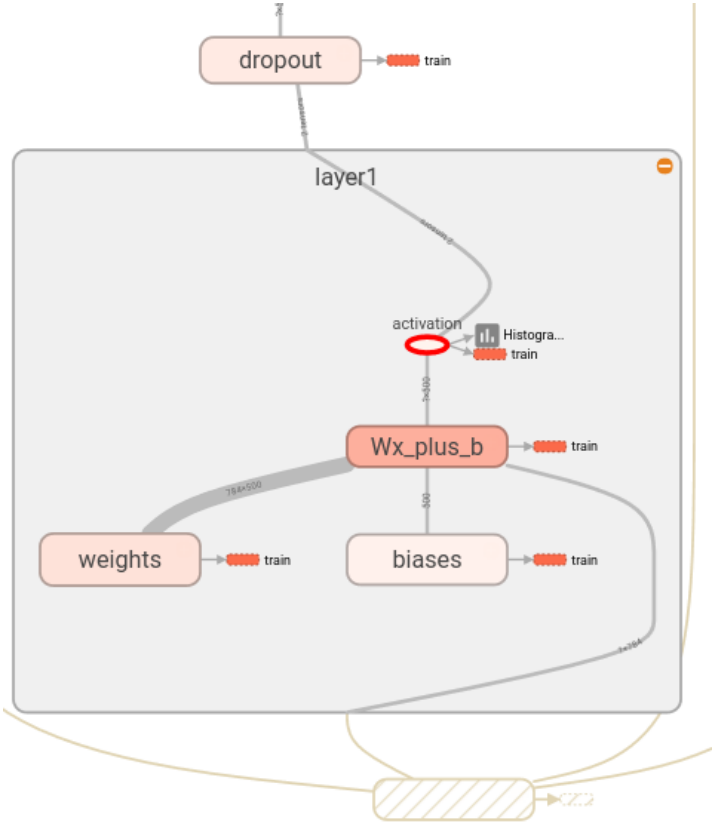
NN: Activation, regularization



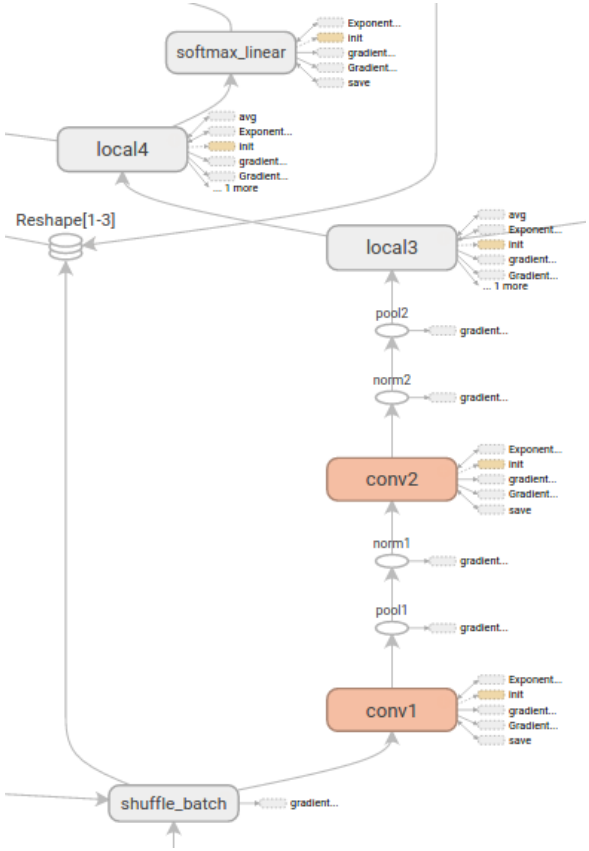
J.-x. Gu *et al.*, `https://arxiv.org/abs/1512.07108`

- Sigmoid and trigonometric activation functions less used these days
  - Biologically motivated
  - Vanishing gradient problem: Gradient ≈ 0 everywhere but for small input values
- Typical activation function today: a rectifier
  - Gradient $= 0$ only for "dead" neurons (sparse network)
- Regularization technique: "Dropout" (random deactivation of neurons during training) (G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, R. R. Salakhutdinov 2012)

