Deep learning and its tempestuous (theoretical, and empirical) relationship with optimization theory's gradient descent

▽ the (misunderstood) gradient

Nabla the Gradient

James G. Shanahan [1,2]

[1]Church and Duncan Group, *[2]iSchool UC Berkeley, CA*

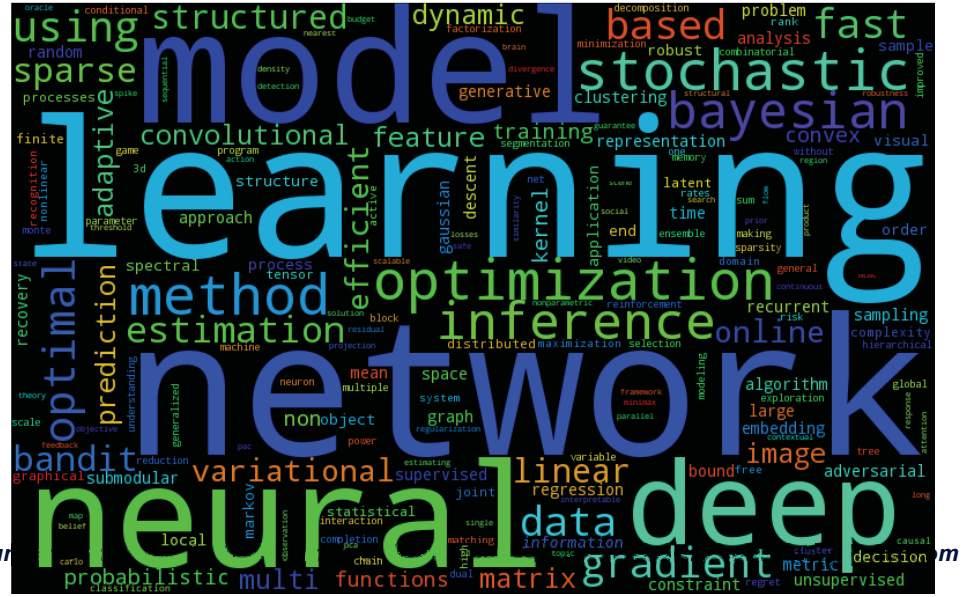
EMAIL: James_DOT_Shanahan_AT_gmail_DOT_com

Abstract

- This talk will present deep learning and its tempestuous (empirical and theoretical) relationship with optimization theory's gradient descent. Shakespeare might have structured such a talk as follows and used the lens of reverse mode autodiff to aid with understanding:
  - The prologue
  - Act 1 Tinker: Hack it up
  - Act 2 BackProp: theory to the rescue
  - Act 3 Layer by layer learning, a medieval pastime
  - Act 4 Introspection: better initialization and activation functions
  - Act 5 Express-laning the gradient: Skip Connections, the SoTA frontier (LSTMs, ResNet, Highway Nets, DenseNets)
  - The epilogue
- These five acts will be supported by examples and Jupyter notebooks in Python and TensorFlow. In addition, this talk will show how reverse mode autodiff provides an efficient and effective calculus framework that is transforming how we do machine learning.

Cast

- The network as itself
- ▽ (Nabla), the gradient as itself
- The neuron (and its nonlinear incarnations) as itself
- Autodiff (the relief cavalry)
- Supporting cast of concepts (e.g., loss function, SGD, etc.)



Executive producers

Donald Hebb
(1904–1985)
Developed a theory explaining how networks of neurons in the brain enable learning.

Marvin Minsky
(born 1927)
Developed one of the first artificial neural networks, but became a proponent of the symbolic approach to AI.

John McCarthy
(1927–2011)
Helped found the discipline of artificial intelligence; championed the use of mathematical logic in AI.

David Rumelhart
(1942–2011)
Developed simulations showing how neural processing could enable perception, and devised much more sophisticated neural networks.

Allen Newell
(1927–1992)
and **Herbert Simon**
(1916–2001)
Created the Logic Theorist, a program designed to mimic the logic skills of a human being.

James McClelland
(born 1948)
Founded the connectionist movement with Rumelhart, and cowrote the seminal work *Parallel Distributed Processing.*

Frank Rosenblatt
(1928–1971)
Created an electronic device called the Perceptron that simulated simple neural learning.

Geoffrey Hinton
(born 1947)
Developed techniques that allowed many more layers to be used in neural networks, creating the foundation for deep learning.

backpropagation, boltzmann machines

Geoff Hinton
Google

convolution

Yann Lecun
Facebook

stacked auto-encoders

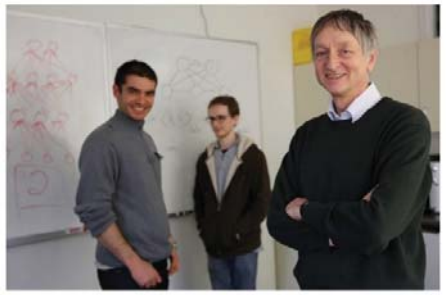
Yoshua Bengio
U. of Montreal

GPU utilization

Andrew Ng
Baidu

dropout

Alex Krizhevsky
Google



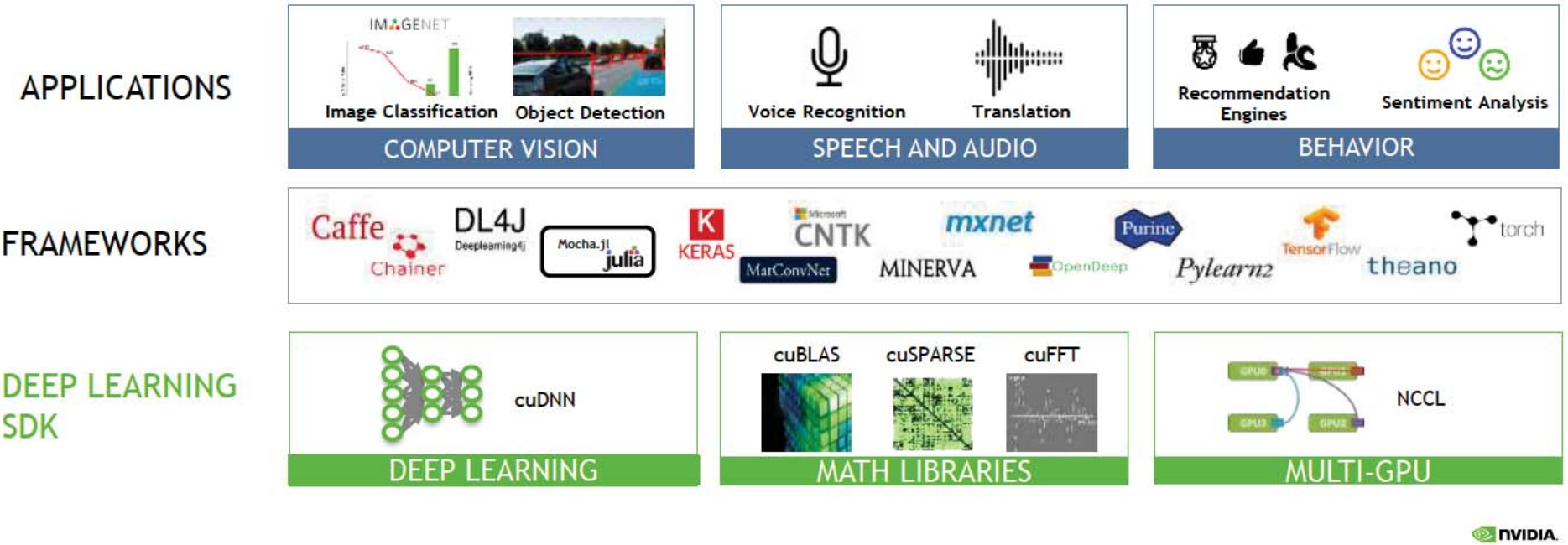
Ilya Sutskever, Alex Krizhevsky, Geoffrey Hinton

https://www.wired.com/2013/03/google_hinton/

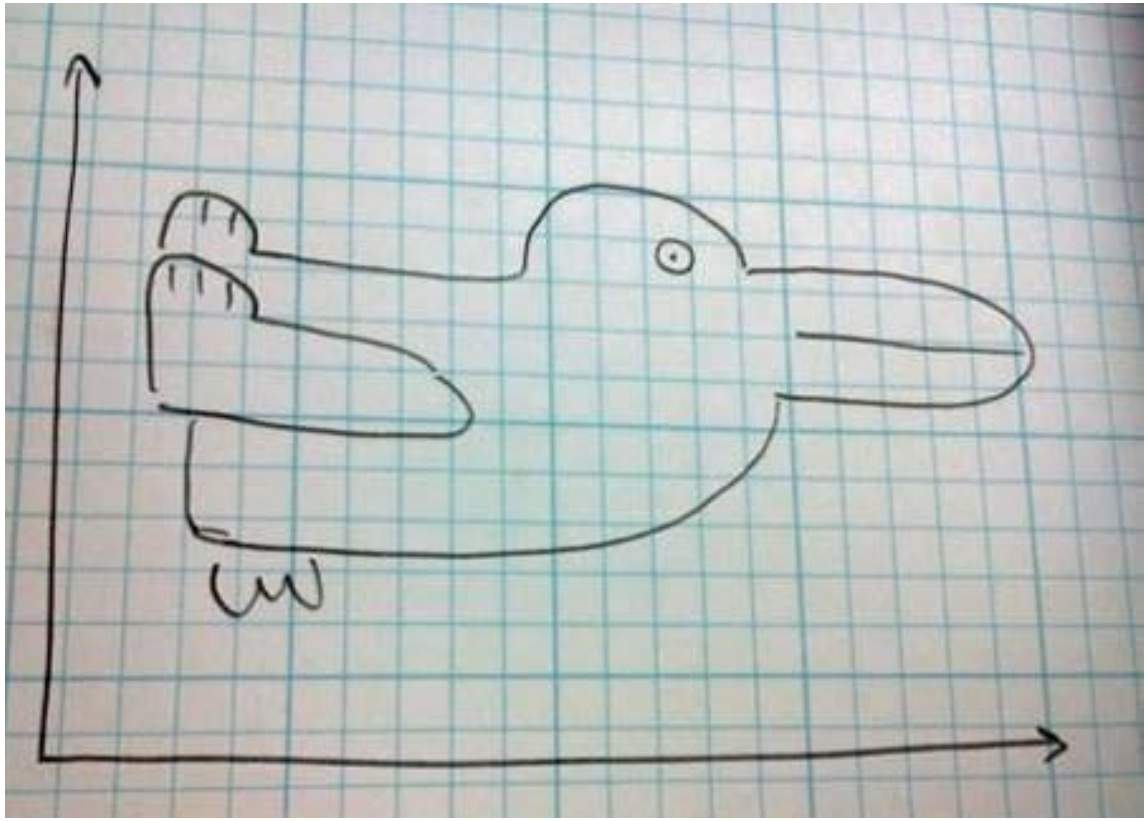
https://www.pinterest.com/pin/526921225134807091

Deep Learning supporting EcoSystem

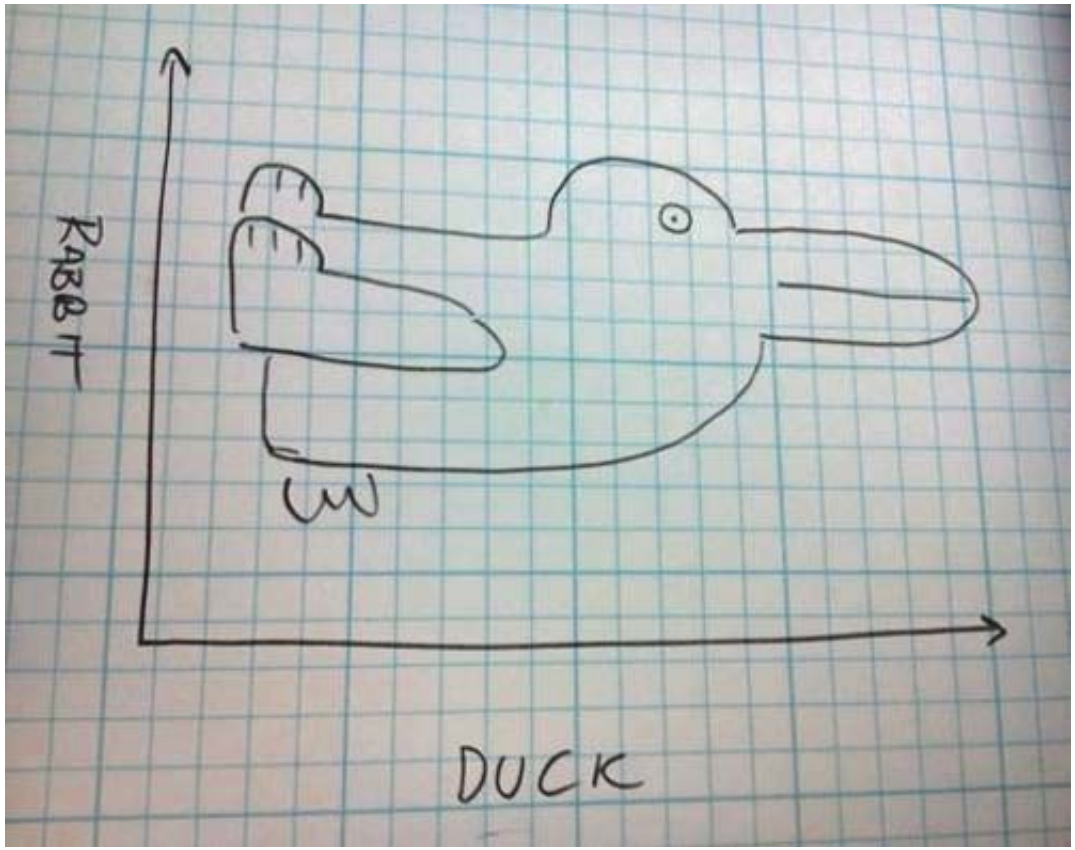
- **High Performance GPU-Acceleration for Deep Learning**



Different perspectives



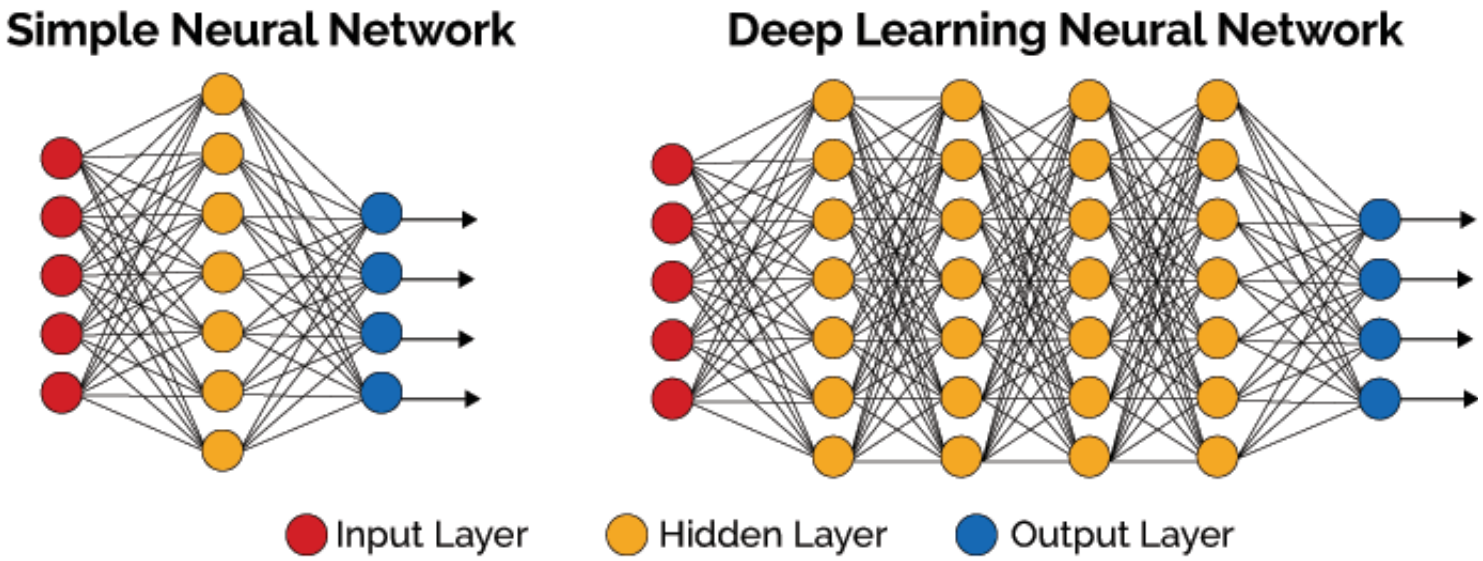
Different perspectives



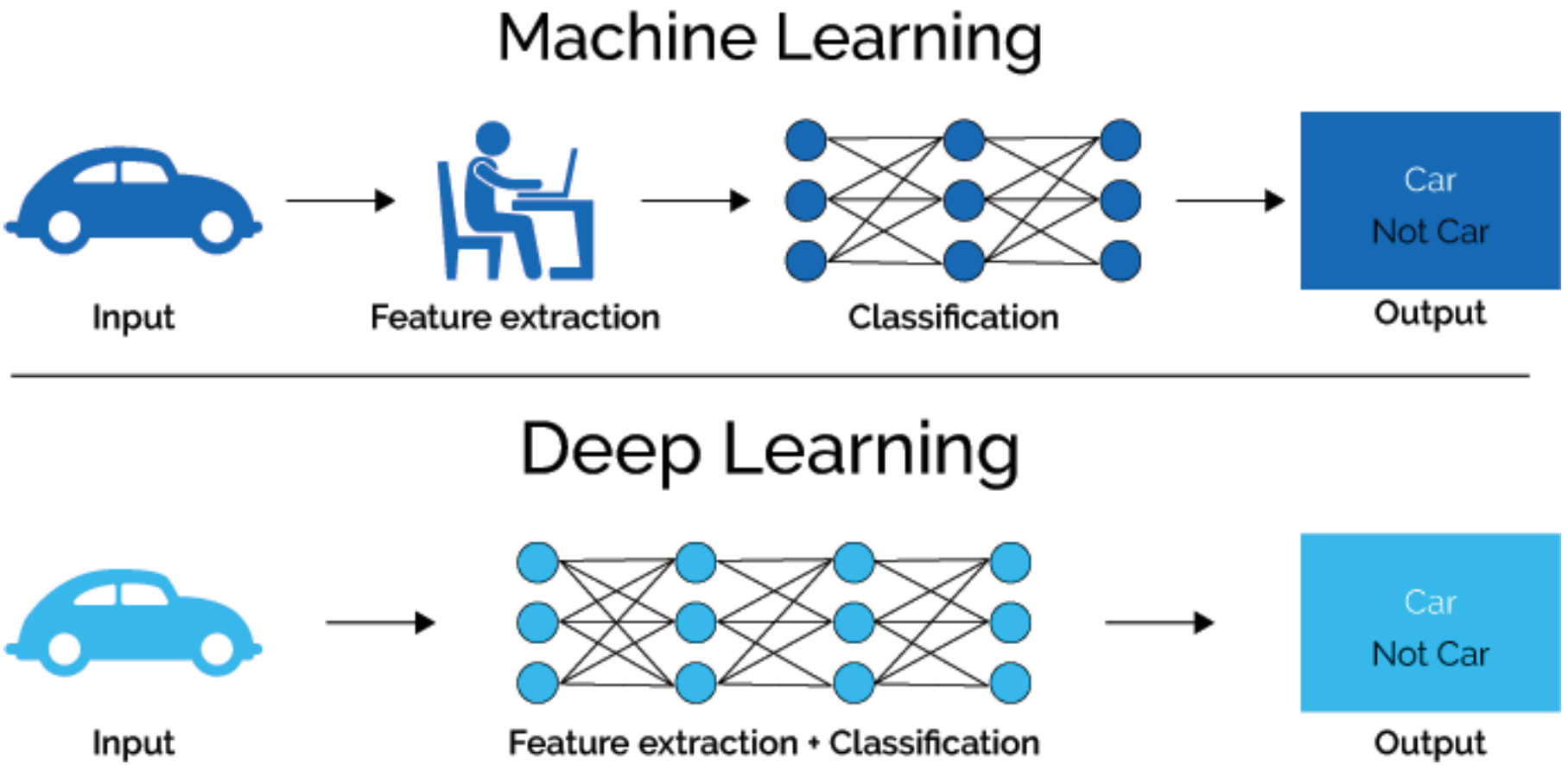
Quiz: What did you see first?

A: Duck
B: Rabbit
C: Both
D: Neither

Deep Learning: Deeper neural networks



Machine Learning a DL ecosystem



Deep Learning Syllabus (~60 Hour course) 1/2
Church and Duncan Group Inc.

PART 1: Introduction and Infrastructure [3-4 hours]

- Unit 0: Infrastructure setup (Local and on Azure cloud)
- Unit 1: Introduction: DNN hype cycle; five cat lives of BackProp

PART 2: Fundamental Core Concepts [3 days]

- Unit 2.1: Optimization Theory [2 hours]
- Unit 2.2: Linear Regression from scratch via optimization theory [2 hours]
- Unit 3: Classification from scratch: KNN, Perceptron, Linear classification via gradient descent, SoftMax [1 Day]
- Unit 4: Neural Networks Generation 1 and 2: from no backprop to backprop [1 day]
- Unit 5: Neural Networks Generation 3: Deep Learning: core concepts [1 day]

http://churchandduncan.com/

PART 3: Deep Learning Specializations [3 days]

- Unit 6: Convolutional Neural Networks (CNNs) and computer vision apps
- Unit 7: Embeddings, and apps using CNNs in Text Classification
- Unit 8: Recurrent Neural Networks, LSTMs/GRUs, Text Processing
- Unit 9: TensorFlow for X

Deep Learning training(~60 Hour course)
Provided by Church and Duncan Group

Deep Learning Syllabus (~60 Hour course) 2/2
Church and Duncan Group Inc.

PART 4: Applications/Case Studies [3 days]

- Unit 10: Time series and forecasting
- Unit 11: Natural language processing
- Unit 12: Healthcare and Advertising
- Unit 13: Speech recognition

…… Chatbots, computer vision, object detection, credit card number extraction

PART 5: Advanced/Recent topics

- Unit 14:Generative Adversarial networks
- Unit 15:Variational Autoencoders (VAEs)
- Unit 16:Custom architectures and custom loss functions

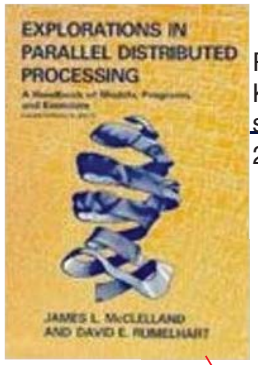
http://churchandduncan.com/

PART 6: Projects

- Unit 17:Project proposals, planning, reporting,
- Unit 18: Wrapup

Deep Learning training(~60 Hour course)
Provided by Church and Duncan Group

Deep Learning milestones



Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. The Journal of Machine Learning Research, 12:2493–2537, 2008, 2011.

Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations, Honglak Lee, Roger Grosse, Rajesh Ranganath, Andrew Y. Ng, ICML 2009

AlphaGo beats a Go World Champion

Mohamed, Dahl and Hinton beat a well-known speech recognition benchmark (TIMIT)

Krizhevsky, Sutskever, and Hinton win the ImageNet object recognition challenge.

Hochreiter and Schmidthuber invent LSTM for recurrent networks with long memory.

Perceptron learning alg.

Perceptron

Resnet 2015

1943 1957 1986 1997 1998 2002 2009 2010 2011 2016

Hinton, Rumelhart and Williams

LeCun, Bottou, Bengio and Haffner publish CNN for Computer Vision

Salakhutdinov and Hinton discover a method to train very deep neural networks.

Neural winter

Microsoft and Google demonstrate breakthrough results on large vocabulary speech recognition.