NN: Computation graph



Single layer

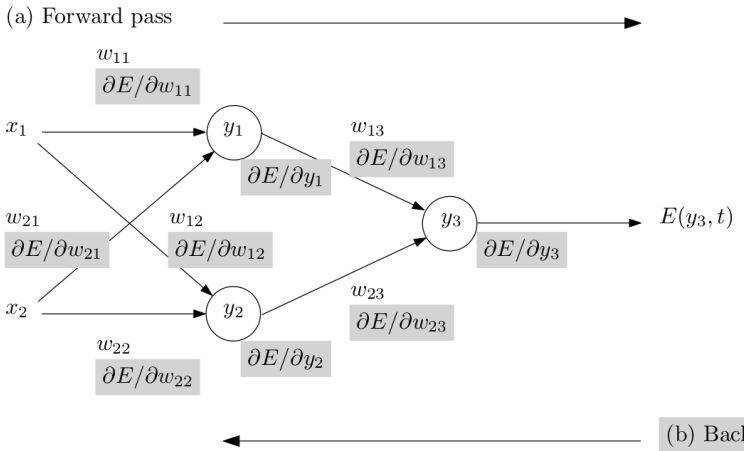


Multiple layers

https://www.tensorflow.org/get_started/graph_viz

- Multiple layers are stacked together, inner (not input/output) are called "hidden"
- "Deep" NN: hidden layers ≥ 2

NN: Automatic differentiation, compilation



Forward Evaluation Trace

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

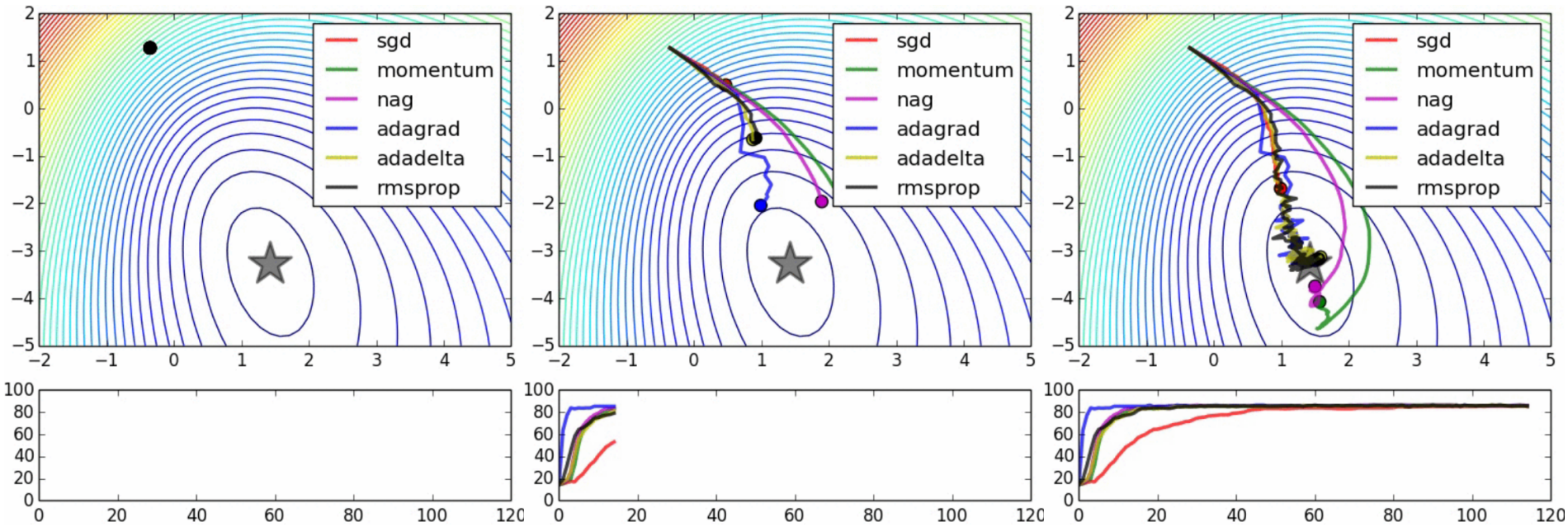
Reverse Adjoint Trace

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

A. G. Baydin *et al.*, `https://arxiv.org/abs/1502.05767`

- Optimization usually pays a high penalty if there is no analytic derivative
- Reverse automatic differentiation (AD)/backpropagation a trick to evaluate all partial derivatives (semi-symbolically)
- Widely used before machine learning: e.g. Speelpenning 1980 (UIUC PhD thesis), ADIFOR (C. Bischof, ANL/Rice 1986)
- Modern implementation for NN: Bergstra *et al.*, Theano: A CPU and GPU Math Expression Compiler, SciPy 2010
  - Compiles entire computation graph (+ derivatives) using GCC/Clang for CPU and CUDA for GPU

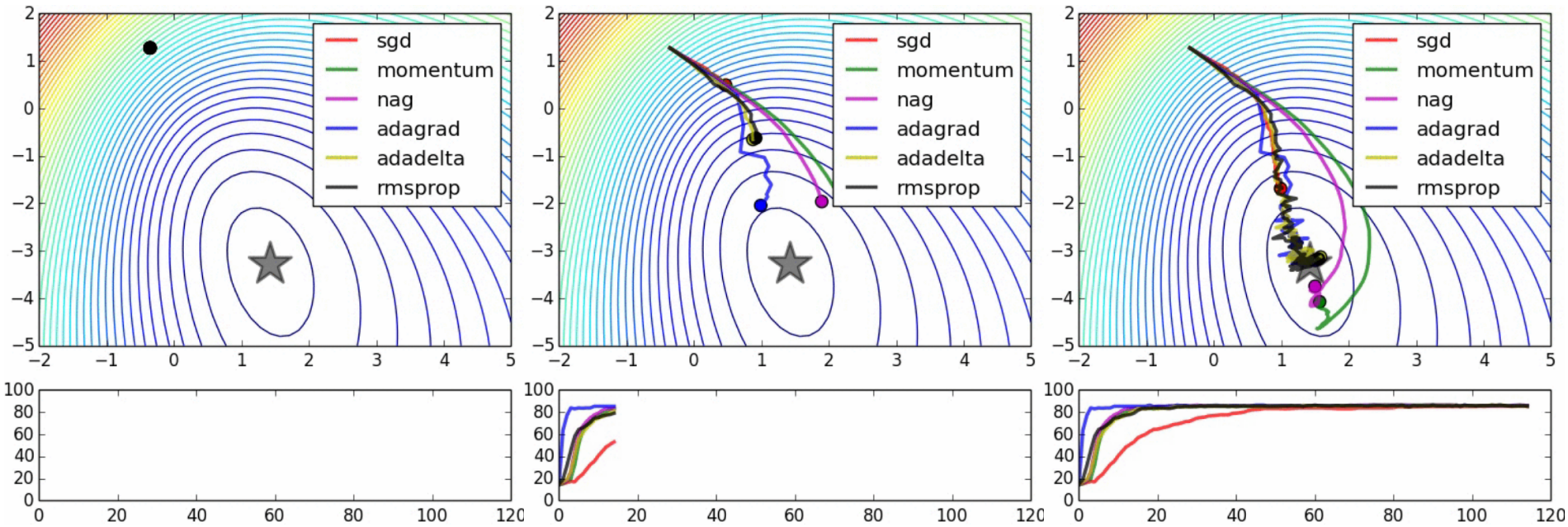
Optimization: Gradient descent



Alec Radford, https://imgur.com/s25RsOr

- Gradient descent: Go in the direction of the gradient of the loss function ($\times$ some step size)
- Stochastic GD (SGD): Approximate the loss function by a subset of the training sample ("batches")
- Can get stuck in saddle points forever