Understanding the difficulty of training deep feedforward neural networks, by Glorot and Bengio, 2010

~1700s • The prologue

1940-1970 • Act 1 Tinker: Hack it up

1970-1995 • Act 2 BackProp: theory to the rescue

1995-2007 • Act 3 Layer by layer learning, a medieval pastime

2007-2015 • Act 4 Introspection: better init. and activation functions

2015 - • Act 5 Express-laning the gradient: Skip Connections, the SoTA frontier (LSTMs, ResNet, Highway Nets, DenseNets)

• The epilogue

▽ the gradient chorus

- **What is the gradient for linear regression?**

- **Chorus**
  - The gradient is the weighted sum of the training data, where the weights are proportional to the error (for each example) !

weight *example*

$$\frac{\partial E}{\partial W} = (O - t)\,\boldsymbol{X}$$

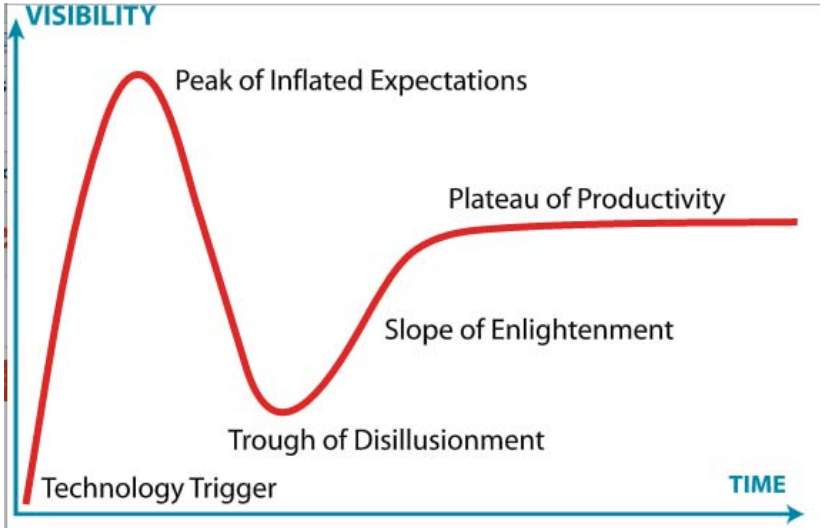
$$W = W - \alpha \times \frac{\partial E}{\partial W}$$

$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} \mathbf{34} \\ \mathbf{1} \\ \mathbf{4} \end{bmatrix}$$

$$4.32 = 5 - 0.01 \times (2 \times 34) \text{ for } i = 1$$

AI Hype

- **1960s AI**
  - Top-down
- **1980s AI**
  - Top-down
- **2010s AI**
  - Bottom up, data driven



~1700s • The prologue

1940-1970 • Act 1 Tinker: Hack it up

1970-1995 • Act 2 BackProp: theory to the rescue

1995-2007 • Act 3 Layer by layer learning, a medieval pastime

2007-2015 • Act 4 Introspection: better init. and activation functions

2015 - • Act 5 Express-laning the gradient: Skip Connections, the SoTA frontier (LSTMs, ResNet, Highway Nets, DenseNets)

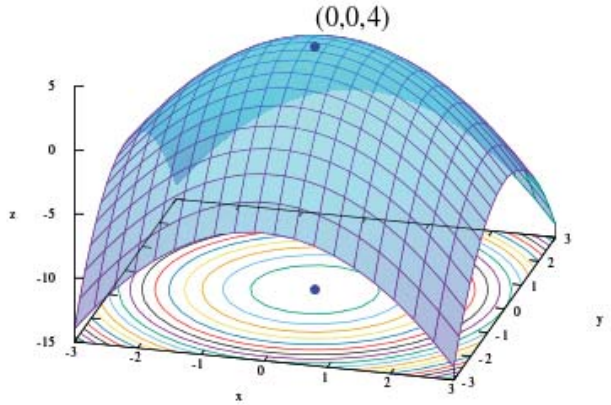
• The epilogue

Unconstrained optimization

- **Find the maximizer of the a function**

Maximize $f(x, y) = -(x^2 + y^2) + 4$.

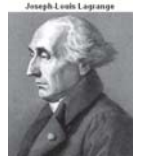
Contour map



Graph of a paraboloid given by $z = f(x, y) = -(x^2 + y^2) + 4$. The global maximum at $(x, y, z) = (0, 0, 4)$ is indicated by a blue dot.

Historical development 1/2

- Isaac Newton (1642-1727): differential calculus, methods of optimization
- Joseph-Louis Lagrange (1736-1813): Calculus of variations, minimization of functionals, method of optimization for constrained problems
- Augustin-Louis Cauchy (1789-1857): Solution by direct substitution, steepest descent method for unconstrained optimization
- Leonhard Euler (1707-1783): Calculus of variations, minimization of functionals
- Gottfried Leibnitz (1646-1716): Differential calculus methods of optimization)



Historical development 2/2

- George Bernard Dantzig (1914-2005)
  (Linear programming and Simplex method (1947))

- Richard Bellman (1920-1984): (Principle of optimality in dynamic programming problems)

- Harold William Kuhn (1925-): (Necessary and sufficient conditions for the optimal solution of programming problems, game theory)

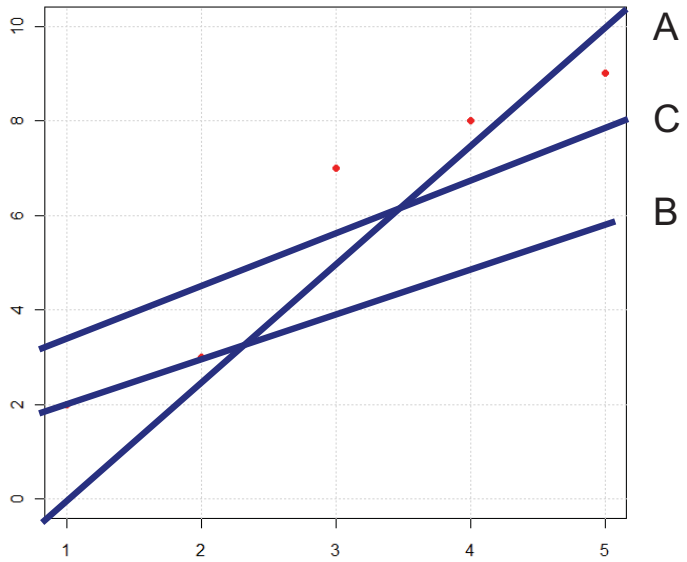
- Albert William Tucker (1905-1995): (Necessary and sufficient conditions for the optimal solution of programming problems, nonlinear programming, game theory: his PhD student was John Nash)

- Von Neumann (1903-1957): (game theory)

Least Square Fit Approximations

Suppose we want to fit the data set.

#	x	y
1	1	2
.	2	3
.	3	7
.	4	8
m	5	9



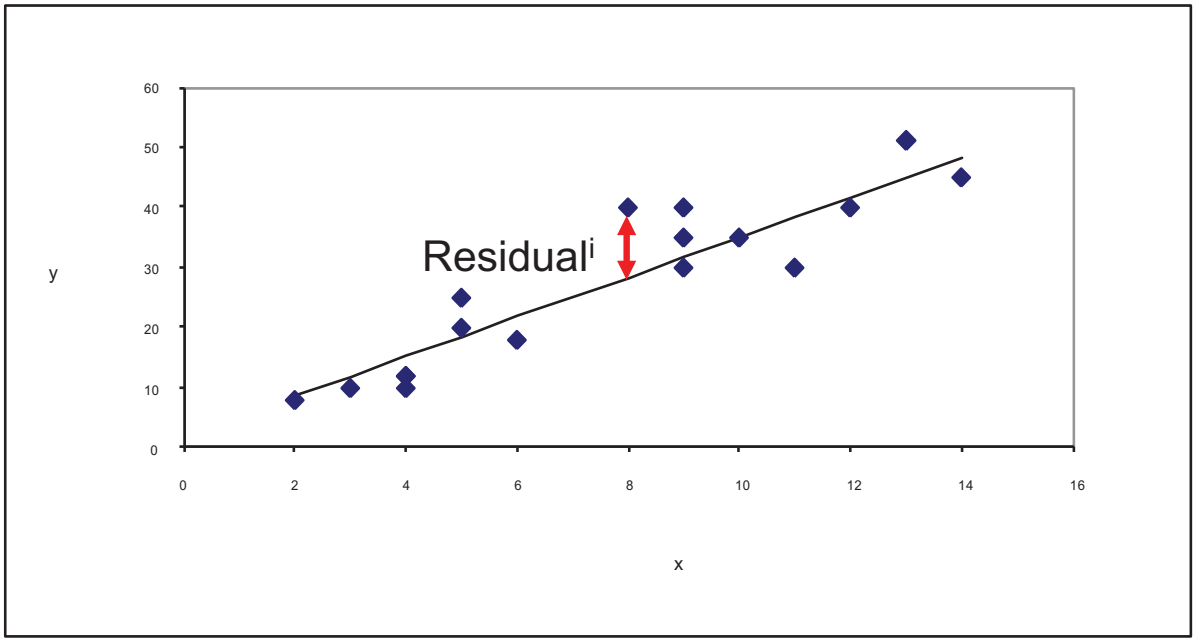
We would like to find the best straight line to fit the data?

$$y = mx + b$$

Residual → Mean Squared Error

$$\text{Residual}^{i} = \left(WX^{i} - y^{i}\right)$$

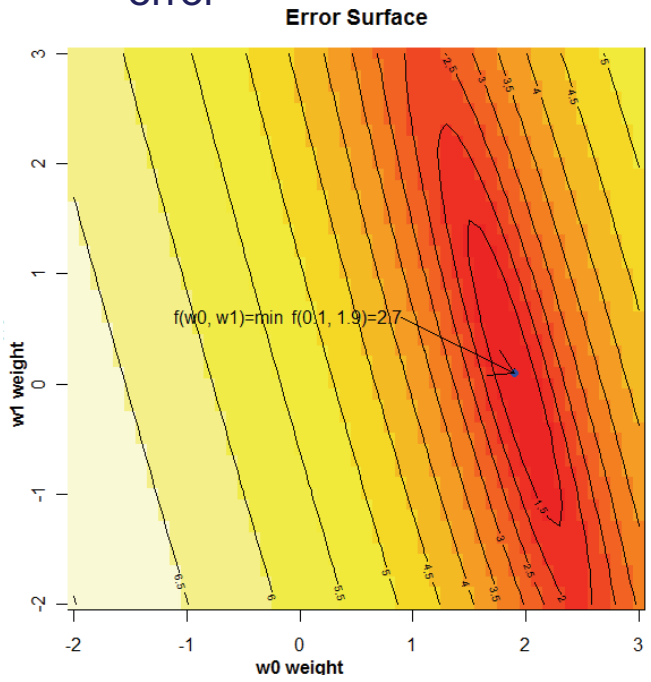
$$J(W) = \frac{1}{2}\sum_{i=1}^{m}\left(WX^{i} - y^{i}\right)^{2}$$



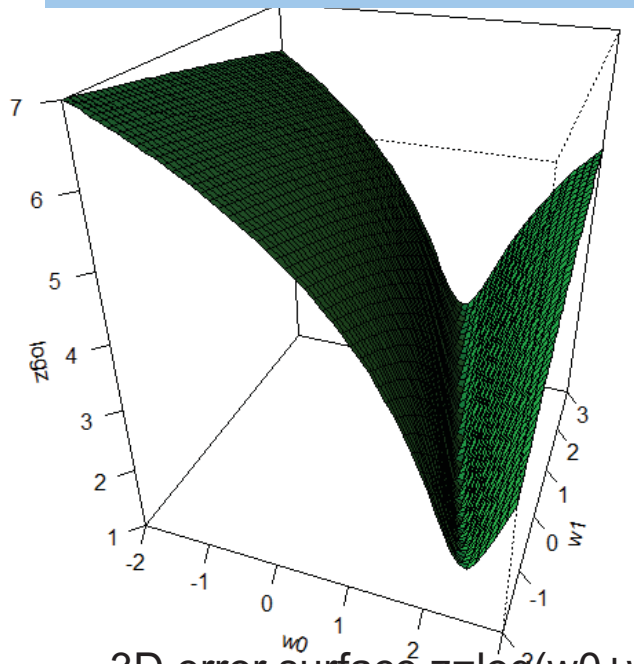
Brute Force Search of Weights

- Enumerate all possible coefficient combinations (in our case coefficient for the y-intercept (bias) and for the c-variable (time)
- Select the weight combination that minimizes the sum of square error

example.OLS_Heatmap()

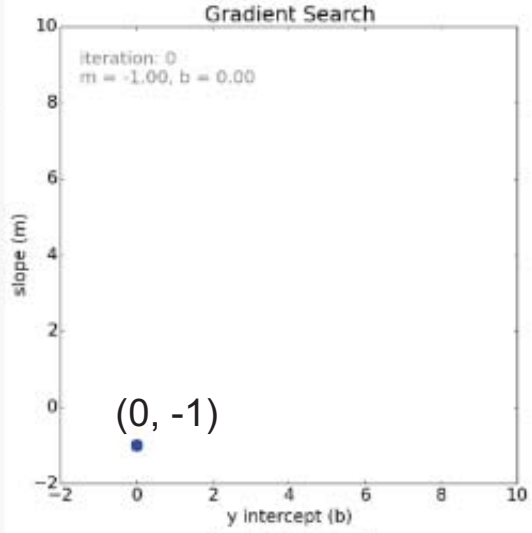


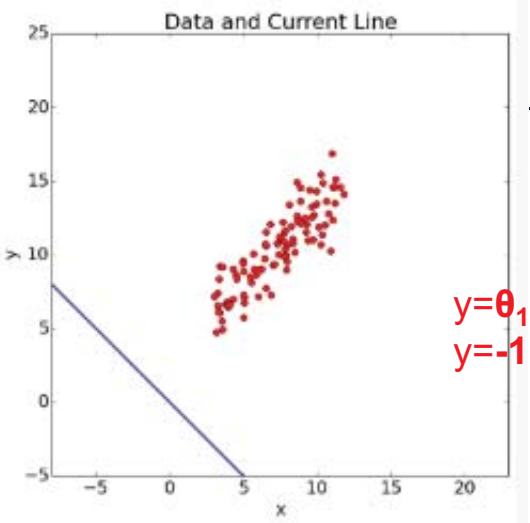
HeatMap with isolines overlayed



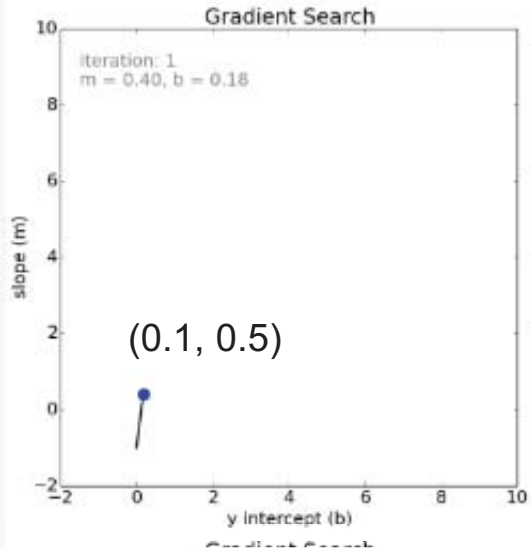
3D error surface z=log(w0+w1∗x)

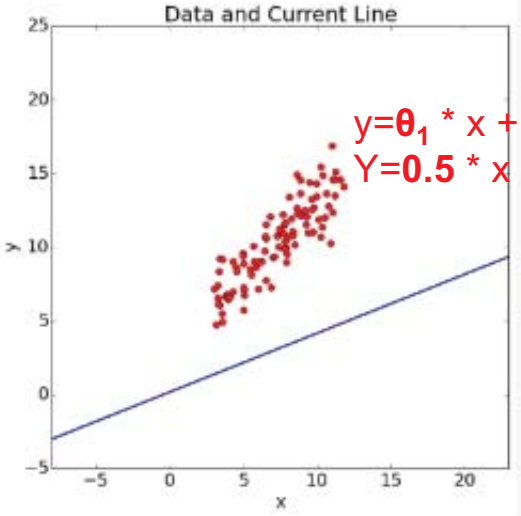
Random





$y=\theta_1 * x + \theta_0$
$y=-1 * x + 0$





$y=\theta_1 * x + \theta_0$
$Y=0.5 * x + 0.1$

Gradient Descent for LR Price~Size Iteration 0

Eqn Line y= aX + b

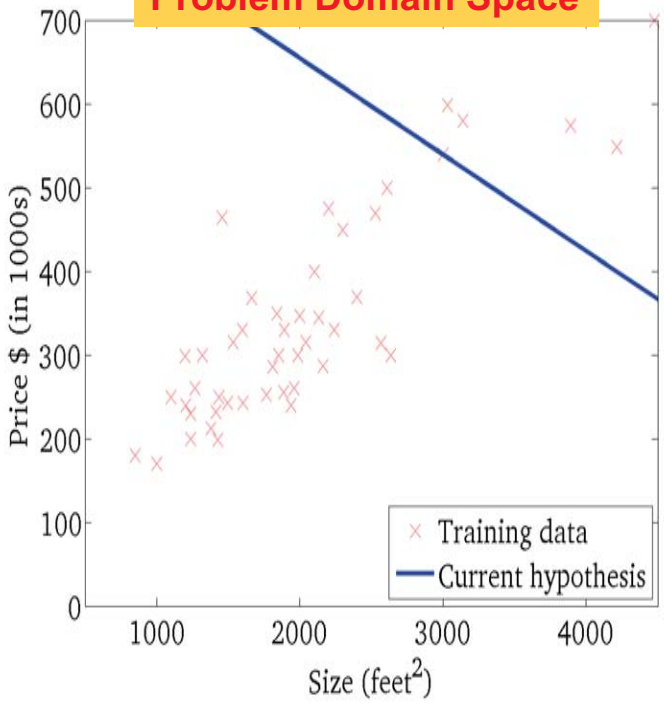
Y = w1X + W0 1

(for fixed θ_0, θ_1 this is a function of x)

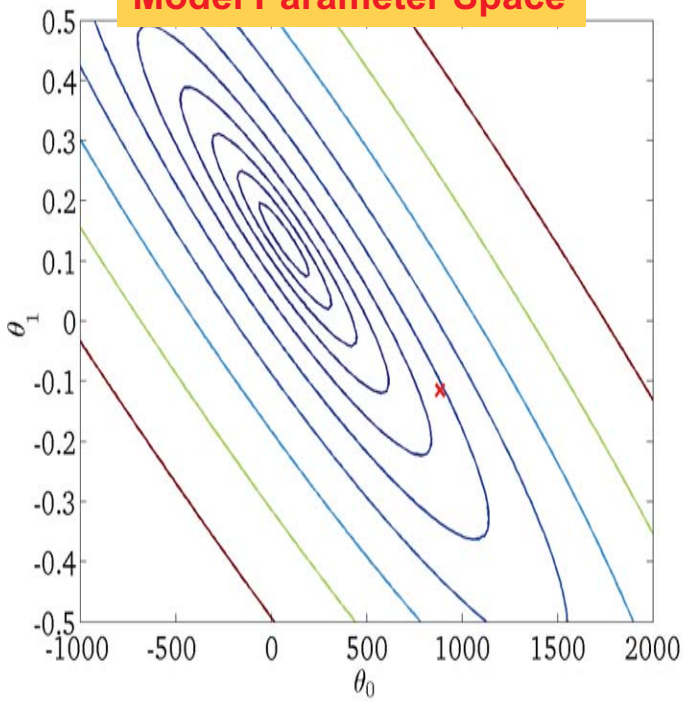
$J(\theta_0, \theta_1$ $J_q(W, X_1^L) = \frac{1}{m}\sum_{i=1}^{m}(W^T X_i - y_i)^2$

(function of the parameters θ_0, θ_1)

Problem Domain Space

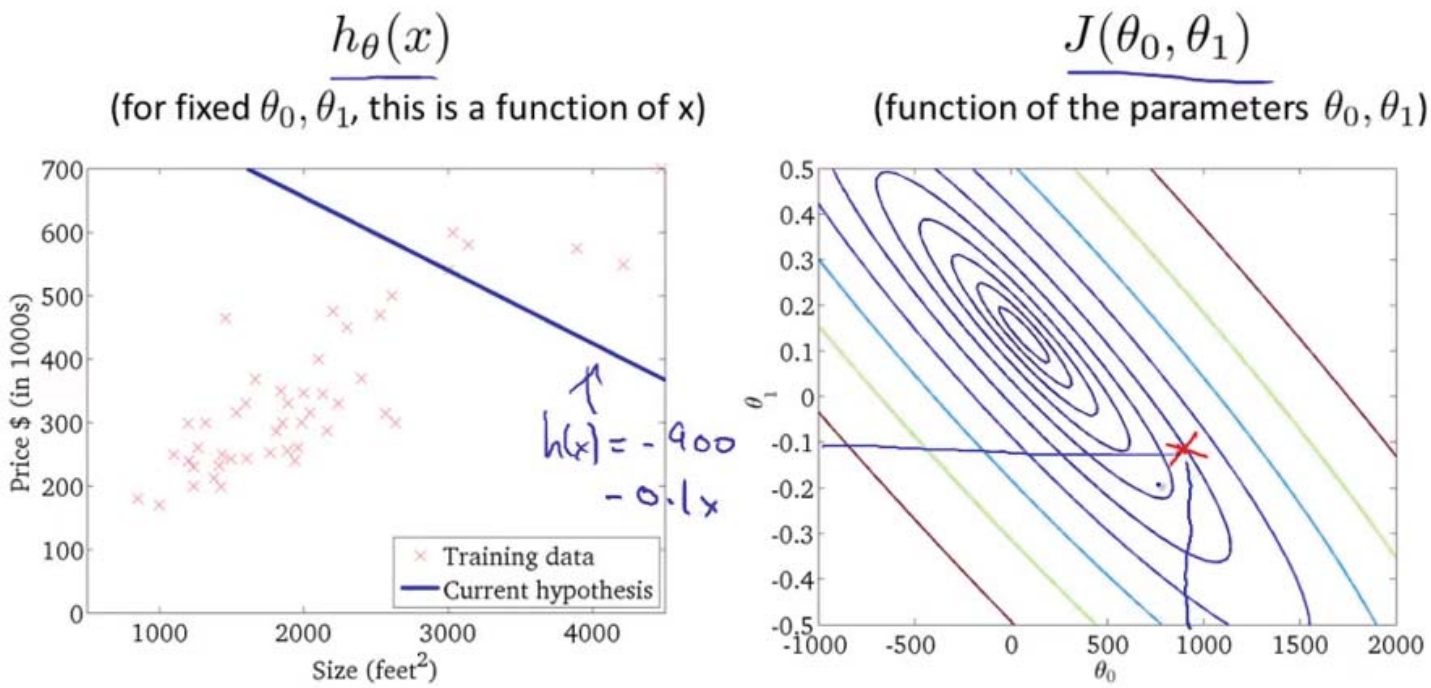


Model Parameter Space



Gradient Descent for LR Price~Size – Iteration 0

- .

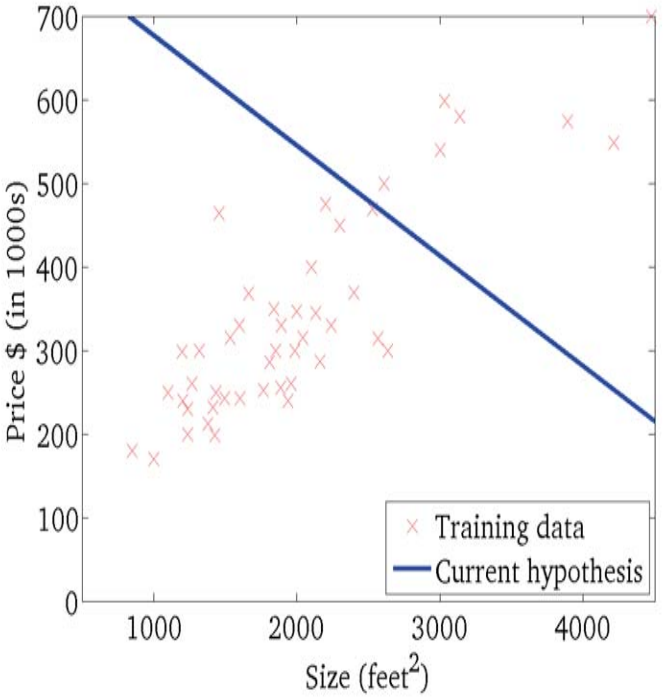


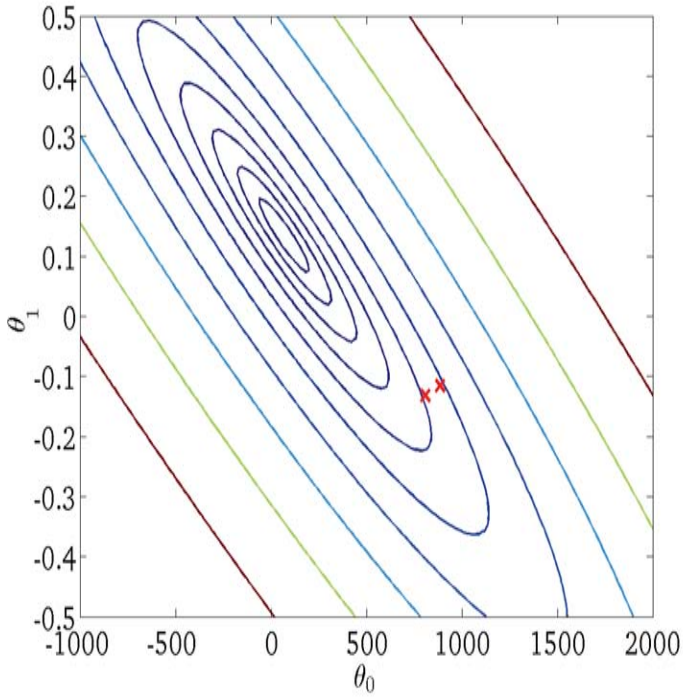
Andrew Ng

Gradient Descent for LR Price~Size – Iteration 1

$h_\theta(x)$ (for fixed θ_0, θ_1 , this is a function of x)

$J(\theta_0, \theta_1)$ (function of the parameters θ_0, θ_1)





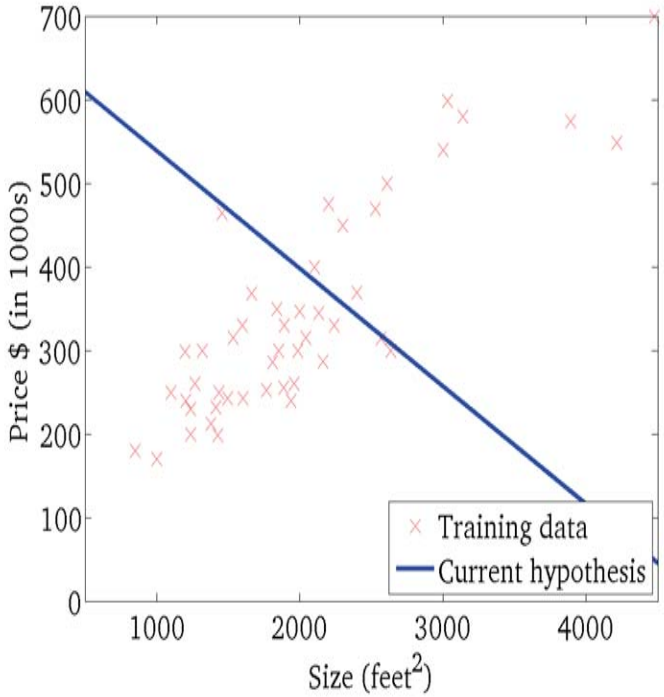
Gradient Descent for LR Price~Size – Iteration 2

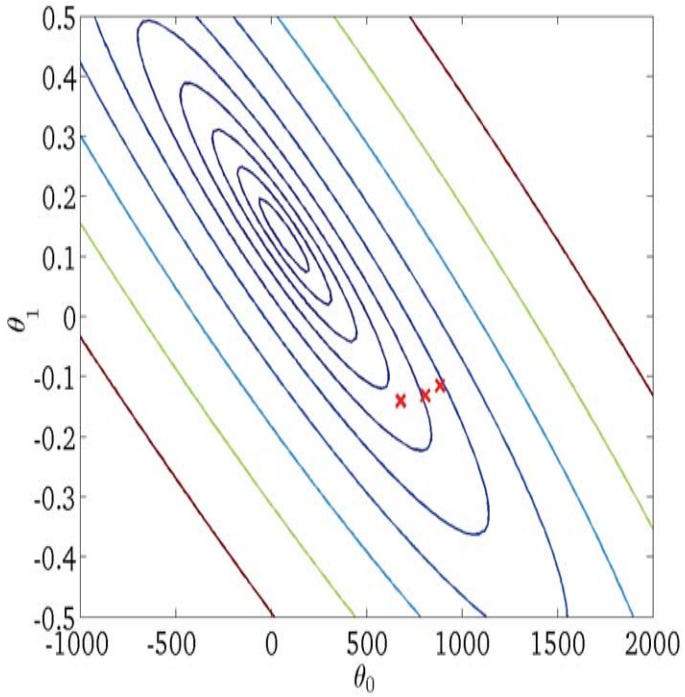
$h_\theta(x)$

(for fixed θ_0, θ_1 , this is a function of x)

$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)

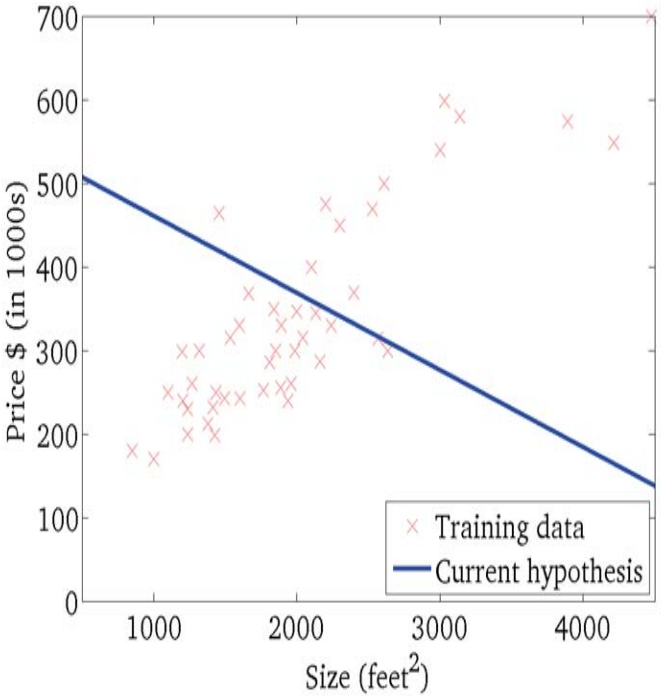


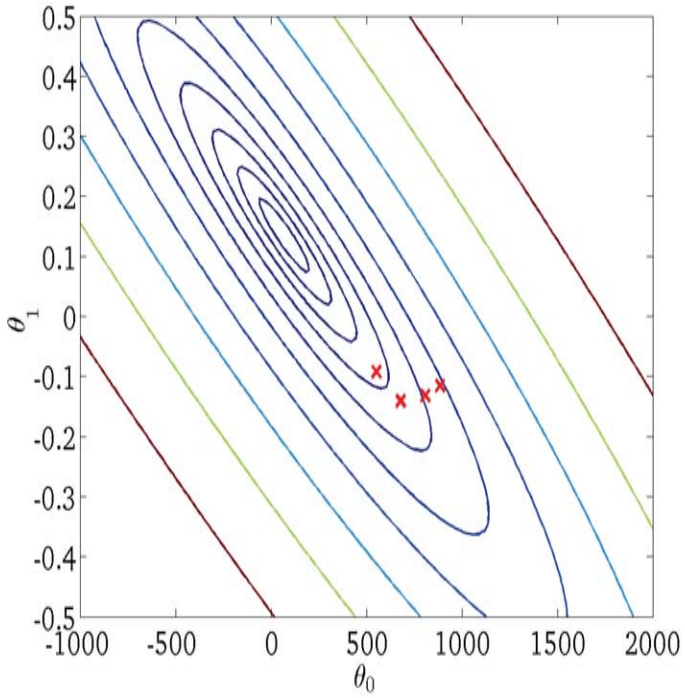


Gradient Descent for LR Price~Size – Iteration 3

$h_\theta(x)$ (for fixed θ_0, θ_1 , this is a function of x)

$J(\theta_0, \theta_1)$ (function of the parameters θ_0, θ_1)

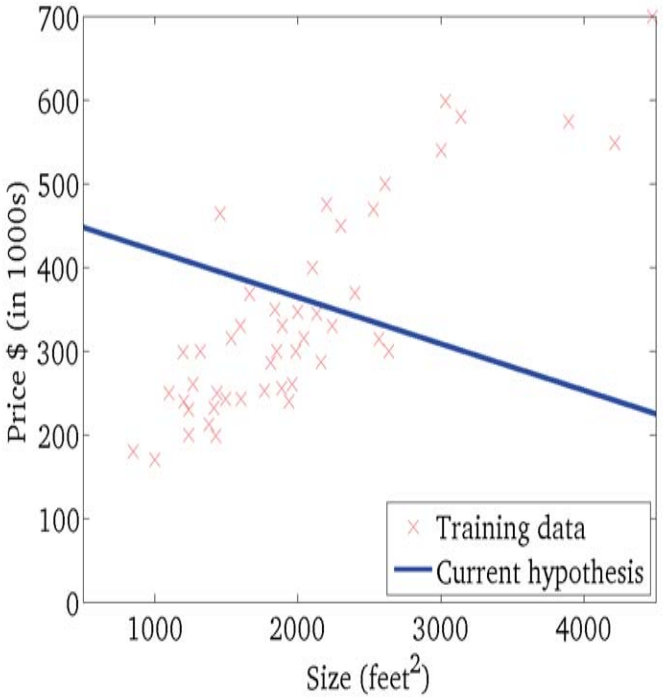


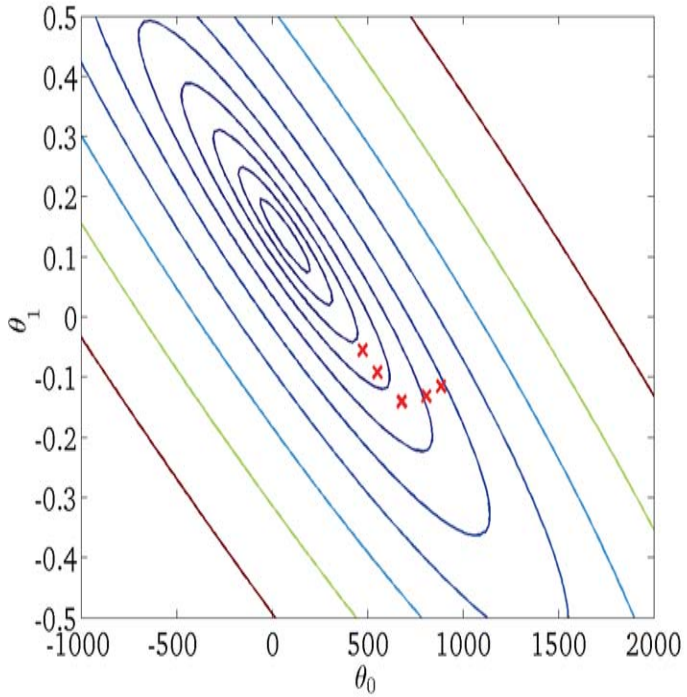


Gradient Descent for LR Price~Size – Iteration 4

$h_\theta(x)$ (for fixed θ_0, θ_1 , this is a function of x)

$J(\theta_0, \theta_1)$ (function of the parameters θ_0, θ_1)





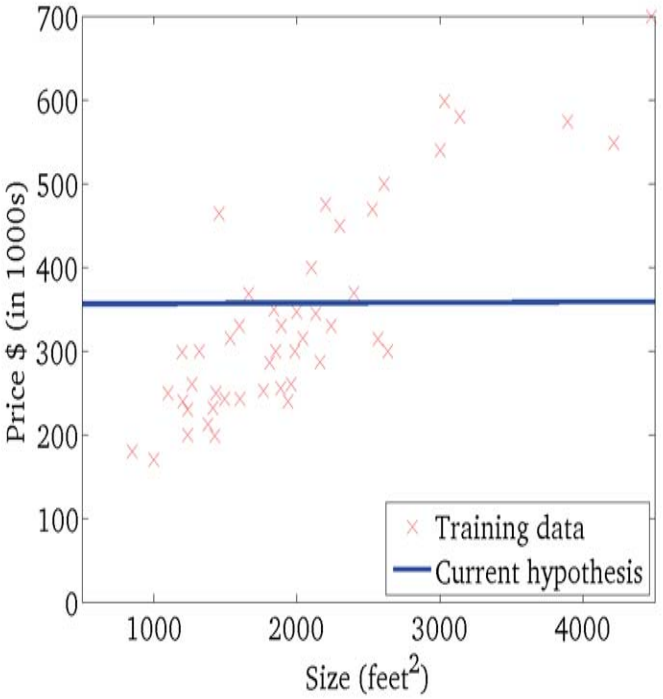
Gradient Descent for LR Price~Size – Iteration 5

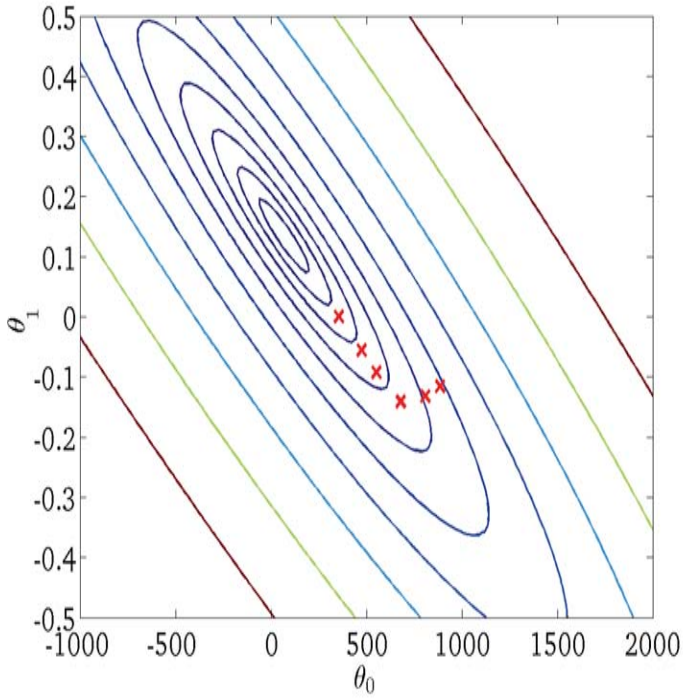
$h_\theta(x)$

(for fixed θ_0, θ_1 , this is a function of x)

$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)





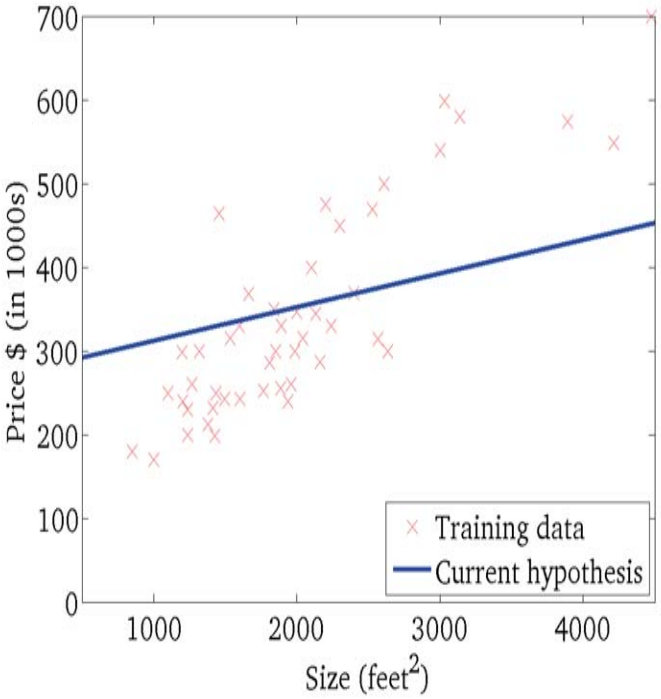
Gradient Descent for LR Price~Size – Iteration 6

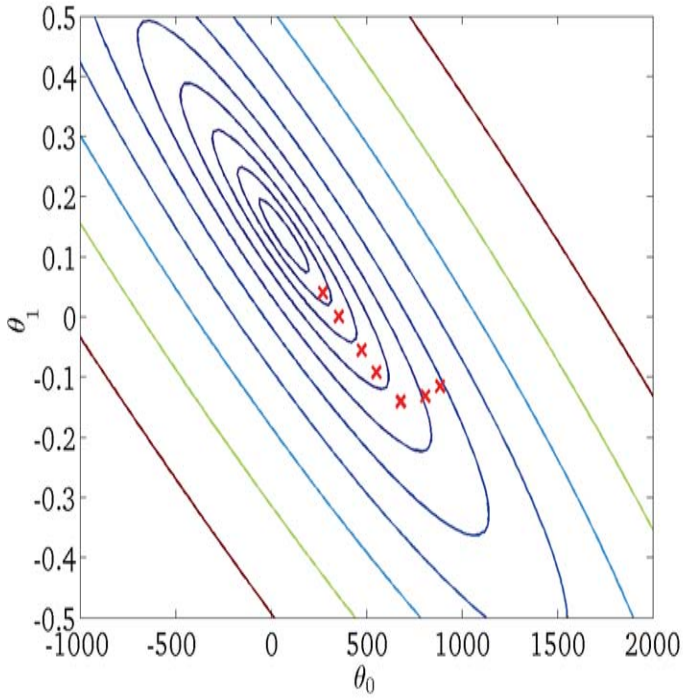
$h_\theta(x)$

(for fixed θ_0, θ_1 , this is a function of x)

$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)

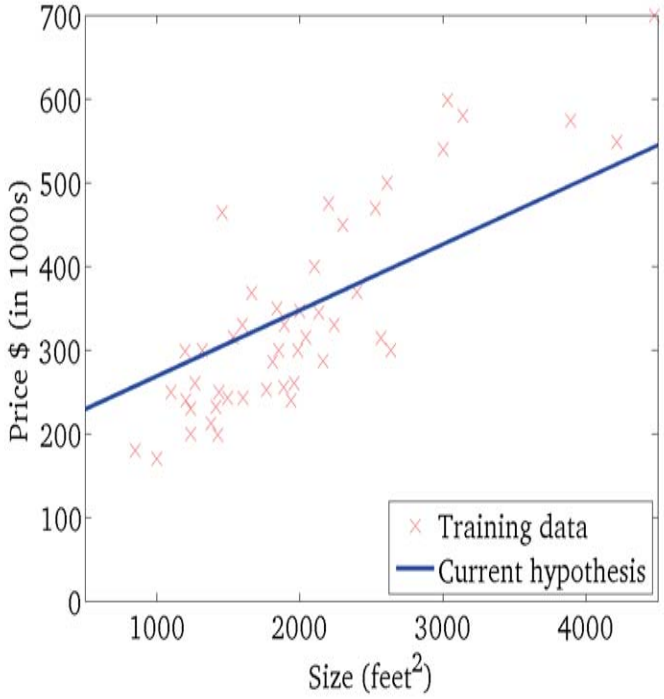


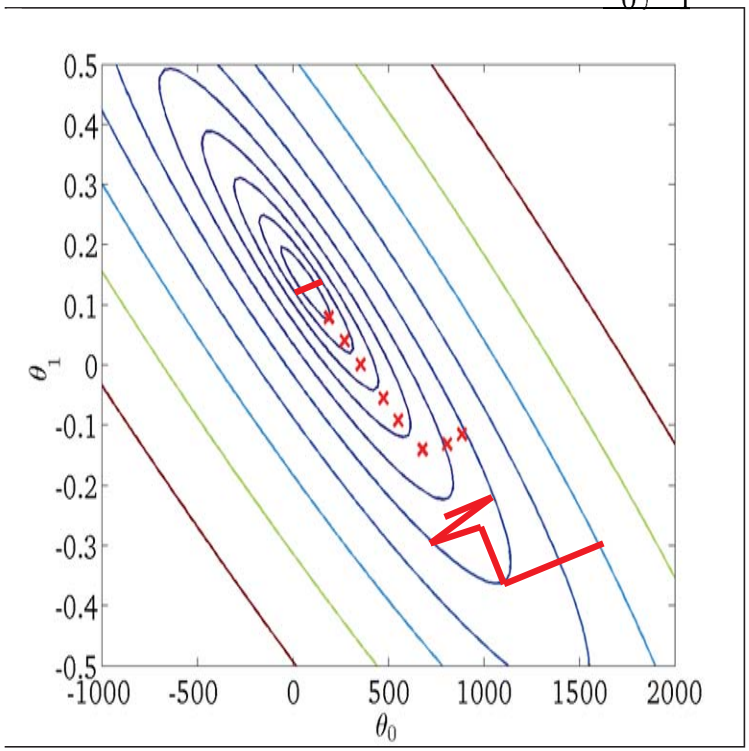


Gradient Descent for LR Price~Size – Iteration 7

$h_\theta(x)$ (for fixed θ_0, θ_1 , this is a function of x)

$J(\theta_0, \theta_1)$ (function of the parameters θ_0, θ_1)

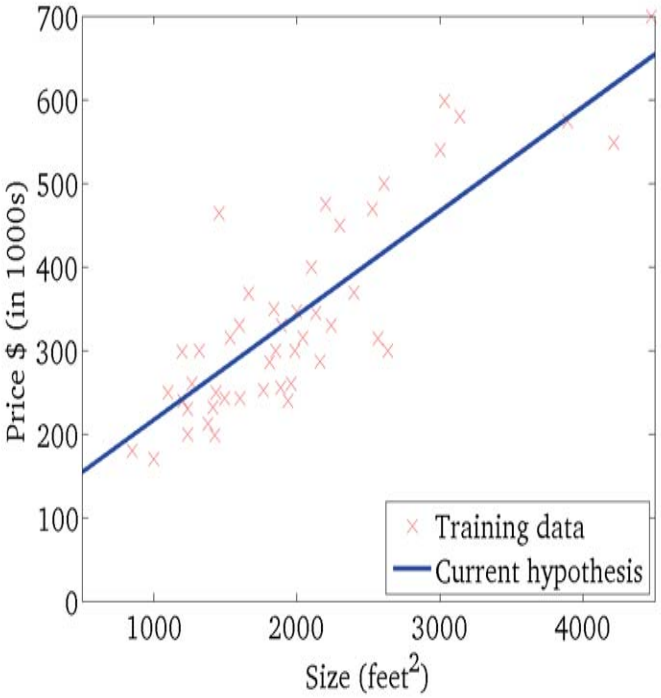


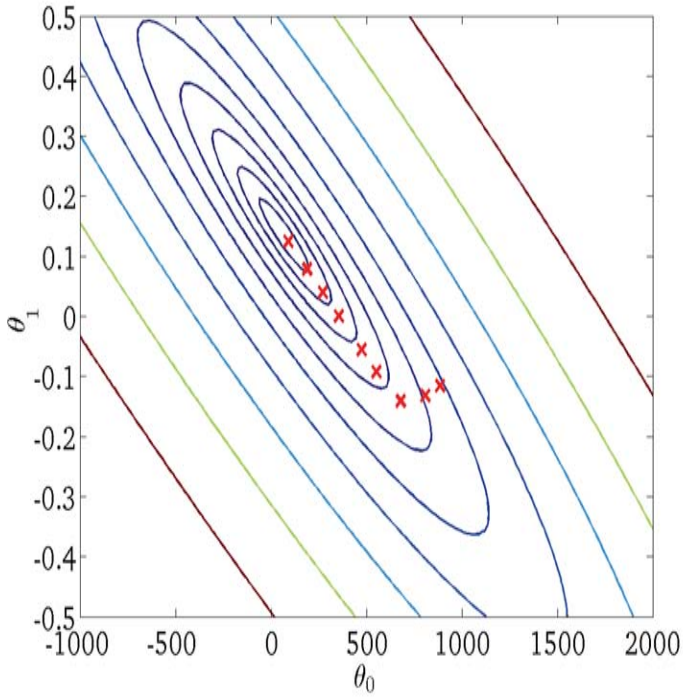


Gradient Descent for LR Price~Size – Converged after 9 steps

$h_\theta(x)$ (for fixed θ_0, θ_1 , this is a function of x)

$J(\theta_0, \theta_1)$ (function of the parameters θ_0, θ_1)





As machine learners why do we care?