Optimization: Momentum, AdaGrad/AdaDelta

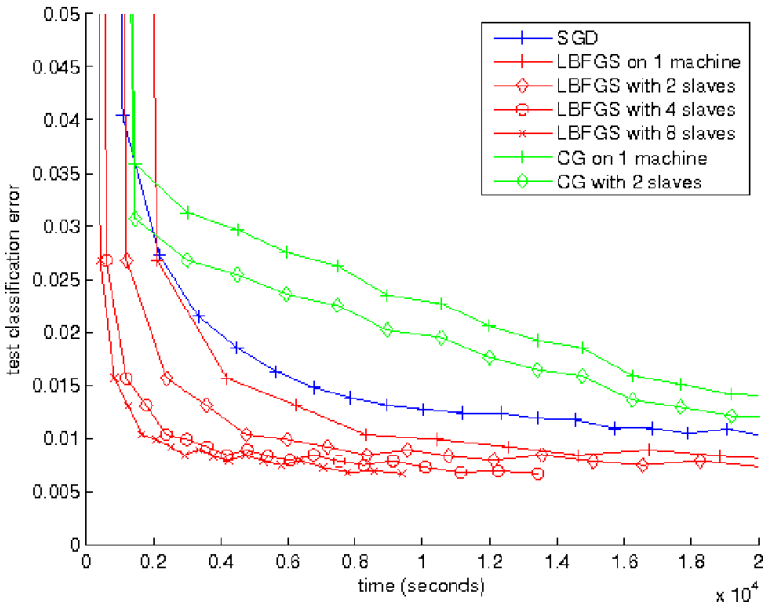


Alec Radford, `https://imgur.com/s25RsOr`

- Momentum: Add velocity, like a ball with mass rolling downhill
- NAG (Nesterov accelerated gradient): Jumps ahead and recalculate the gradient, check for overshooting.
- AdaGrad/AdaDelta (Duchi, Hazan, Singer, 2011): Keep a history of gradients and decrease learning rate in directions with a history of large gradients

Optimization: L-BFGS

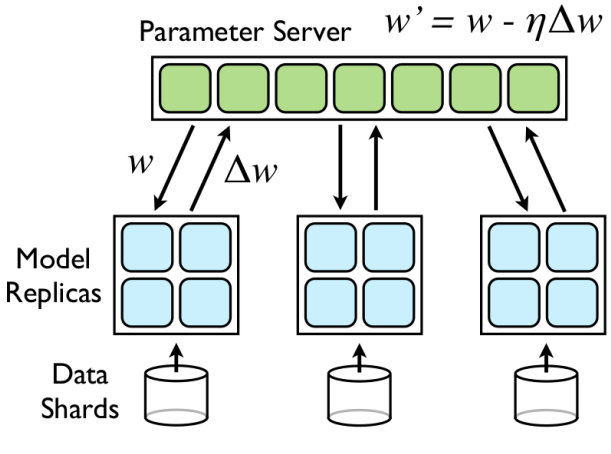
- Broyden–Fletcher–Goldfarb–Shanno, quasi-Newton method (high dimensional generalization of secant method)
- BFGS needs to keep the approximate Hessian in memory, take AlexNet with 60M parameters $\Rightarrow$ 15 PB memory needed
- Limited memory BFGS (L-BFGS): approximate the Hessian from the N last updates, keep only a history of updates and evaluate the direction on the fly
- Uses line search (multiple evaluation of function value/gradient) to avoid overshooting the minimum