- Optimization is at the heart of many (most practical?) machine learning algorithms.
  - Linear regression:

$$\underset{w}{\text{minimize}} \quad \|Xw - y\|^2$$

  - Classification (logistic regresion or SVM):

$$\underset{w}{\text{minimize}} \sum_{i=1}^{n} \log\left(1 + \exp(-y_i x_i^T w)\right)$$

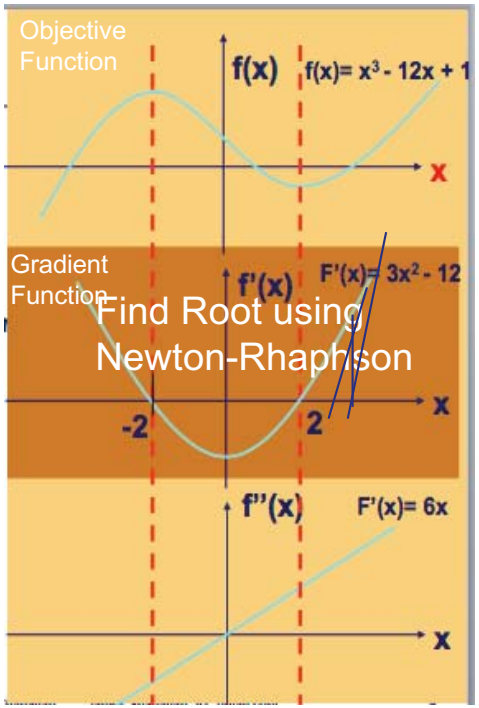
$$\text{or} \quad \|w\|^2 + C\sum_{i=1}^{n} \xi_i \quad \text{s.t.} \quad \xi_i \geq 1 - y_i x_i^T w, \xi_i \geq 0.$$

Gradient Descent (a simpler root finder)

GIVEN: Minimize f(x)

STEP 1: Find the zeros of the gradient function f'(x)

- Using Bisection method; OR Newton-Raphson; OR Gradient Descent



$$x^{i+1} = x^i - \left[f''(x^i)\right]^{-1} f'(x^i) \qquad \text{Univariate case}$$

$$x^{i+1} = x^i - \left[H(x^i)\right]^{-1} J(x^i) \qquad \text{Multivariate Case}$$

Newton-Raphson

Calculating f''(x), the Hessian H in multivariate case, and inverting it is complex so simpler algorithms have been developed such as gradient descent

learning rate

$$x^{i+1} = x^i - \alpha^i f'(x^i) \qquad \text{Univariate case}$$

$$x^{i+1} = x^i - \alpha^i J(x^i) \qquad \text{Multivariate Case}$$

Gradient Descent

How large should I step in the positive gradient direction (gradient ascent)

- or in the negative gradient direction (gradient descent)

Convergence criteria. E.g., decision vector X does not change that much or error does not change

Linear Regression via GD

$$\frac{\partial E}{\partial W_{jk}} = \frac{\partial}{\partial W_{jk}} \frac{1}{2} \sum_{k \in K} (O_k - t_k)^2$$

All training data

$$\frac{\partial E}{\partial W_i} = \frac{\partial}{\partial W_i} \sum_{n=1:Train} \frac{1}{2} (X_{in} W^T - t_i)^2$$

One example

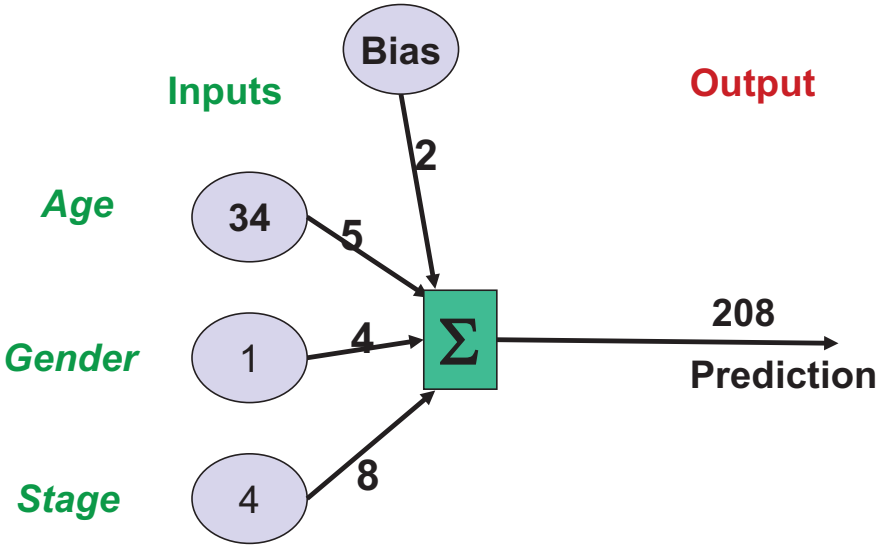
$$\frac{\partial E}{\partial W_i} = \frac{\partial}{\partial W_i} \frac{1}{2} (O_i - t_i)^2$$

$$\frac{\partial E}{\partial W_i} = (O_i - t_i) \frac{\partial}{\partial W_i} (O_i - t_i)$$

$$\frac{\partial E}{\partial W_i} = (O_i - t_i) \frac{\partial}{\partial W_i} (X_i W^T - t_i)$$

$$\frac{\partial E}{\partial W} = (O - t) X$$

Linear Regression Model



$$Y = X\beta + \varepsilon$$

Independent variables

x1, x2, x3

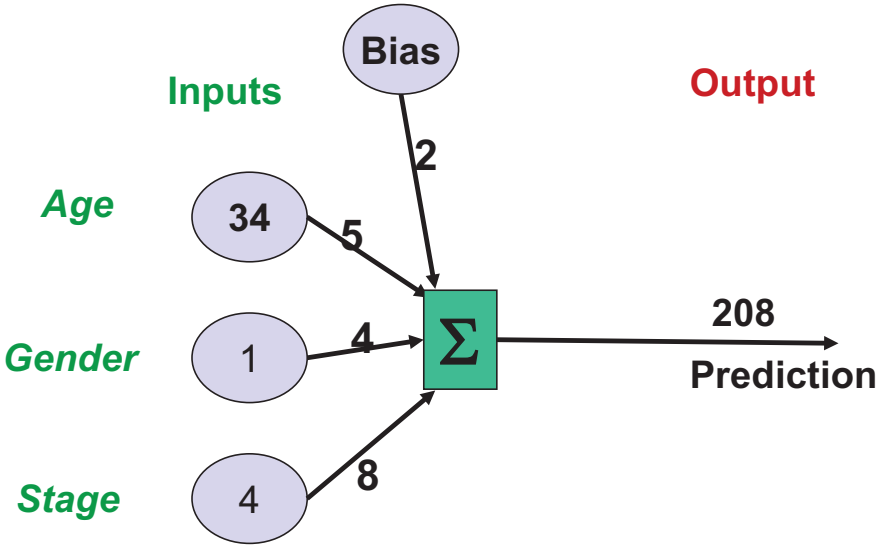
Coefficients

a, b, c; bias

Dependent variable

Prediction

Linear Regression Model



$$Y = X\beta + \varepsilon$$

Independent variables

x1, x2, x3

Coefficients

a, b, c; bias

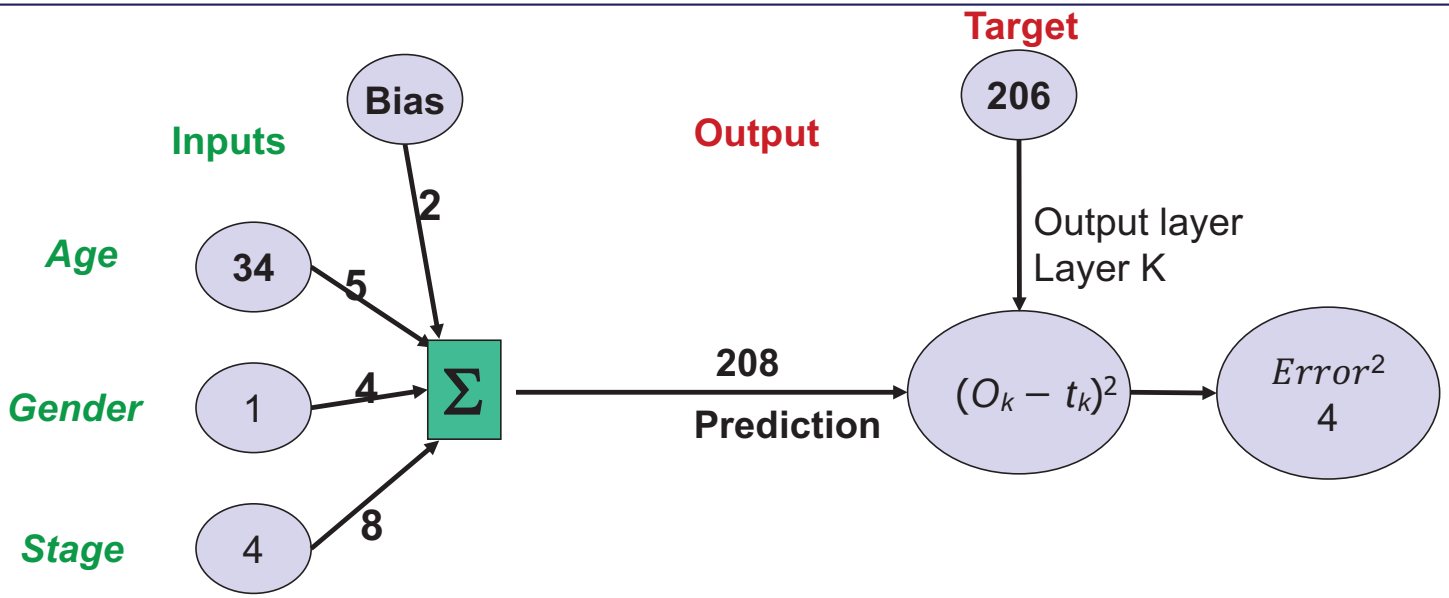
Dependent variable

Prediction

Training example[34,1,4]

Weight vector = [5, 4, 8] and b =2

Linear Regression Model: MSE



Independent variables

x1, x2, x3

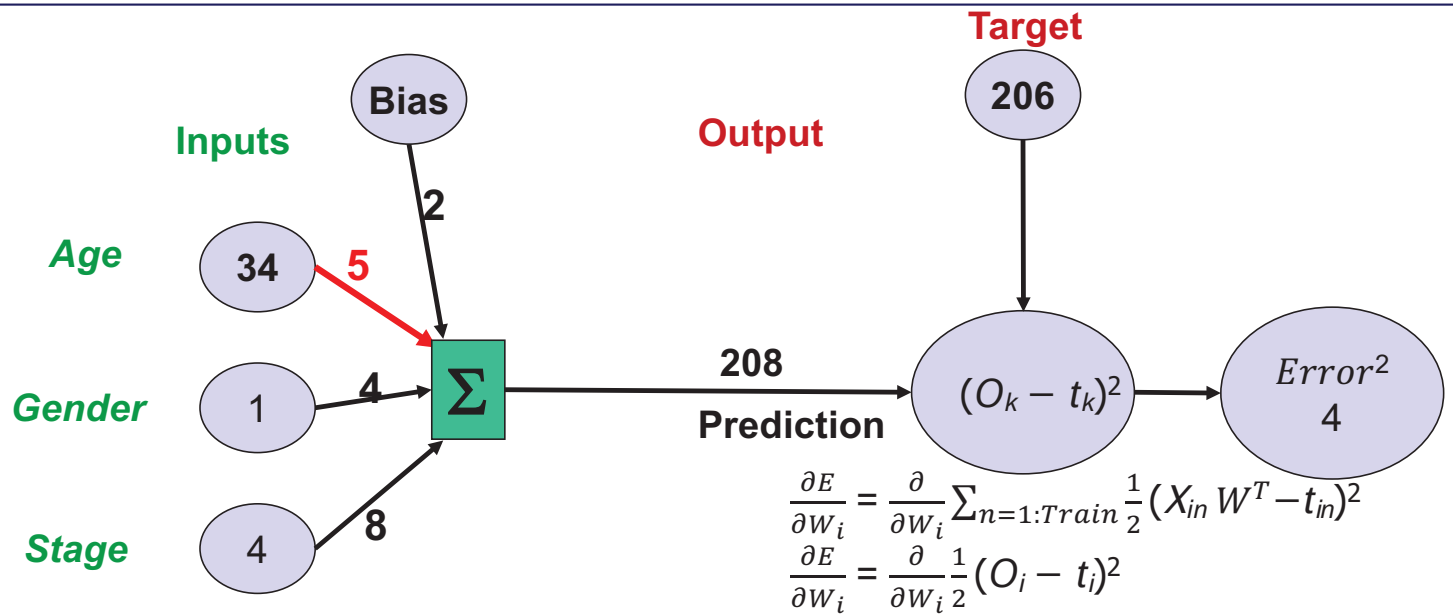
Coefficients

a, b, c; bias

Dependent variable

Prediction

Linear Regression Model: MSE



$$\frac{\partial E}{\partial W_i} = \frac{\partial}{\partial W_i} \sum_{n=1:Train} \frac{1}{2} (X_{in} W^T - t_{in})^2$$

$$\frac{\partial E}{\partial W_i} = \frac{\partial}{\partial W_i} \frac{1}{2} (O_i - t_i)^2$$

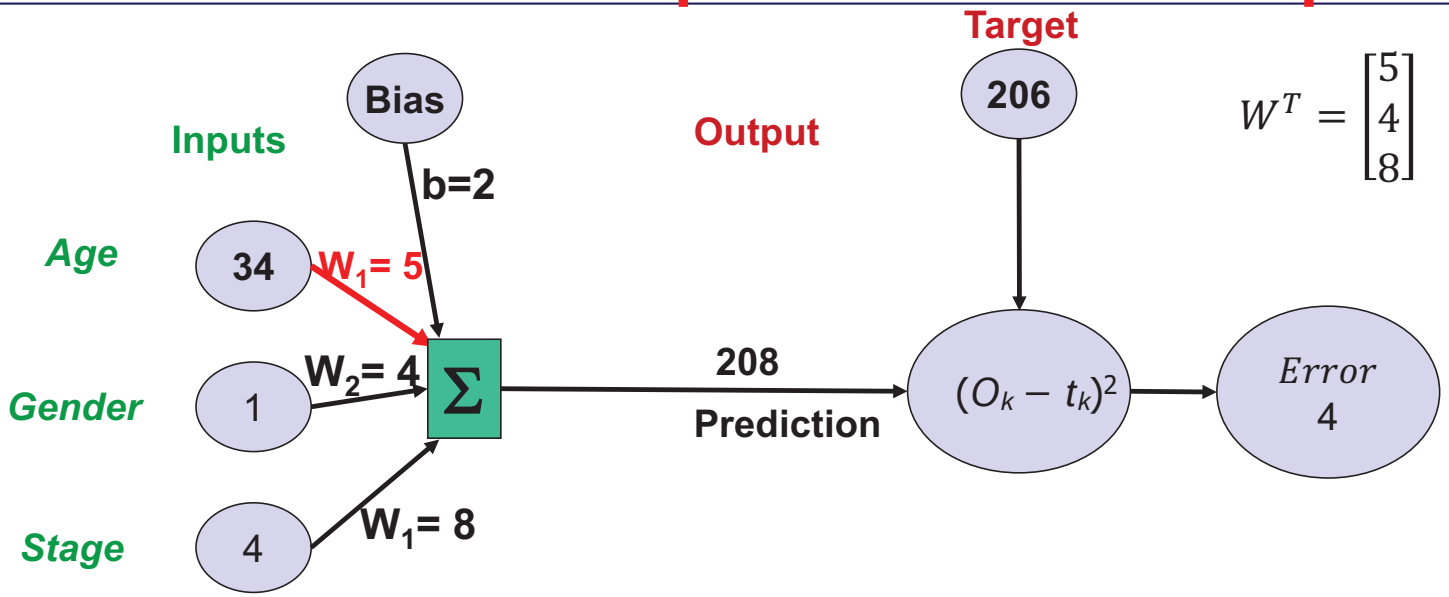
$$\frac{\partial E}{\partial W_i} = (O_i - t_i) \frac{\partial}{\partial W_i} (O_i - t_i)$$

$$\frac{\partial E}{\partial W_i} = (O_i - t_i) \frac{\partial}{\partial W_i} (X_i W^T - t_i)$$

$$\frac{\partial E}{\partial W_i} = (O_i - t_i) X_i$$

$$W = W - \alpha \frac{\partial E}{\partial W}$$

Linear Regression Model: Gradient update for one example

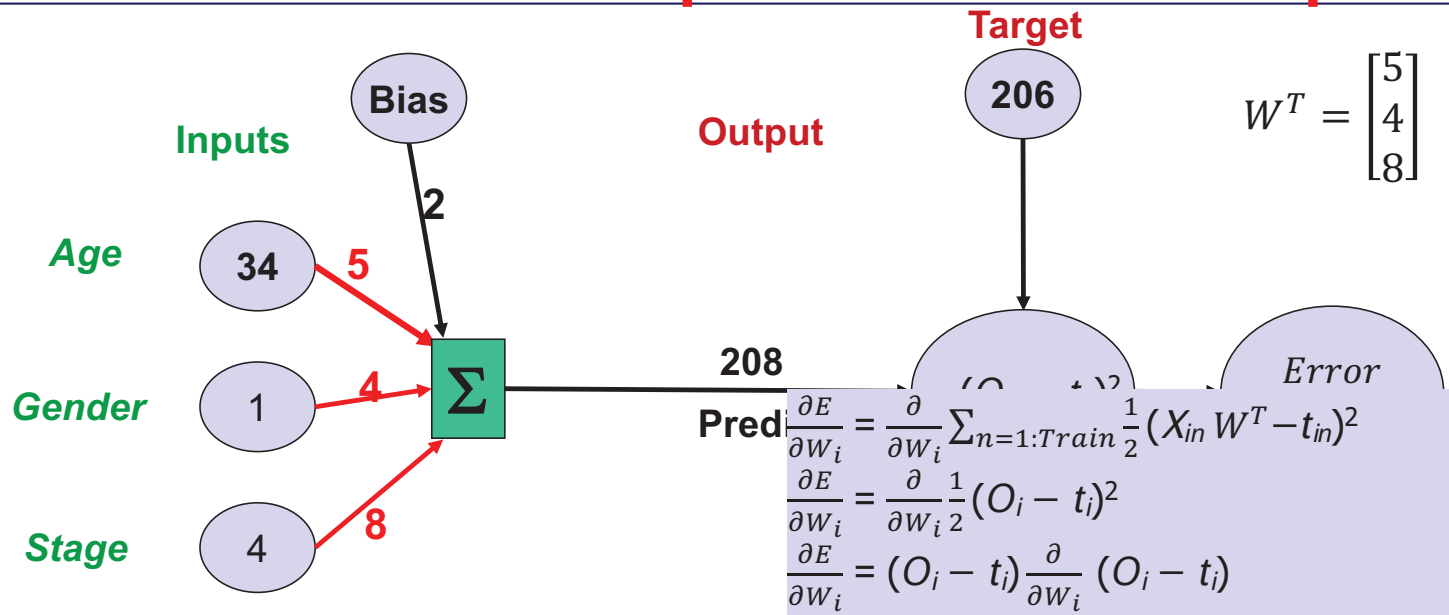


Quiz Prep example

Weight Gradient for this example[34,1,4]

Gradient for b

Linear Regression Model: Gradient update for one example



$$W^T = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix}$$

$$\frac{\partial E}{\partial W_i} = \frac{\partial}{\partial W_i} \sum_{n=1:Train} \frac{1}{2} (X_{in} W^T - t_{in})^2$$

$$\frac{\partial E}{\partial W_i} = \frac{\partial}{\partial W_i} \frac{1}{2} (O_i - t_i)^2$$

$$\frac{\partial E}{\partial W_i} = (O_i - t_i) \frac{\partial}{\partial W_i} (O_i - t_i)$$

$$\frac{\partial E}{\partial W_i} = (O_i - t_i) \frac{\partial}{\partial W_i} (X_i W^T - t_i)$$

$$\frac{\partial E}{\partial W} = (O - t)\, X$$

$$W = W - \alpha \frac{\partial E}{\partial W}$$

$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} 34 \\ 1 \\ 4 \end{bmatrix}$$

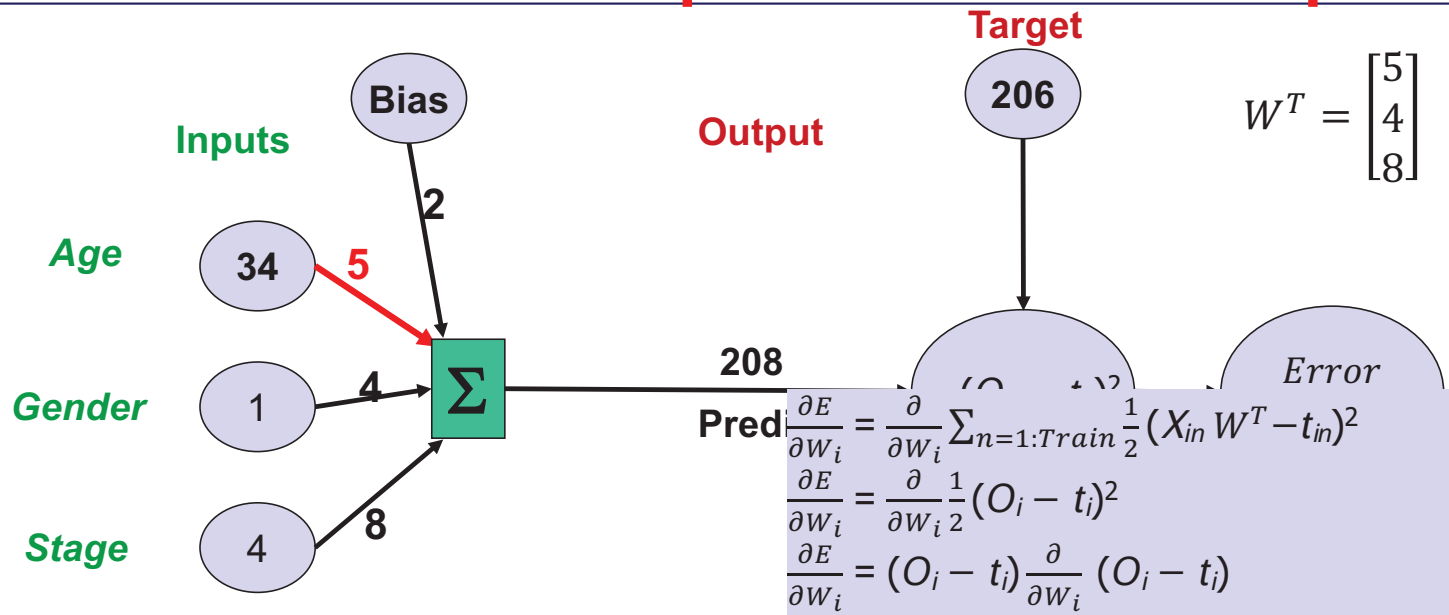
$$4.32 = 5 - 0.01 \times (2 \times 34) \text{ for } i = 1$$

Weight Gradient for this example[34,1,4]

Gradient for b = 2

Assume a learning rate α=0.01

Linear Regression Model: Gradient update for one example



$$W^T = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix}$$

$$\frac{\partial E}{\partial W_i} = \frac{\partial}{\partial W_i} \sum_{n=1:Train} \frac{1}{2} (X_{in} W^T - t_{in})^2$$

$$\frac{\partial E}{\partial W_i} = \frac{\partial}{\partial W_i} \frac{1}{2} (O_i - t_i)^2$$

$$\frac{\partial E}{\partial W_i} = (O_i - t_i) \frac{\partial}{\partial W_i} (O_i - t_i)$$

$$\frac{\partial E}{\partial W_i} = (O_i - t_i) \frac{\partial}{\partial W_i} (X_i W^T - t_i)$$

$$\frac{\partial E}{\partial W} = (O - t)\, X$$

$$W = W - \alpha \frac{\partial E}{\partial W}$$

$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} 34 \\ 1 \\ 4 \end{bmatrix}$$

$$4.32 = 5 - 0.01 \times (2 \times 34) \text{ for } i = 1$$

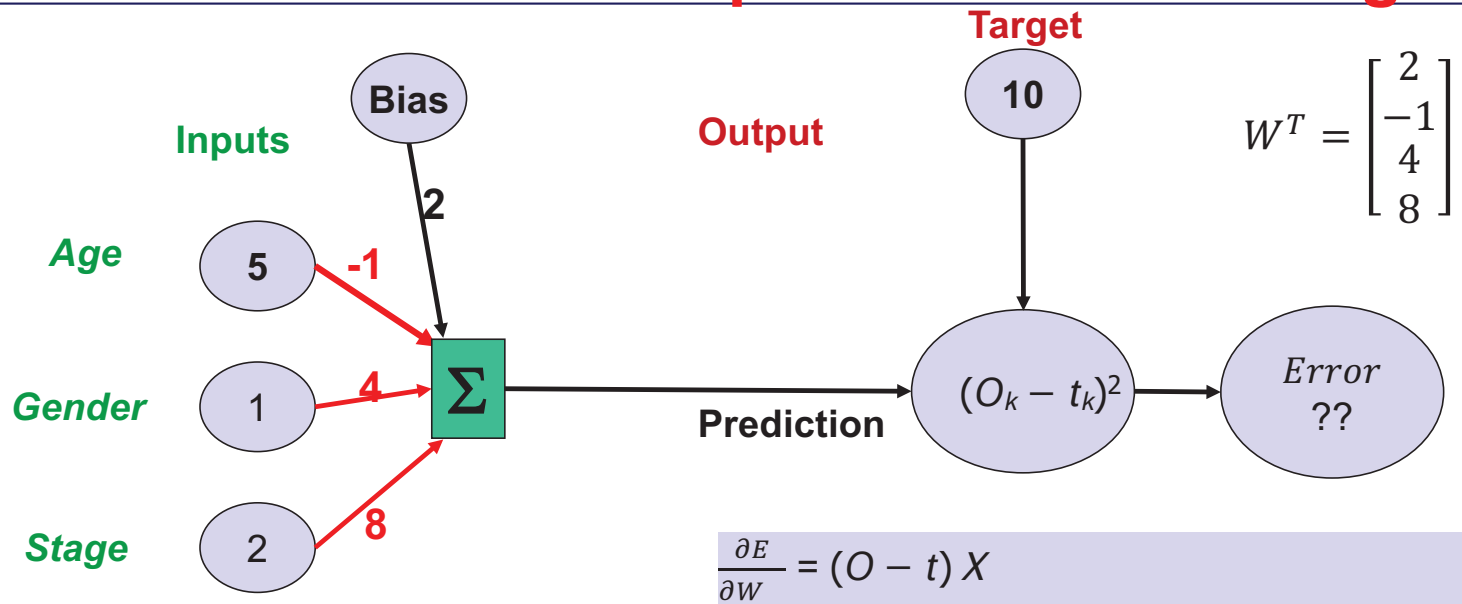
NOTE: the gradient is a weighted sum of all examples; here the weight is 2 for this example

Gradient for this example[34,1,4]

Gradient for b = 2

Assume a learning rate α=0.01

Quiz 2.1: calculate gradient descent update for this setting



$$W^T = \begin{bmatrix} 2 \\ -1 \\ 4 \\ 8 \end{bmatrix}$$

Gradient for this example[5, 1,2]

Gradient for bias?

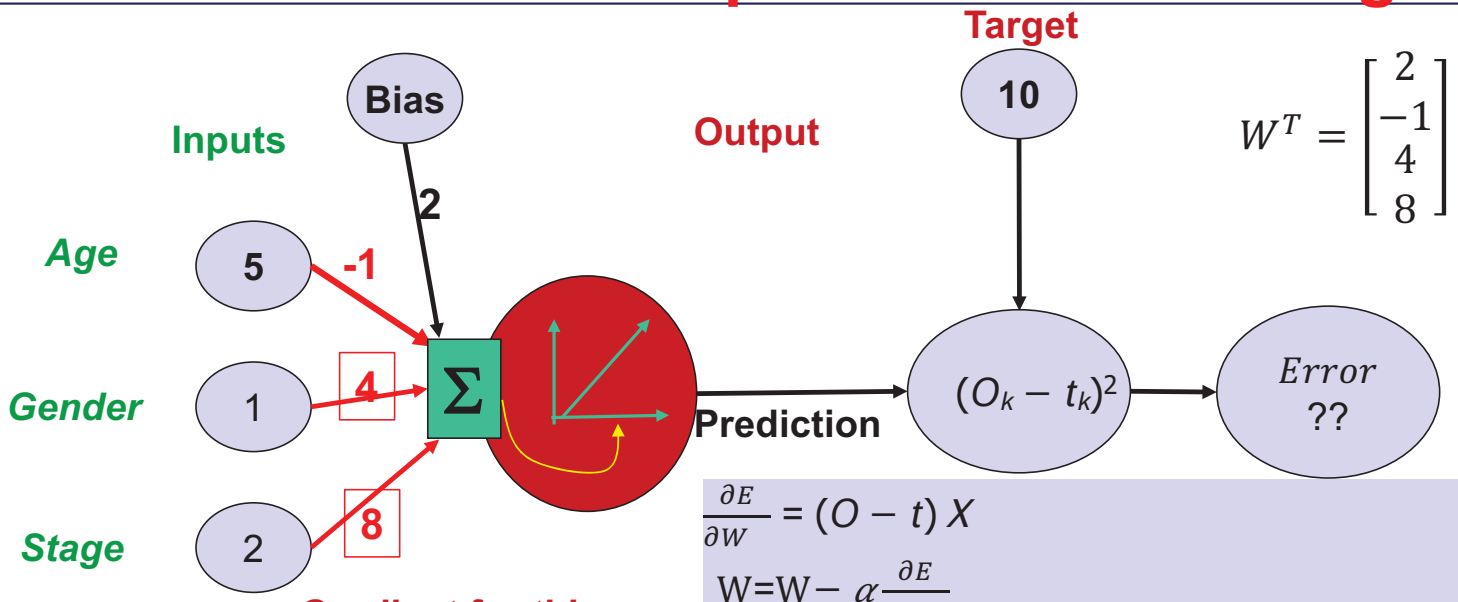
$$\frac{\partial E}{\partial W} = (O - t)\,X$$

$$W = W - \alpha \frac{\partial E}{\partial W}$$

$$\begin{bmatrix} ? \\ ? \\ ? \\ ? \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times ? \times \begin{bmatrix} 1 \\ 5 \\ 1 \\ 2 \end{bmatrix}$$

T1: https://canvas.instructure.com/courses/1159217/quizzes/2102928

Quiz: calculate gradient descent update for this setting



$W^T = \begin{bmatrix} 2 \\ -1 \\ 4 \\ 8 \end{bmatrix}$

Gradient for this example[5, 1,2]

Gradient for bias?

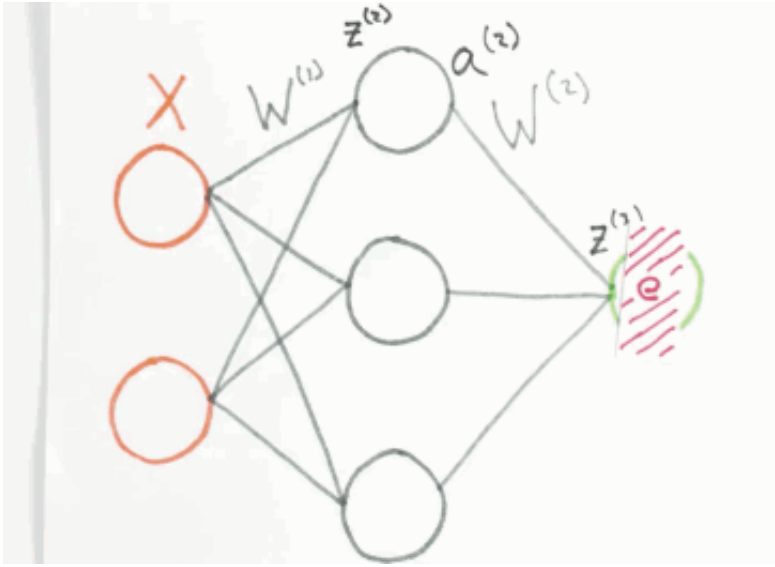
$$\frac{\partial E}{\partial W} = (O - t)\,X$$

$$W = W - \alpha \frac{\partial E}{\partial W}$$

$$\begin{bmatrix} ? \\ ? \\ ? \\ ? \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times ? \times \begin{bmatrix} 1 \\ 5 \\ 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1.93 \\ -1.35 \\ 3.93 \\ 7.86 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 7 \times \begin{bmatrix} 1 \\ 5 \\ 1 \\ 2 \end{bmatrix}$$

Backpropagation



http://www.cbcity.de/tutorial-neuronale-netze-einfach-erklaert

▽ the gradient chorus

- **What is the gradient for linear regression?**

- **Chorus**
  - The gradient is the weighted sum of the training data, where the weights are proportional to the error (for each example) !

weight *example*

$$\frac{\partial E}{\partial W} = (O - t)\,\boldsymbol{X}$$

$$W = W - \alpha \times \frac{\partial E}{\partial W}$$

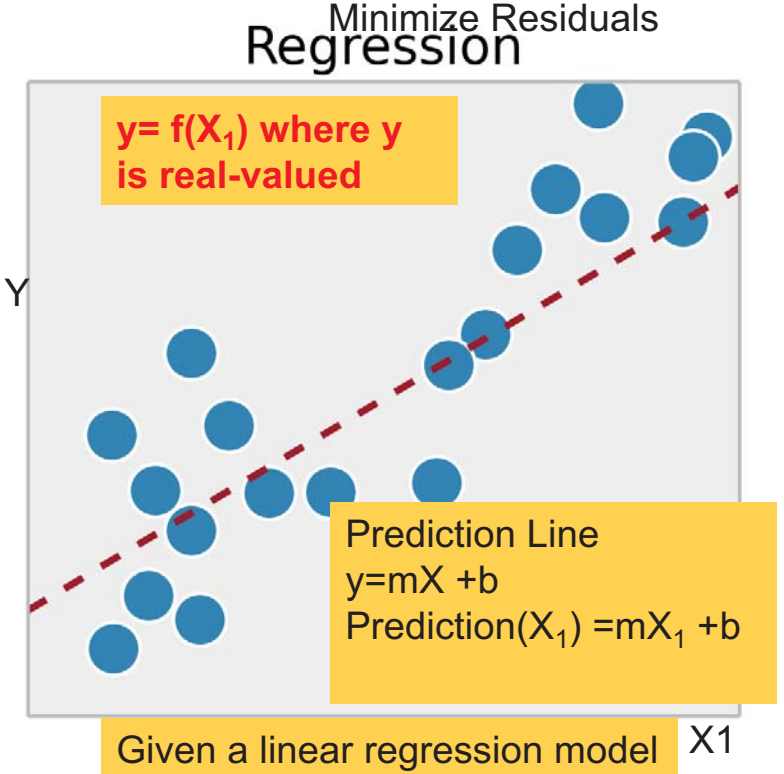
$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} \mathbf{34} \\ \mathbf{1} \\ \mathbf{4} \end{bmatrix}$$

$$4.32 = 5 - 0.01 \times (2 \times 34) \text{ for } i = 1$$

Linear Regression Loss Function

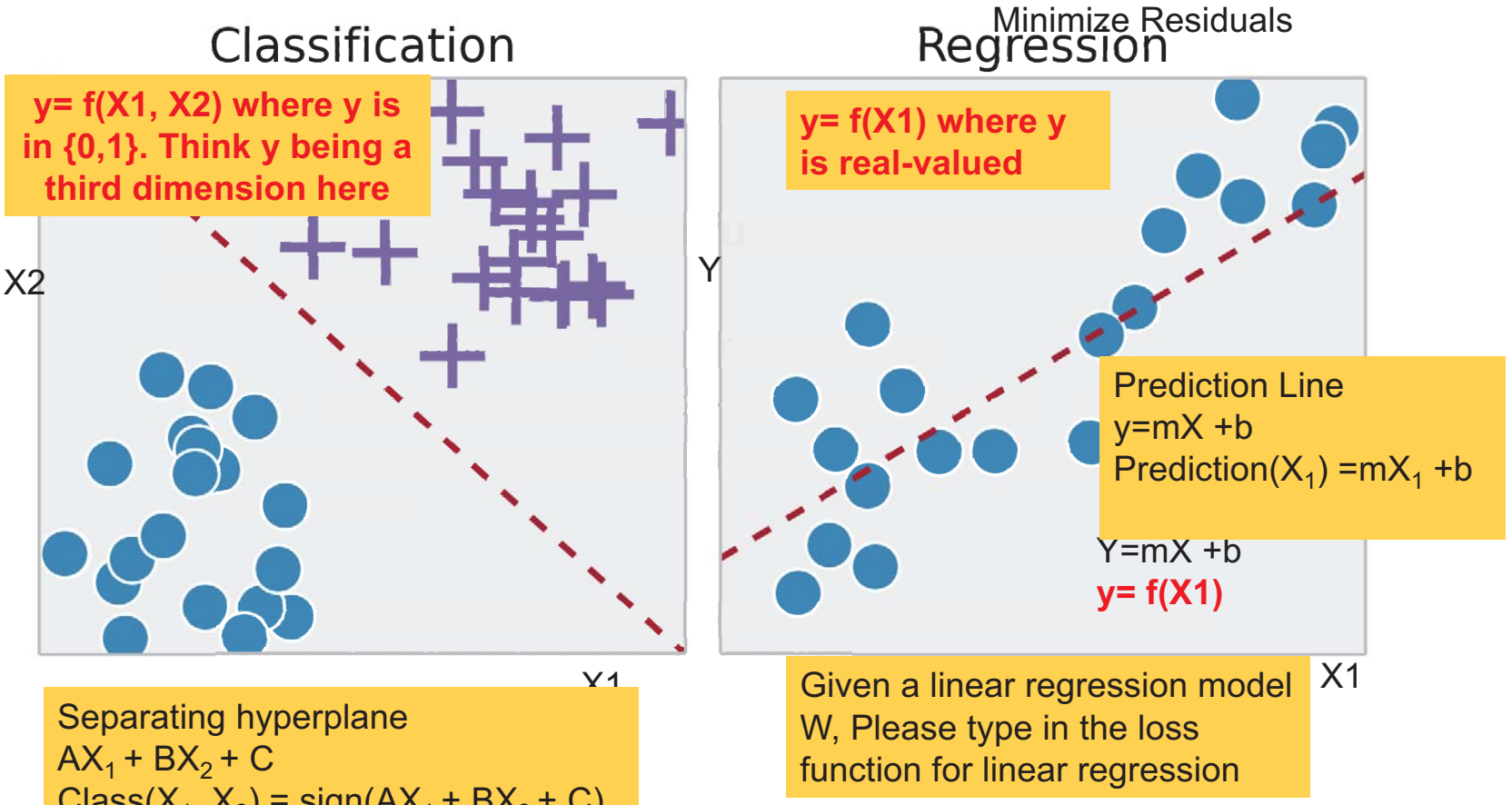
$$MSE(W) = \frac{1}{n}\sum (y_i - W^T X_i)^2$$

Minimize Residuals



Given a linear regression model W, Please type in the loss function for linear regression

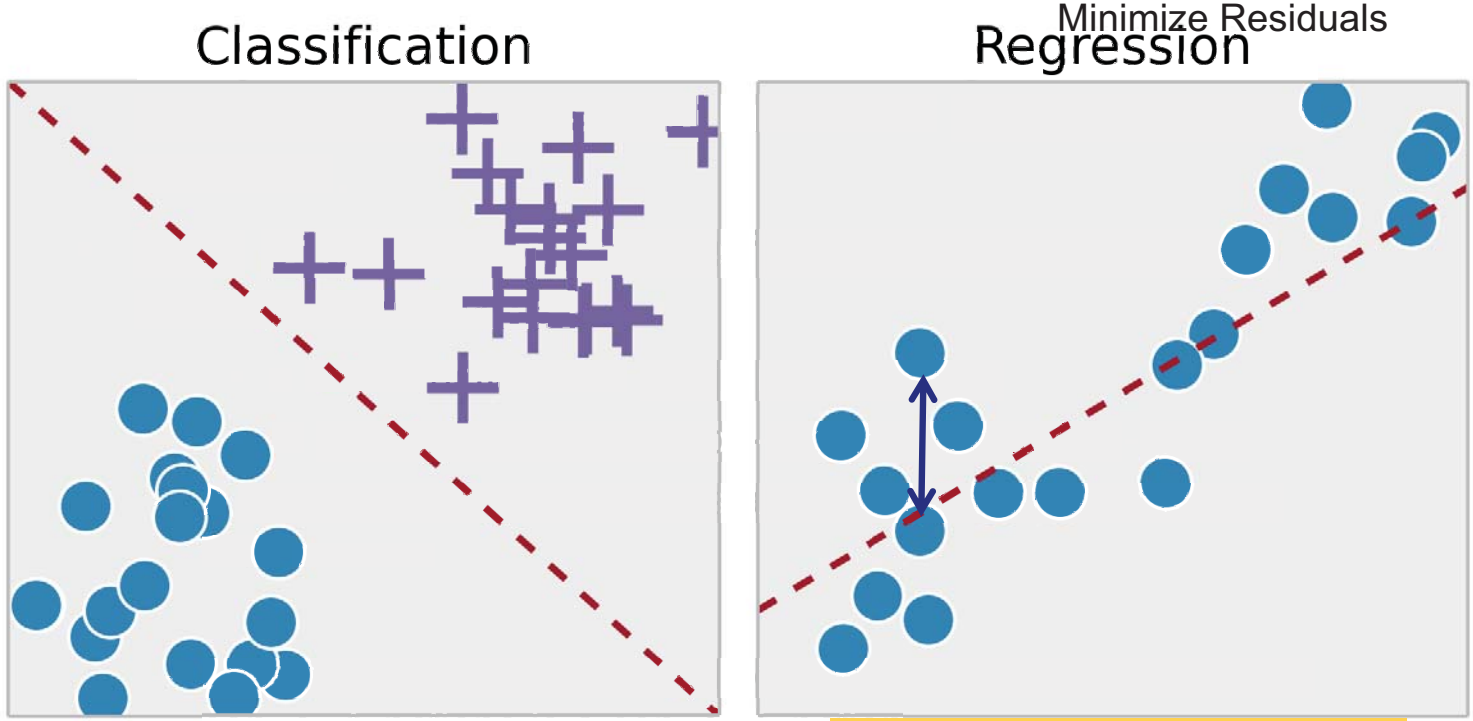
Convex optimization in ML: Loss Functions



Convex optimization in ML

$$MSE(W) = \frac{1}{n}\sum (y_i - W^T X_i)^2$$

Minimize Residuals



Please type in the loss function for linear regression (W is the model)

Convex optimization in ML

$$J_P(W, X_1^L) = \sum_{\{X_i | y_i \langle W, X_i \rangle < 0\}} (W^T X_i y_i)$$

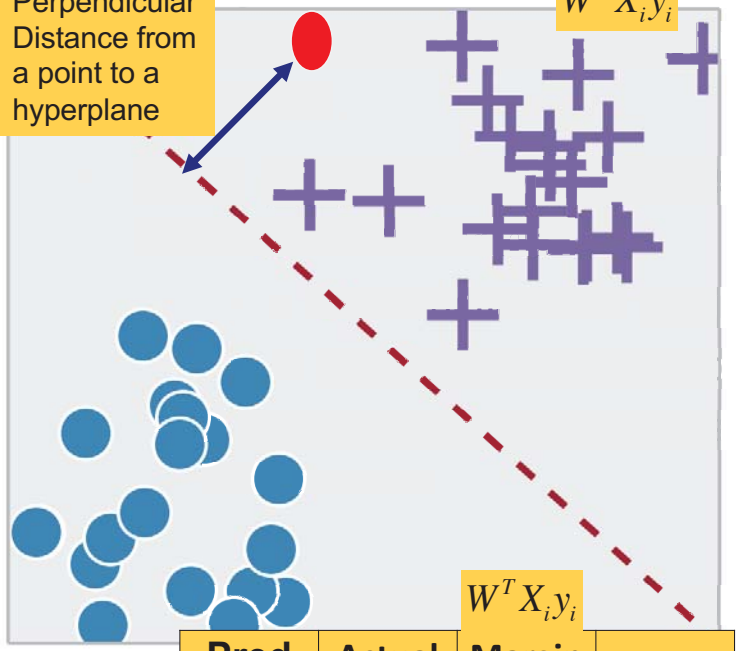
$$MSE(W) = \frac{1}{n} \sum (y_i - W^T X_i)^2$$

Minimize Residuals

Classification

Perpendicular Distance from a point to a hyperplane

$W^T X_i y_i$

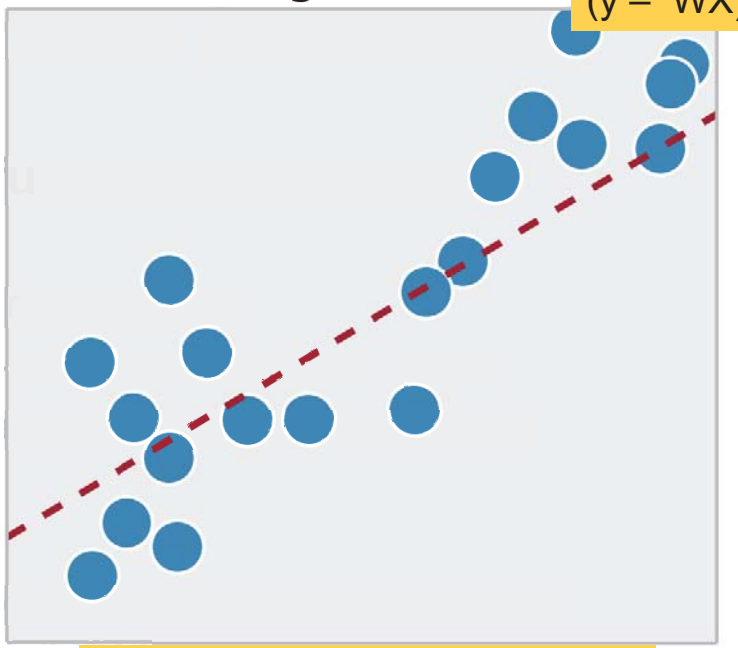


$W^T X_i y_i$

Pred	Actual	Margin	
+	+	+	(GOOD)
+	-	-	LOSS
-	+	-	LOSS
-	-	+	(GOOD)

Regression

$(y - WX)^2$



Please type in the loss function for linear regression (W is the model)

Loss functions; a unifying view

- Loss function consists of:
  - loss term ($L(m_i(w)$, expressed in terms of the margin of each training example) and
  - regularization term ($R(w)$ expressed as a function of the model complexity)

Loss term regularization term

$$J(w) = \sum_i L(m_i(w)) + \lambda R(w) \quad (14.1)$$

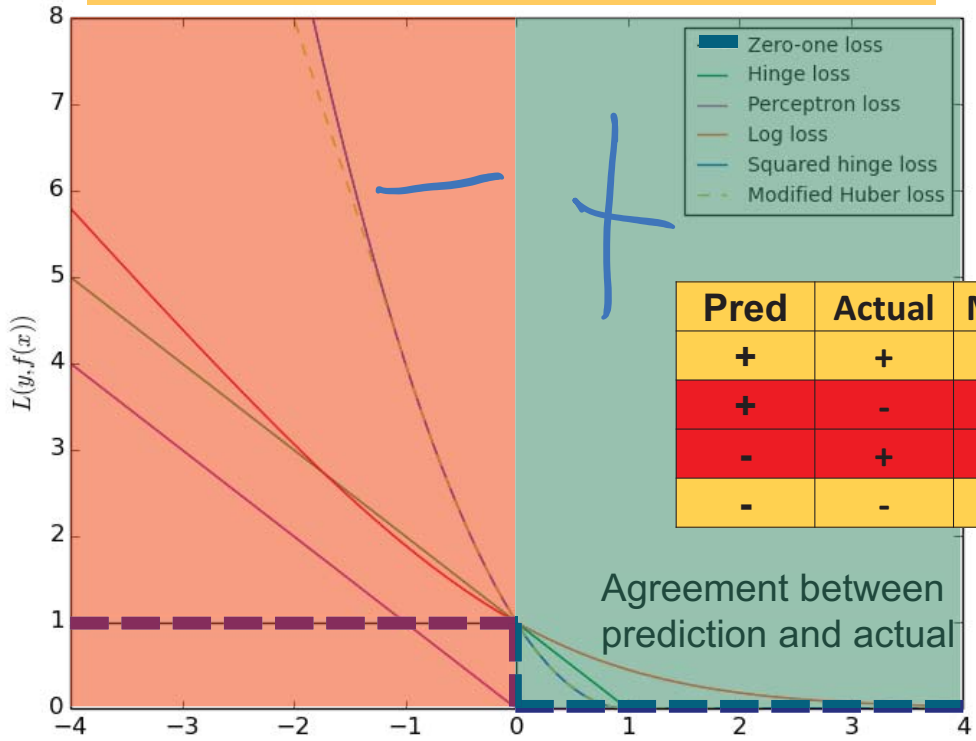
$$m_i = y^{(i)} f_w(x^{(i)}) \quad (14.2)$$

f_w

Focus only on the loss term
E.g., Linear regression, logistic regression Soft SVMs

Machine Learning Objective Function: classification

Classification loss functions



Loss

Pred	Actual	Margin	
+	+	+	(GOOD)
+	-	-	LOSS
-	+	-	LOSS
-	-	+	(GOOD)

Agreement between prediction and actual

Disagreement between prediction and actual

y ✕ f(X)

+ +=+

Margin

From linear to non-linear models
From Convex to non-convex optimization

- The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex.
- This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs.
- Convex optimization converges starting from any initial parameters (in theory—in practice it is very robust but can encounter numerical problems).