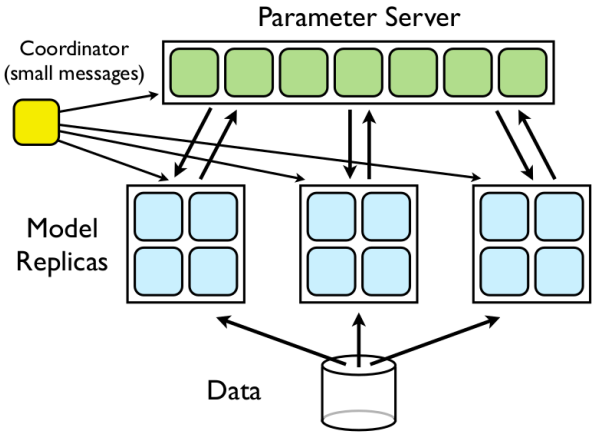


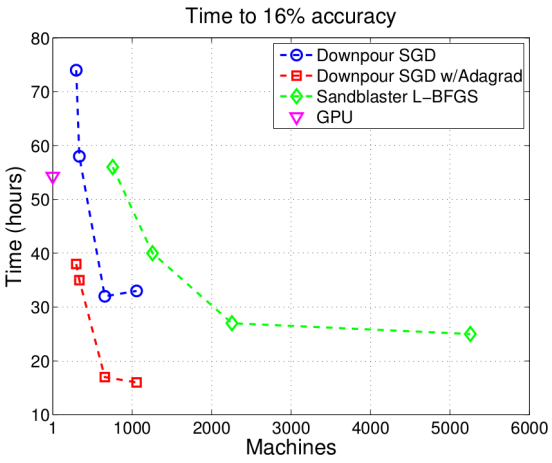
Q. V. Le *et al.*, On optimization methods for deep learning, ICML 2011

- Still difficult to apply to batches (need to see the entire data)
- Currently the cutting edge for large scale deep machine learning

Optimization: Data parallelism





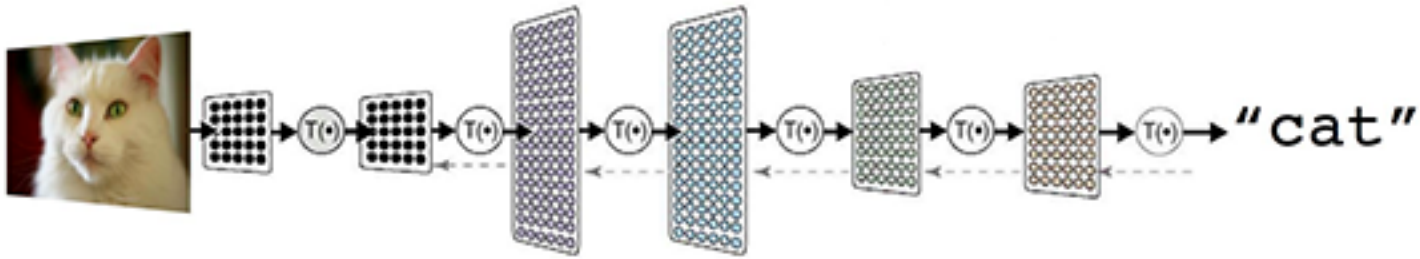


https://research.google.com/archive/large_deep_networks_nips2012.html

- Downpour SGD: Split SGD into different nodes and average their gradients, synchronize the new average from time to time
- Sandblaster L-BFGS: Use the fact that BFGS involves line searches, distribute the function/gradient evaluations operation among nodes
- Scaling difficult beyond $\approx 10^3$ nodes

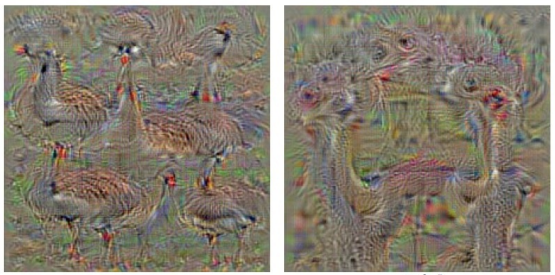
NN: Visualization

- A complete deep NN:

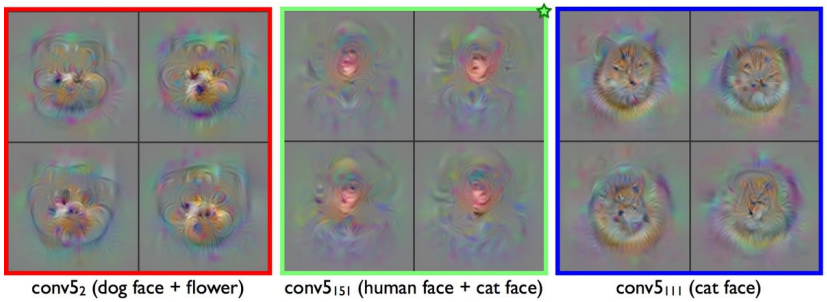


https://cloud.google.com/ml-engine/docs/quickstarts/datalab

- NN trained on images can be visualized by searching for a regularized (no extreme pixel fluctuation) maximum in its activation
- A way to see that neural networks are "universal approximators" to complex target functions
- From a network trained on ImageNet:



Select on keyword (final layer)



Select on intermediate neurons

http://yosinski.com/deepvis

NN: Universal approximation theorem

- G. Cybenko (1989) & K. Hornik (1991): Single hidden layer neural network (if large enough) is a universal approximator for bounded, nonconstant activation function
- Extended to unbounded functions used today by H. N. Mhaskar & C. A. Micchelli (1993)

$$
\begin{aligned}
n_1(x) &= Relu(-5x - 7.7) \\
n_2(x) &= Relu(-1.2x - 1.3) \\
n_3(x) &= Relu(1.2x + 1) \\
n_4(x) &= Relu(1.2x - .2) \\
n_5(x) &= Relu(2x - 1.1) \\
n_6(x) &= Relu(5x - 5)
\end{aligned}
$$

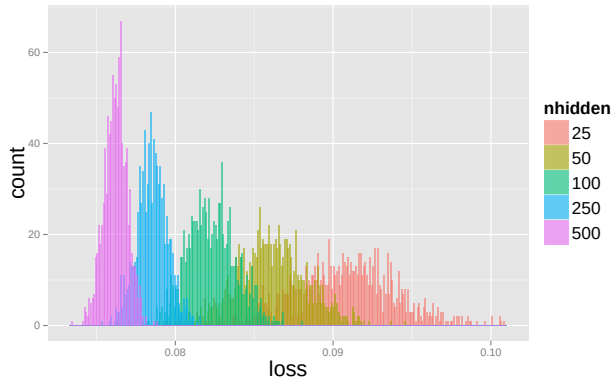
$$
\begin{aligned}
Z(x) = &-n_1(x) - n_2(x) - n_3(x) \\
&+ n_4(x) + n_5(x) + n_6(x)
\end{aligned}
$$

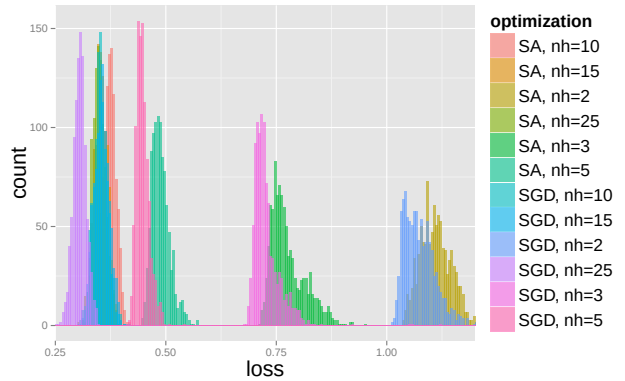
https://medium.com/towards-data-science/can-neural-networks-really-learn-any-function-65e106617fc6

NN: Loss surface

- Neural networks are closely related to the Hamiltonian of spin-glass models
- Around 2010 progresses in understanding local minima vs. ground state of the spin-glass models have been made by A. Auffinger, G. B. Arous, J. Černý (https://arxiv.org/abs/1003.1129)
- A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, Y. LeCun (2015) extended this to neural networks, and found:
  - Most local minima are equivalent
  - Probability of finding a "bad" local minimum decreases quickly with the size of the network
  - Global minima are actually "bad" in the sense they tend to be overfitting solutions

(SA = simulated annealing, a global optimization technique by including random/"thermal" motion)