Train NN using MAXIMUM LIKELIHOOD

- **Most modern neural networks are trained using maximum likelihood**

- **Minimizing MSE will output a maximum likelihood hypothesis**
  - A straightforward Bayesian analysis will show that under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will output a maximum likelihood hypothesis.
- **Minimizing cross entropy will output a maximum likelihood hypothesis**

See Tom Mitchell's ML Book, 1997 (section 6.4)
See Goodfellow et al. DL Book, 2016 (section 6.2.1)

MAXIMUM LIKELIHOOD AND LEAST-SQUARED ERROR HYPOTHESES

- **A straightforward Bayesian analysis will show that under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will output a maximum likelihood hypothesis.**

SGD applied to non-convex loss functions…

- Stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- For feedforward neural networks, it is important to initialize all weights to small random values.
- The biases may be initialized to zero or to small positive values.
- As with other machine learning models, to apply gradient-based learning we must choose a cost function, and we must choose how to represent the output of the model.

Cost, aka, loss, Functions

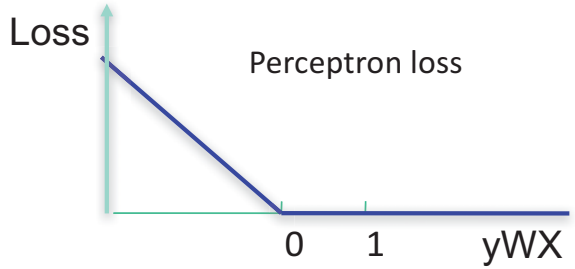
- An important aspect of the design of a deep neural network is the choice of the cost function.
- Fortunately, the cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.

~1700s	• The prologue
1940-1970	• Act 1 Tinker: Hack it up
1970-1995	• Act 2 BackProp: theory to the rescue
1995-2007	• Act 3 Layer by layer learning, a medieval pastime
2007-2015	• Act 4 Introspection: better init. and activation functions
2015 -	• Act 5 Express-laning the gradient: Skip Connections, the SoTA frontier (LSTMs, ResNet, Highway Nets, DenseNets)
	• The epilogue

Artificial Neurons

- **1943: The McCulloch & Pitts (perceptron) neuron**
  - Weighted sum of inputs
  - Activation Function
    - "Link" function in GLM/regression (e.g., sigmoid or hyperbolic tangent link for logistic regression)
- **1957: Perceptron learning algo (Rosenblatt 1957)**
  - Learning rate and learning (update) rule
  - Classification problems
  - Neural network as a network of logistic regressions

Penalizes incorrectly classified examples x proportionally to the size of |w'x|

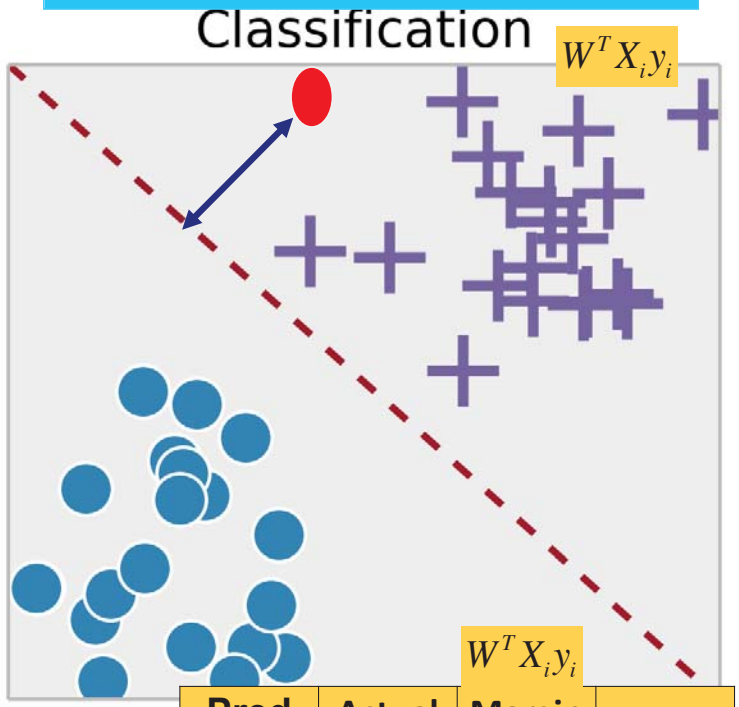


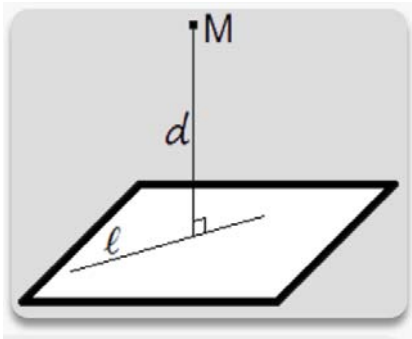
Classification ala Perceptron

$$J_P(W, X_1^L) = \sum_{\{X_i | y_i \langle W, X_i \rangle < 0\}} (W^T X_i y_i)$$

Goal in the 1950s: make no mistakes!

Perpendicular Distance from a point to a hyperplane





The distance from a point to a plane — is equal to length of the perpendicular lowered from a point on a plane.

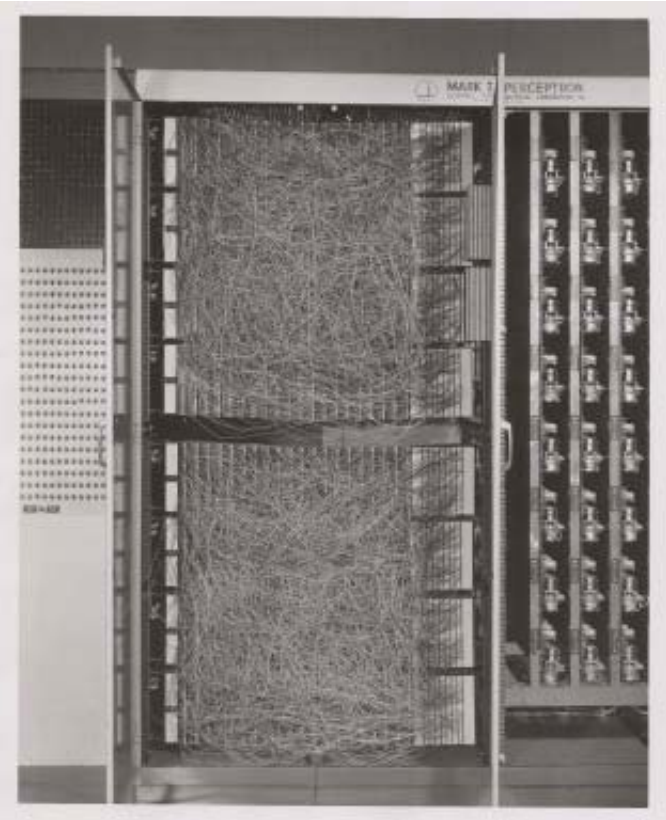
If $Ax + By + Cz + D = 0$ is a plane equation, then distance from point $M(M_x, M_y, M_z)$ to plane can be found using the following formula

$$d = \frac{|A \cdot M_x + B \cdot M_y + C \cdot M_z + D|}{\sqrt{A^2 + B^2 + C^2}}$$

$W^T X_i y_i$

Pred	Actual	Margin	
+	+	+	(GOOD)
+	-	-	LOSS
-	+	-	LOSS
-	-	+	(GOOD)

Perceptron machine



- The Mark I Perceptron machine was the first implementation of the perceptron algorithm. The machine was connected to a camera that used 20×20 cadmium sulfide photocells to produce a 400-pixel image. The main visible feature is a patchboard that allowed experimentation with different combinations of input features. To the right of that are arrays of potentiometers that implemented the adaptive weights.[2]:213

Perceptron: No loss function

The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm.

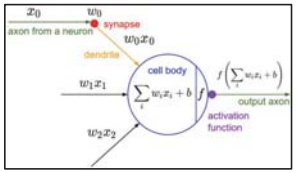
The machine was connected to a camera that used 20×20 cadmium sulfide photocells to produce a 400-pixel image.

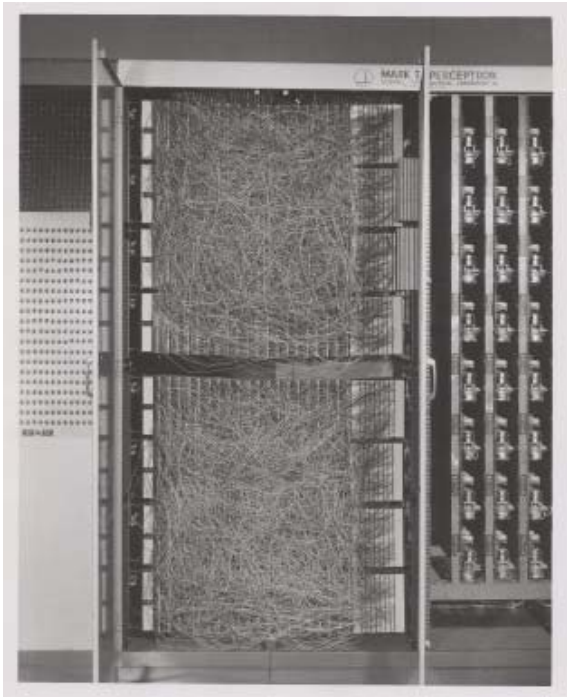
recognized
letters of the alphabet

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

update rule:

$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i},$$





Frank Rosenblatt, ~1957: Perceptron

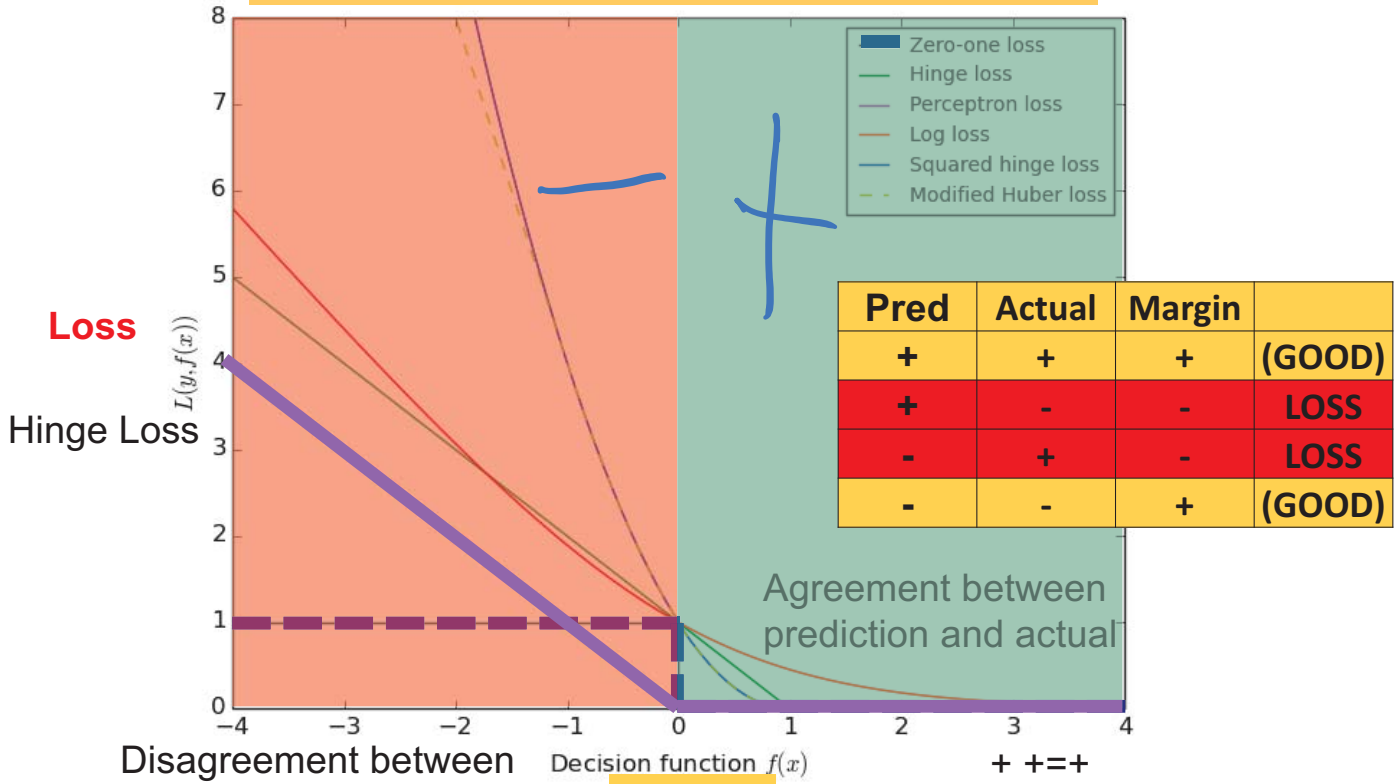
No loss function; could not use gradient descent
Linear models only

The perceptron was intended to be a machine, rather than a program, and while its first implementation was in software for the IBM 704, it was subsequently implemented in custom-built hardware as the "Mark 1 perceptron".

64

Machine Learning Objective Function: classification

Classification loss functions

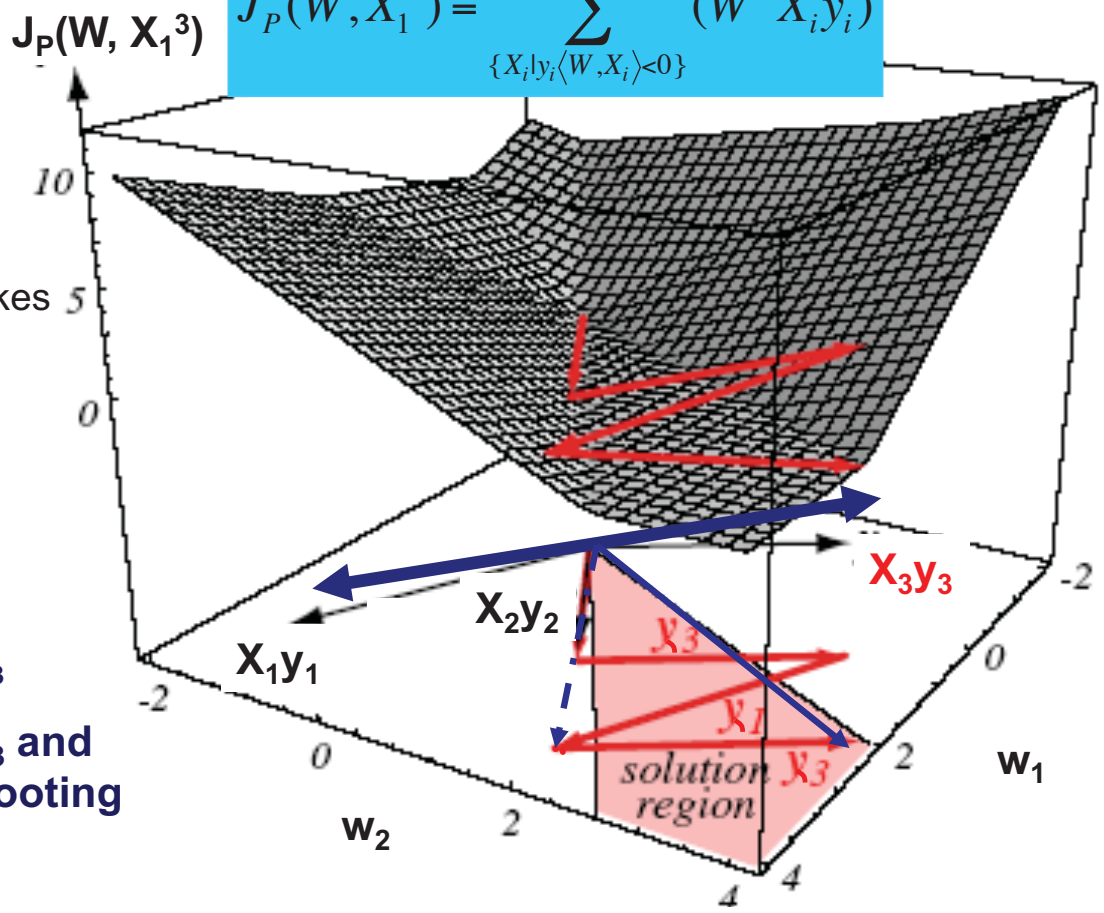


Pred	Actual	Margin	
+	+	+	(GOOD)
+	-	-	LOSS
-	+	-	LOSS
-	-	+	(GOOD)

Perceptron Update Example

Initialize W to [0,0]

$$J_P(W, X_1^L) = \sum_{\{X_i | y_i \langle W, X_i \rangle < 0\}} (W^T X_i y_i)$$



negative for mistakes

$$W' = W - (-Xy_i)$$

$$W' = W + Xy_i$$

Start with W = [0, 0]
Update Sequence:
X_2, X_3, X_1, X_3

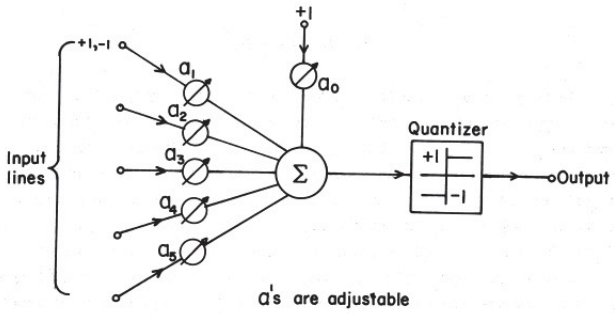
Updating with X_3y_3 and X_2y_1 cause overshooting

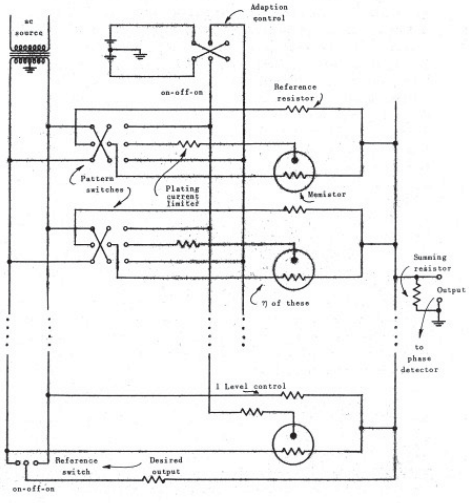
Perceptron machine: updates performed by electric motors

- The perceptron algorithm was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt,[3] funded by the United States Office of Naval Research.[4]
- The perceptron was intended to be a machine, rather than a program, and while its first implementation was in software for the IBM 704, it was subsequently implemented in custom-built hardware as the "Mark 1 perceptron".
- This machine was designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors

Adaline/Madaline

Hardware based
Stacked Perceptron
MLP





Widrow and Hoff, ~1960: Adaline/Madaline

Many ADALINE: MultiLayer sign function neurons

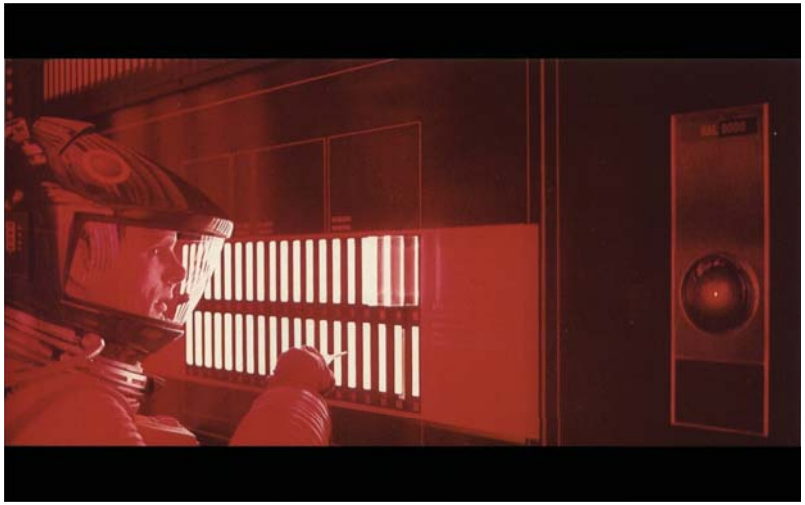
- ADALINE[1] It was developed by Professor Bernard Widrow and his graduate student Ted Hoff at Stanford University in 1960. Perceptron like but update rule was different

- MADALINE (Many ADALINE[1]) is a three-layer (input, hidden, output), fully connected, feed-forward artificial neural network architecture for classification that uses ADALINE units in its hidden and output layers, i.e. its activation function is the sign function.[2]

Many ADALINE: MultiLayer sign function neurons

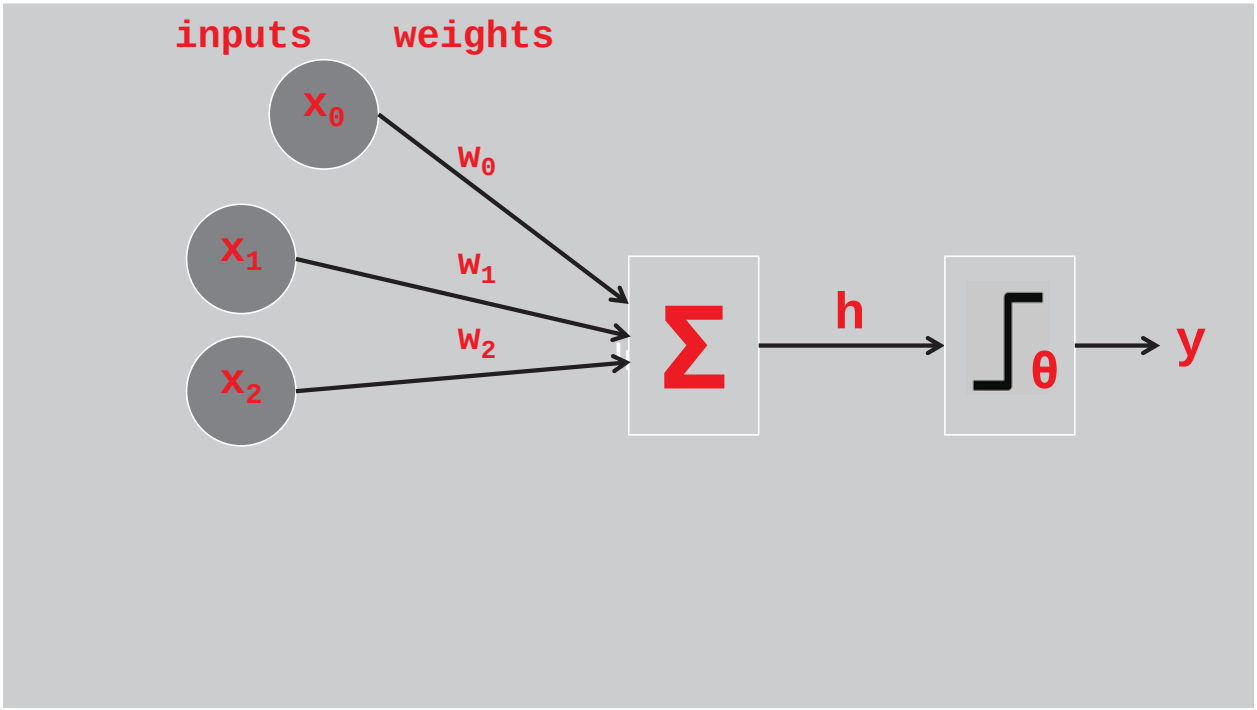
- Three different training algorithms for MADALINE networks, which cannot be learned using backpropagation because the sign function is not differentiable, have been suggested, called Rule I, Rule II and Rule III.
  - The first of these dates back to 1962 and cannot adapt the weights of the hidden-output connection.[3]
  - The second training algorithm improved on Rule I and was described in 1988.[1]
  - The third "Rule" applied to a modified network with sigmoid activations instead of signum; it was later found to be equivalent to backpropagation.[3]

Perceptrons are The Future!

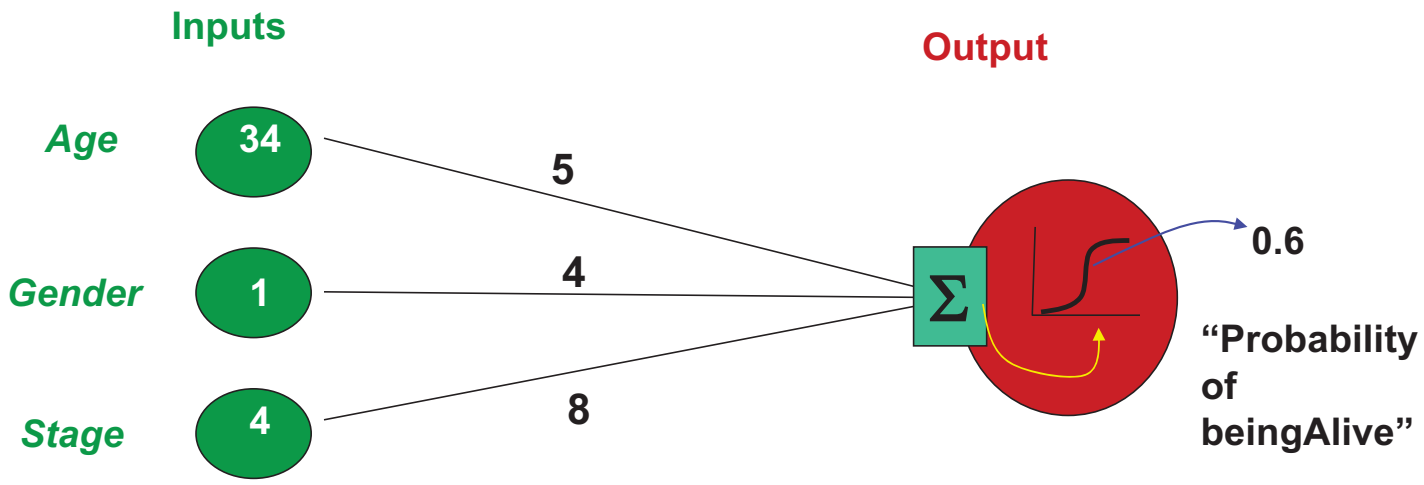


~1700s	• The prologue
1940-1970	• Act 1 Tinker: Hack it up
1970-1995	• Act 2 BackProp: theory to the rescue
1995-2007	• Act 3 Layer by layer learning, a medieval pastime
2007-2015	• Act 4 Introspection: better init. and activation functions
2015 -	• Act 5 Express-laning the gradient: Skip Connections, the SoTA frontier (LSTMs, ResNet, Highway Nets, DenseNets)
	• The epilogue

Perceptron Networks



Logistic Regression Model



Independent variables

x1, x2, x3

Coefficients

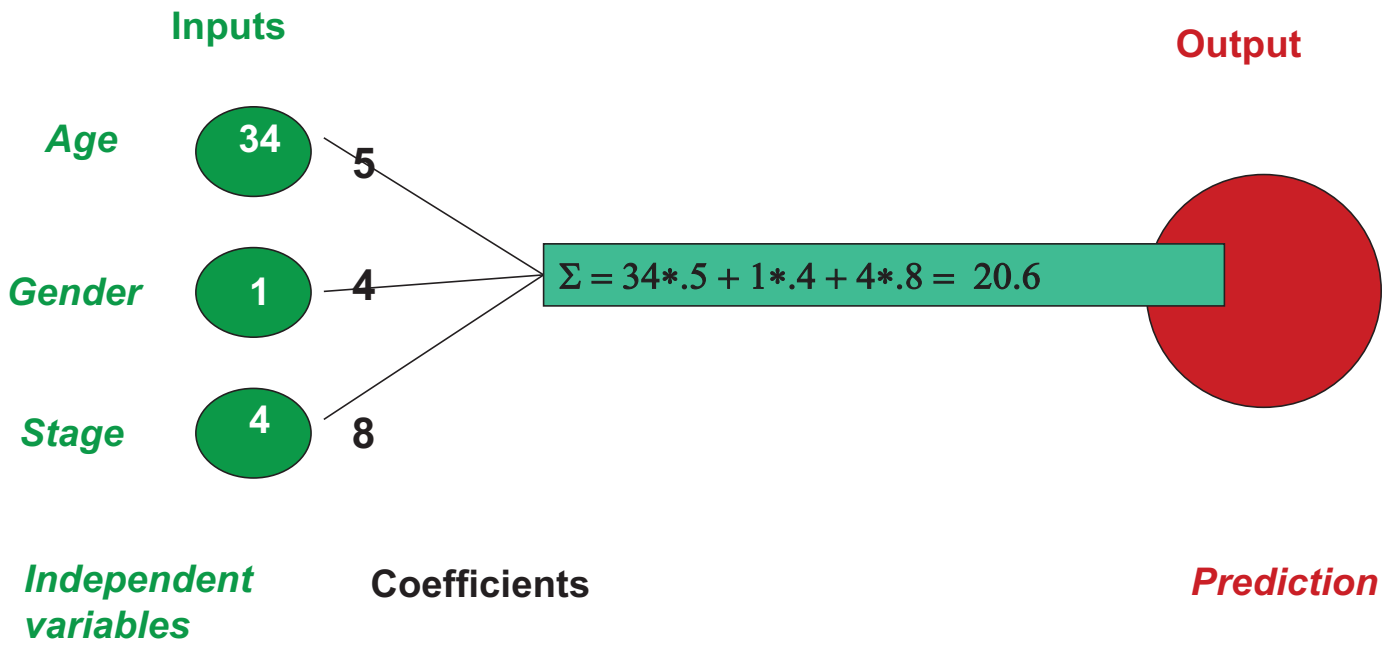
a, b, c

Dependent variable

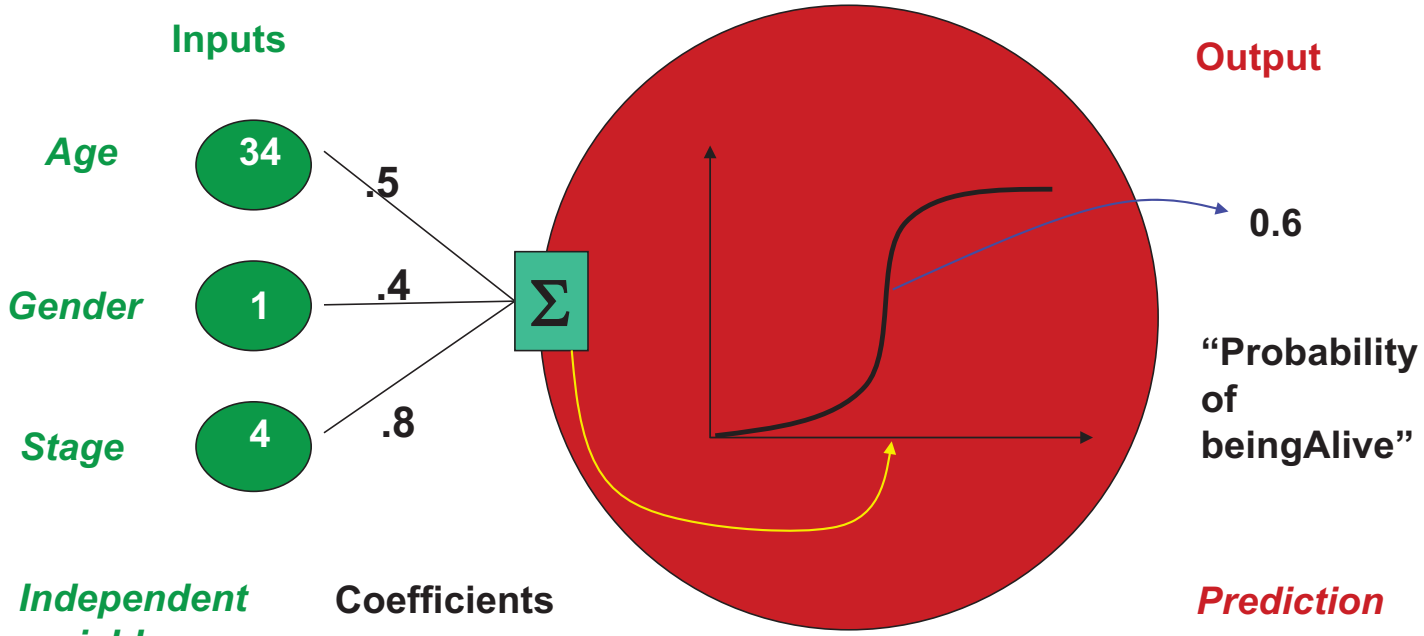
p

Prediction

Σ is the sum of inputs * weights



Logistic function



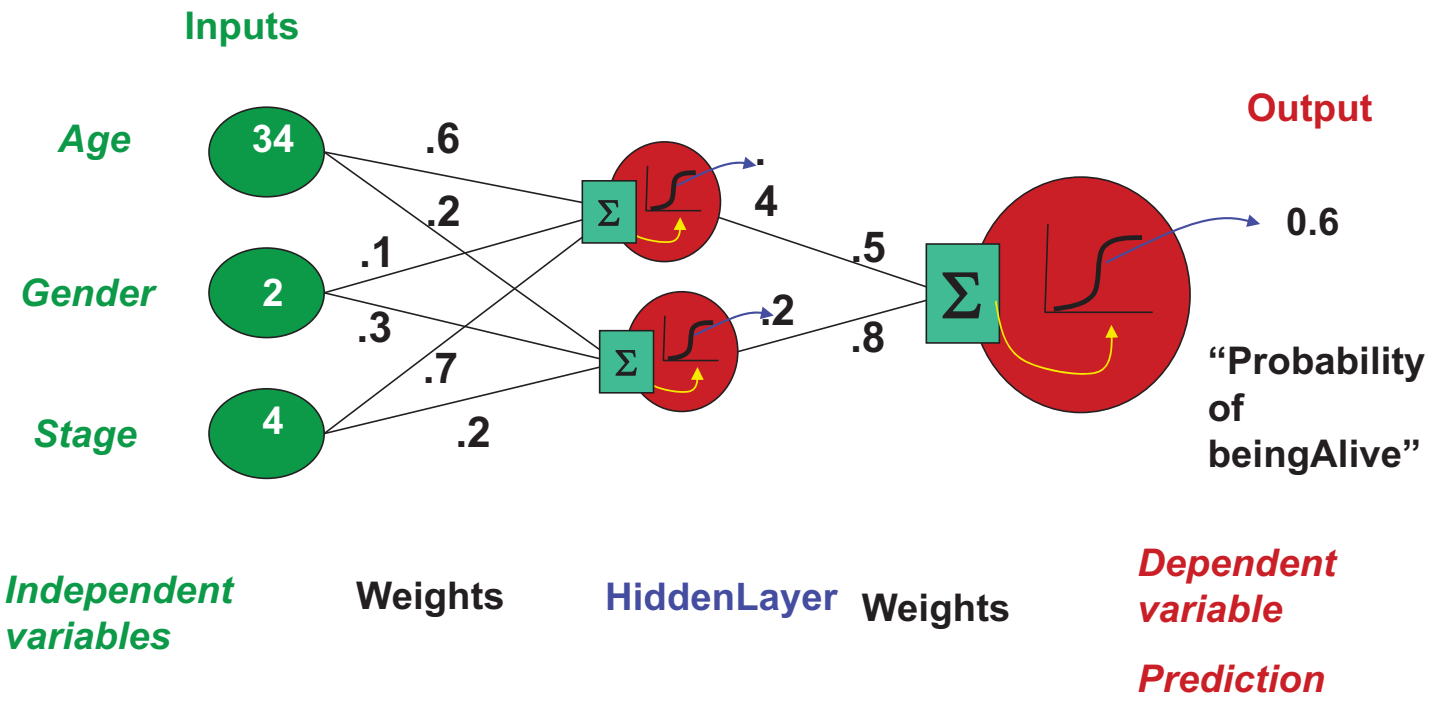
$$p = \frac{1}{1 + e^{-XW}}$$

Activation Functions...

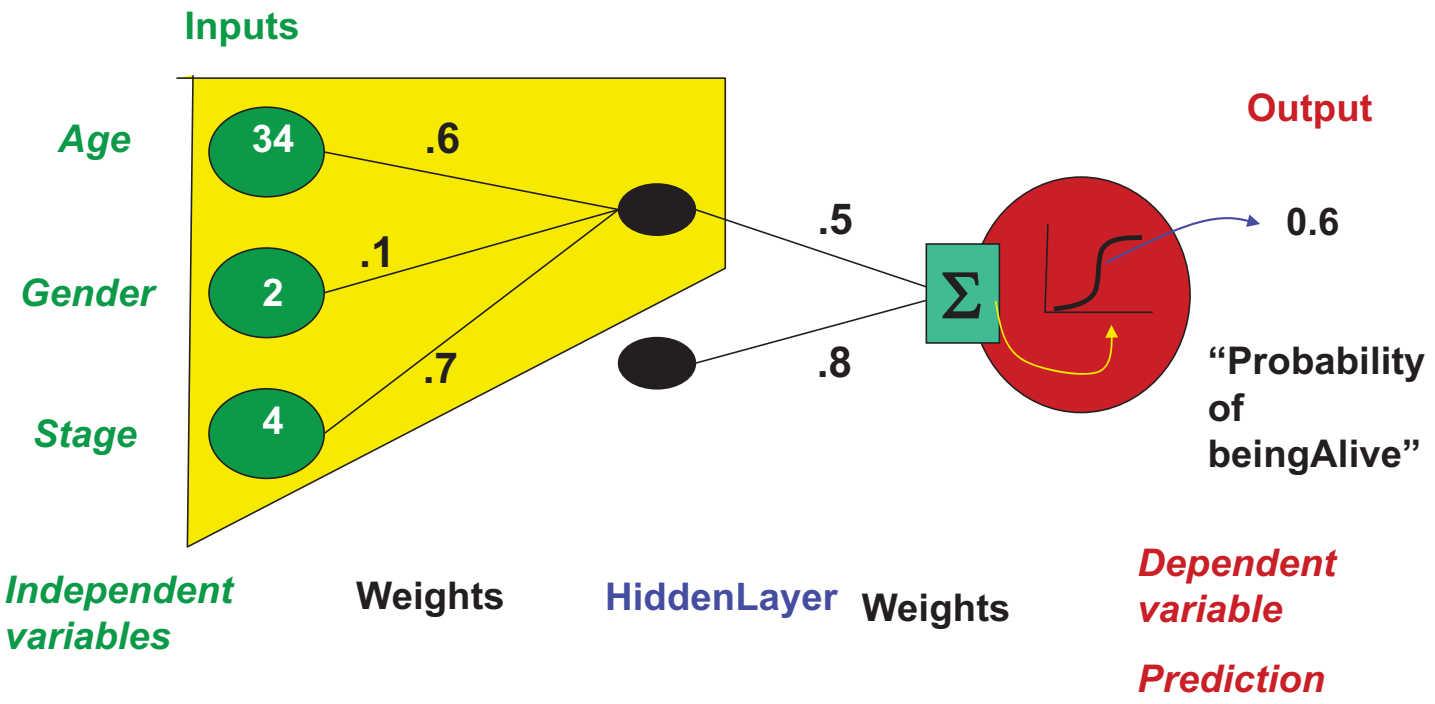
- Linear

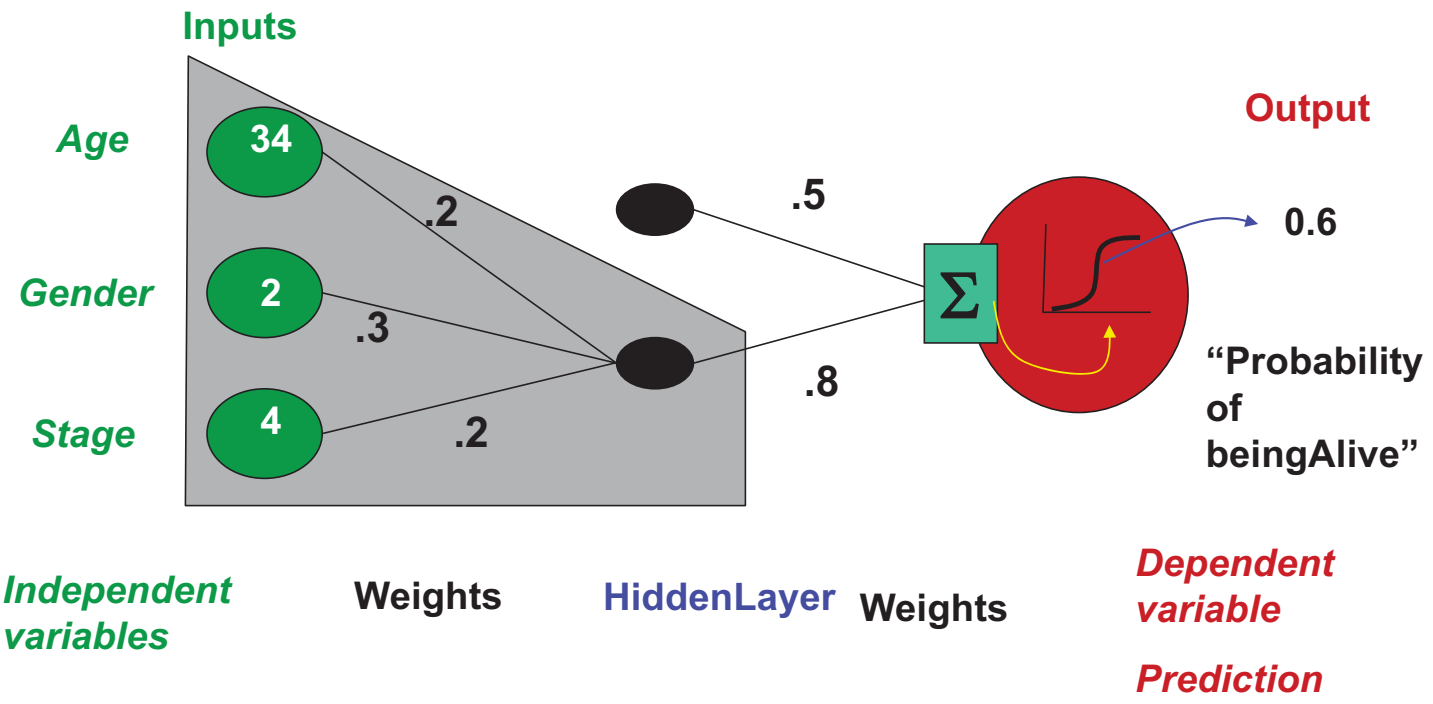
- Threshold or step function
- Logistic, sigmoid, “squash”
- Hyperbolic tangent

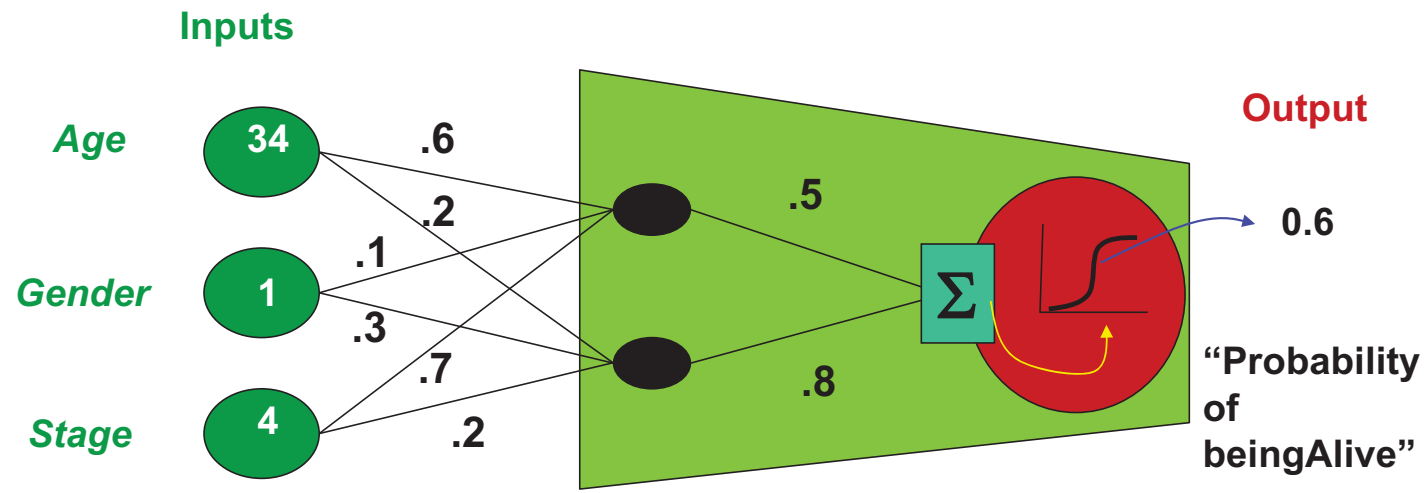
Neural Network Model: → nonlinear model



“Combined logistic models”

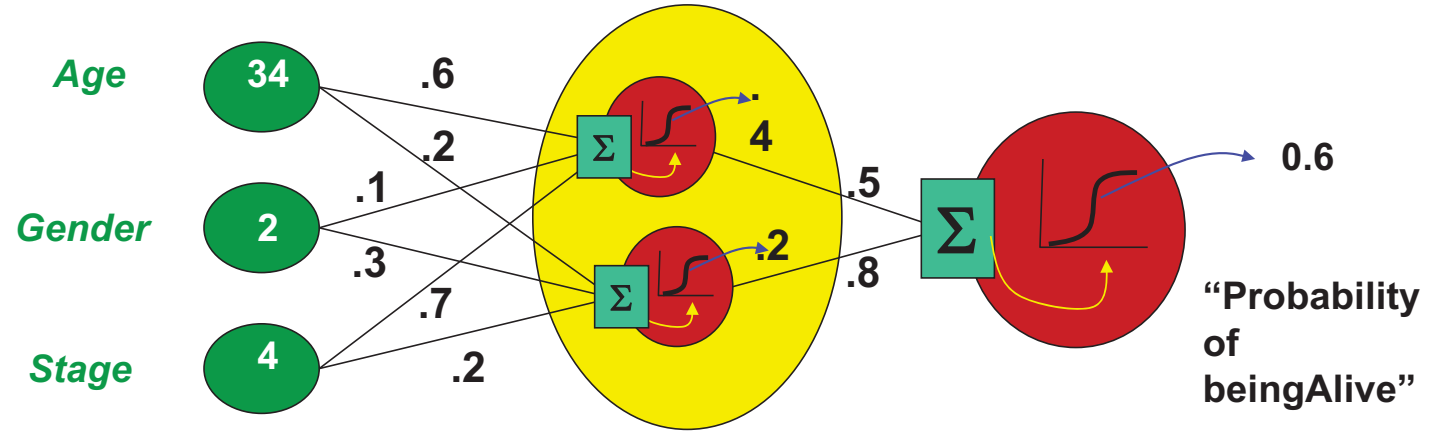



Inputs
Age
Gender
Stage
34
2
4
.2
.3
.2
.5
.8
Σ
Output
0.6
"Probability of beingAlive"
Independent variables
Weights
HiddenLayer
Weights
Dependent variable
Prediction


Inputs
Age
34
Gender
1
Stage
4
.6
.2
.1
.3
.7
.2
.5
.8
Σ
Output
0.6
"Probability of beingAlive"
Independent variables
Weights
HiddenLayer
Weights
Dependent variable
Prediction

no target for hidden units...