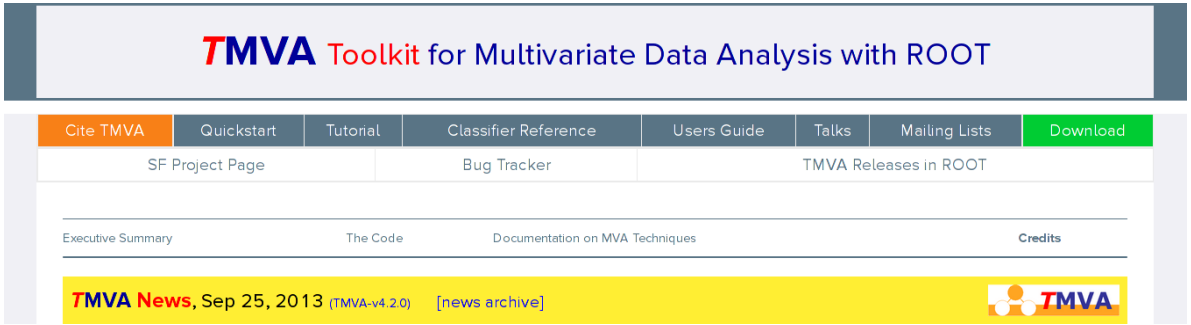


A. Choromanska *et al.*, https://arxiv.org/abs/1412.0233

What I did not discuss

- Non-parametric methods: k-nearest neighbor (kNN), naïve Bayes
- Kernel-based methods: Has high computational complexity ($O(n^2)$), and became "out of fashion" with deep neural networks
- Unsupervised learning: Autoencoder as fast Monte Carlo (where first principle Monte Carlo is expensive, e.g. cosmology)
- Recurrent neural networks, e.g. for time series
- Whole topic of reinforced learning

Tools



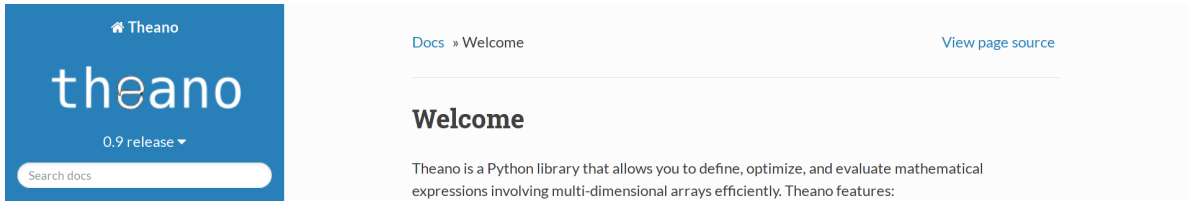
http://tmva.sourceforge.net/, https://root.cern.ch/

- Integrated into ROOT (and the only one)
- Most useful for e.g. LDA, BDT, NN is only useful for small networks, SVM is useless

http://scikit-learn.org/stable/

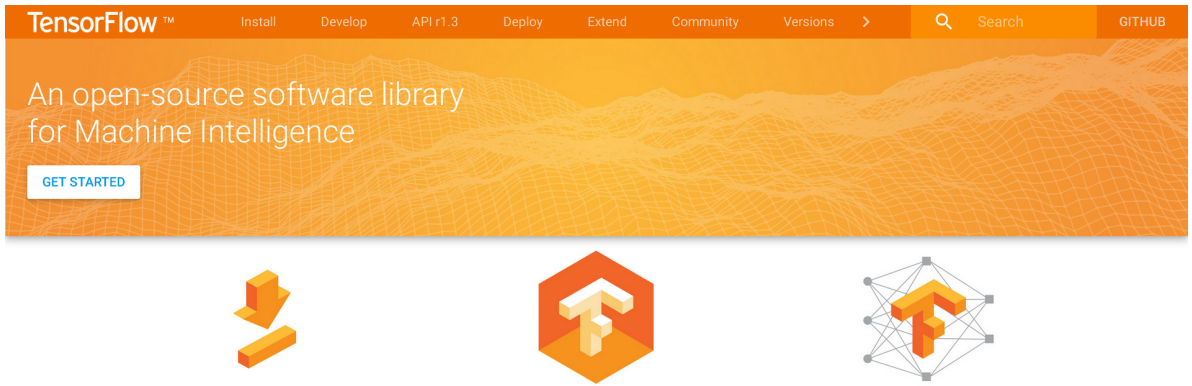
- Extremely large set of methods
- Not for large scale datasets

Tools



http://deeplearning.net/software/theano/

- Can do arbitrary computation graphs (not just NN), but widely used for NN
- Execution is fast, but compilation is slow (can take hours for large NN)
- Single node, but allows multiple GPU on a single node



https://www.tensorflow.org/

- Also arbitrary computation graphs, but mostly used for NN
- Execution slower than Theano, only recently with an experimental compiler (XLA)
- Distributed training across multiple nodes