Hidden neurons help us model feature interactions
See manifold section below for more details



Independent variables

Weights

HiddenLayer

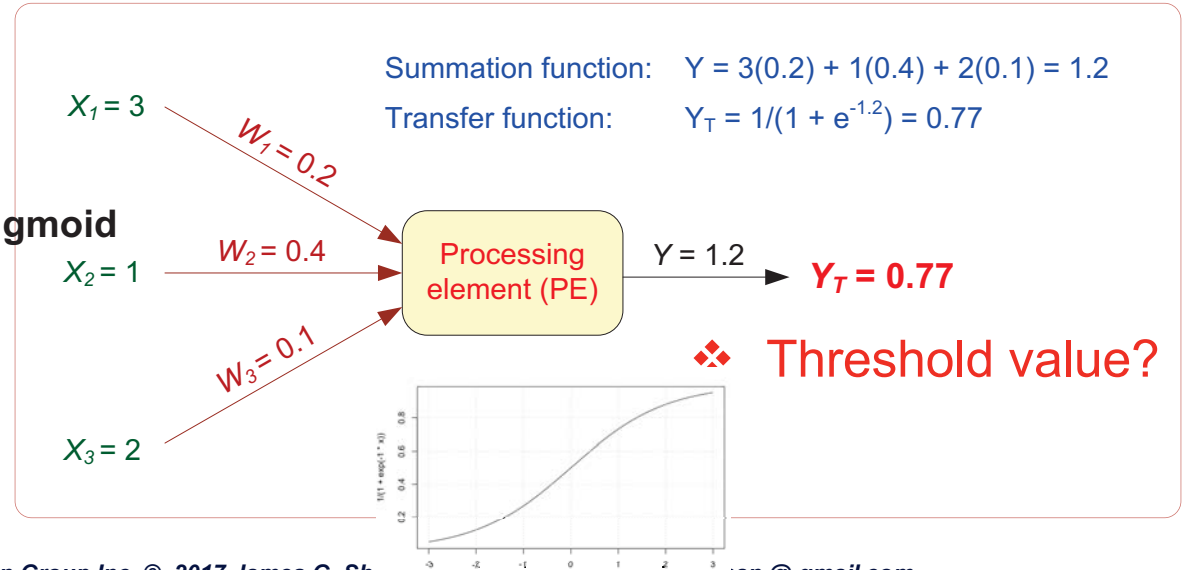
Weights

Dependent variable

Prediction

Elements of ANN

- **Transformation (Transfer) Function**
  - Linear function
  - Sigmoid (logical activation) function [0 1]
  - Tangent Hyperbolic function [-1 1]



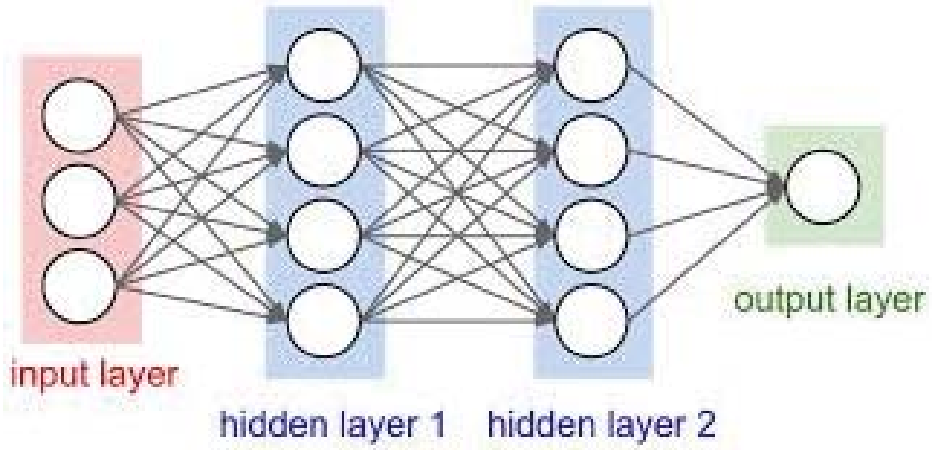
Exp versus Sigmoid

Neural Network Architectures

- **Several ANN architectures exist**
  - Feedforward
  - Autoencoders
  - Recurrent
  - Associative memory
  - Probabilistic
  - Self-organizing feature maps
  - Hopfield networks
  - … many more …

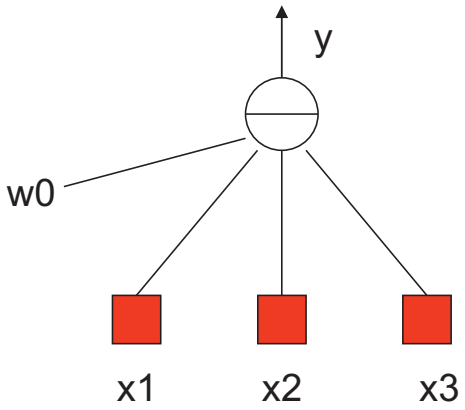
1980s style architectures

MLP, feed forward neural network



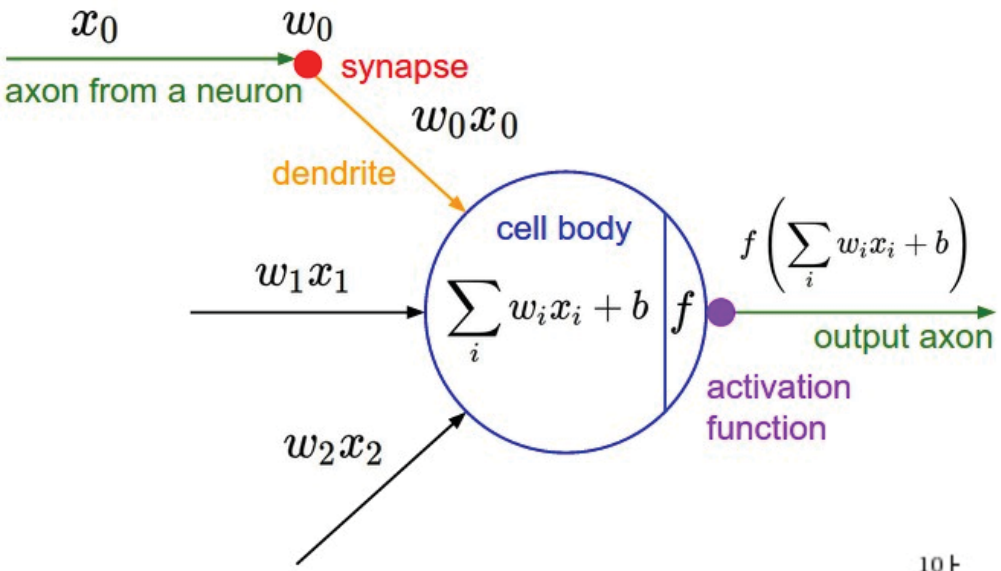
What is an artificial neuron ?

- **Definition : Non linear, parameterized function with restricted output range**

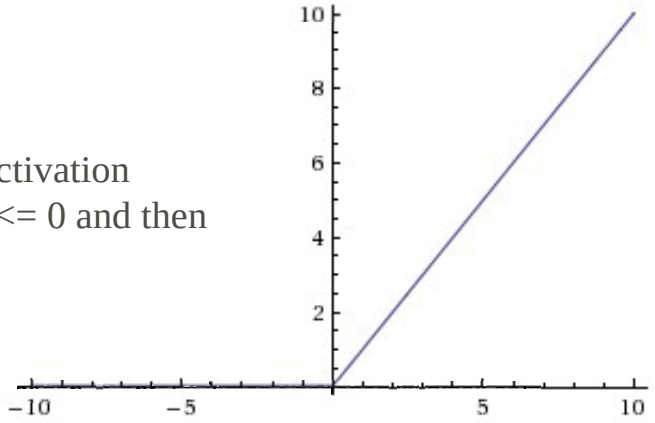


$$y = f\left(w_0 + \sum_{i=1}^{n-1} w_i x_i \right)$$

ReLU activation function



Rectified Linear Unit (ReLU) activation function, which is zero when x <= 0 and then linear with slope 1 when x > 0.



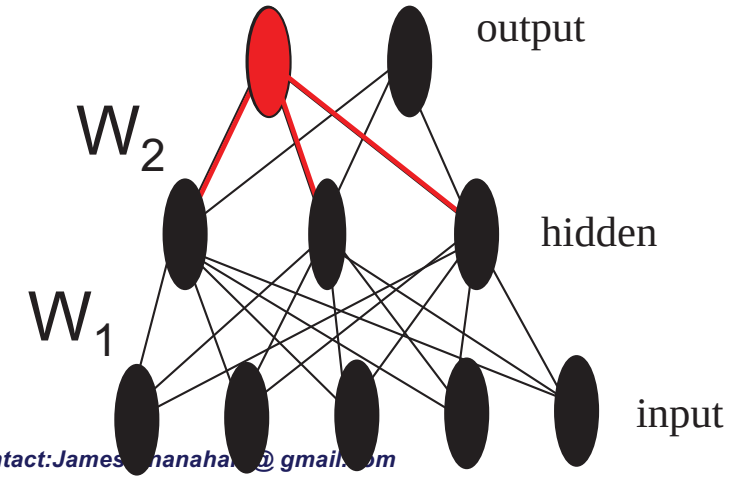
General (two-layer) feedforward network (k output units)

$g(W_2\ (f(W_1\ X)))$

- $$g_k(x) \equiv z_k = f_k\left(\sum_{j=1}^{n_H} w_{kj}\, f_j\left(\sum_{i=1}^{d} w_{ji} x_i + w_{j0}\right) + w_{k0}\right)$$

  $$(k = 1, \ldots, c)$$

  - The hidden units with their activation functions can express non-linear functions
  - The activation functions can be different at neurons (but the same one is used in practice)



Train NN using MAXIMUM LIKELIHOOD

- **Most modern neural networks are trained using maximum likelihood**

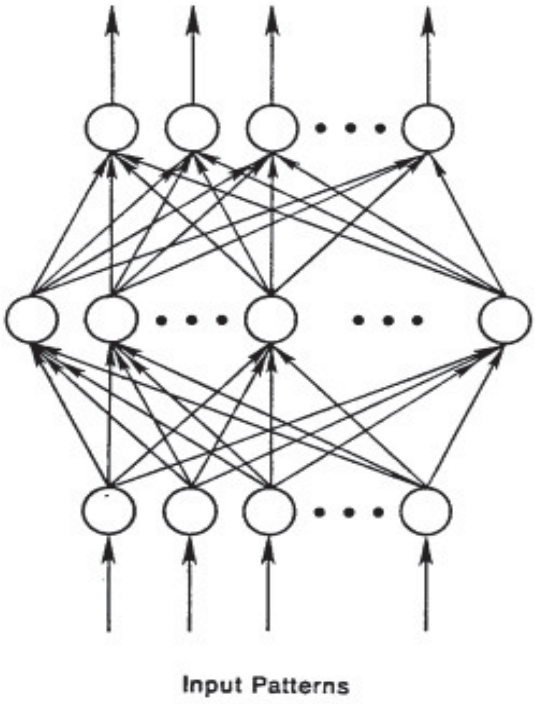
- **Minimizing MSE will output a maximum likelihood hypothesis**
  - A straightforward Bayesian analysis will show that under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will output a maximum likelihood hypothesis.
- **Minimizing cross entropy will output a maximum likelihood hypothesis**

See Tom Mitchell's ML Book, 1997 (section 6.4)
See Goodfellow et al. DL Book, 2016 (section 6.2.1)

From linear to non-linear models
From Convex to non-convex optimization

- **Nonlinearity of a neural network causes loss function to become non-convex**
  - The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex.
- **Use gradient descent finds a local minimum (not a global one)**
  - This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs.
  - Convex optimization converges starting from any initial parameters (in theory—in practice it is very robust but can encounter numerical problems).

BackProp



To be more specific, then, let

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2 \quad (2)$$

be our measure of the error on input/output pattern p and let $E = \sum E_p$ be our overall measure of the error. We wish to show that the delta rule implements a gradient descent in E when the units are linear. We will proceed by simply showing that

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_{pi},$$

which is proportional to $\Delta_p w_{ji}$ as prescribed by the delta rule. When there are no hidden units it is straightforward to compute the relevant derivative. For this purpose we use the chain rule to write the derivative as the product of two parts: the derivative of the error with respect to the output of the unit times the derivative of the output with respect to the weight.

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ji}}. \quad (3)$$

The first part tells how the error changes with the output of the j th unit and the second part tells how much changing w_{ji} changes that output. Now, the derivatives are easy to compute. First, from Equation 2

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) = -\delta_{pj}. \quad (4)$$

Not surprisingly, the contribution of unit u_j to the error is simply proportional to δ_{pj}. Moreover, since we have linear units,

$$o_{pj} = \sum_i w_{ji} i_{pi}, \quad (5)$$

from which we conclude that

$$\frac{\partial o_{pj}}{\partial w_{ji}} = i_{pi}.$$

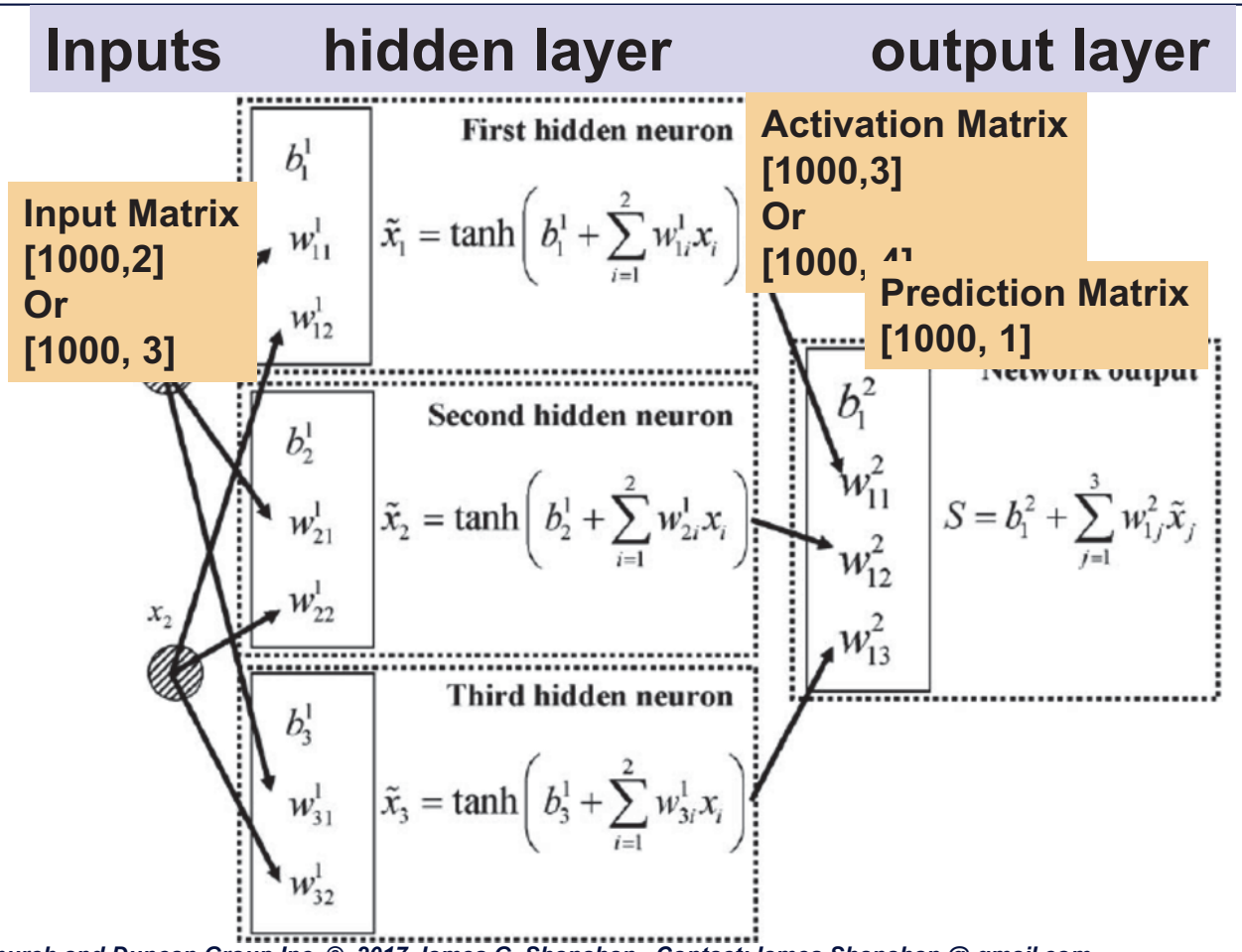
Thus, substituting back into Equation 3, we see that

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_i \quad (6)$$

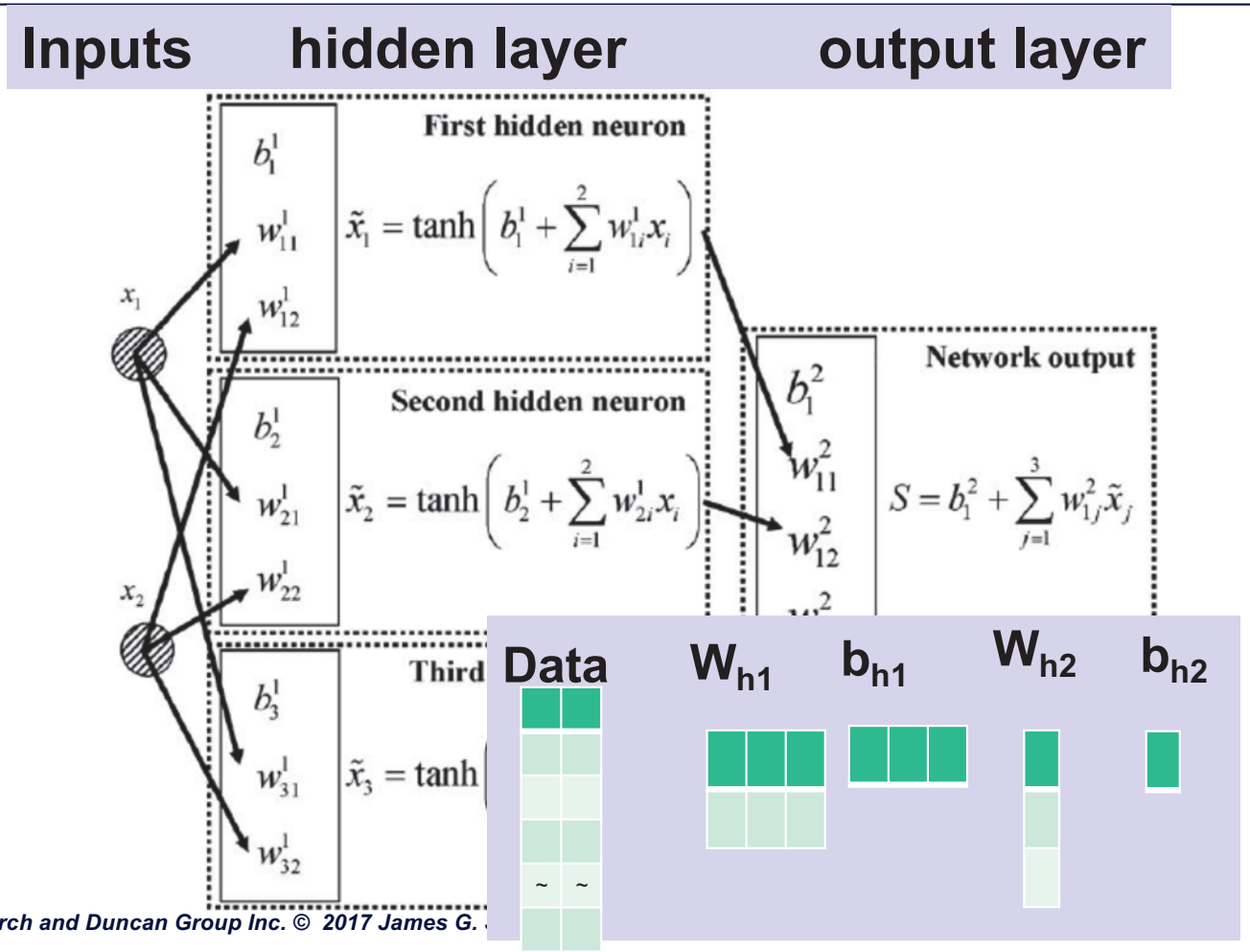
Principled credit assignment using a loss function and update rule

Rumelhart et al. 1986: First time back-propagation became popular

Activation functions in NN: 2-3-1

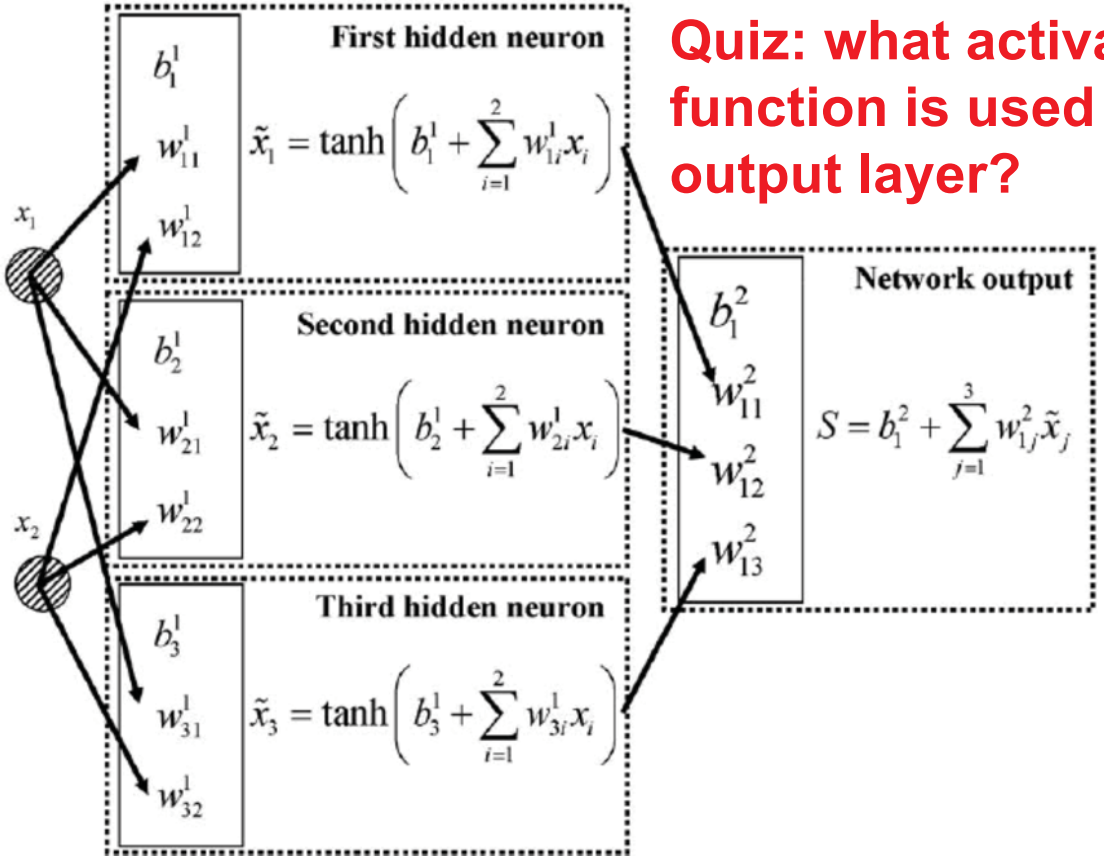


Activation functions in NN: 2-3-1



Activation functions in NN: 2-3-1

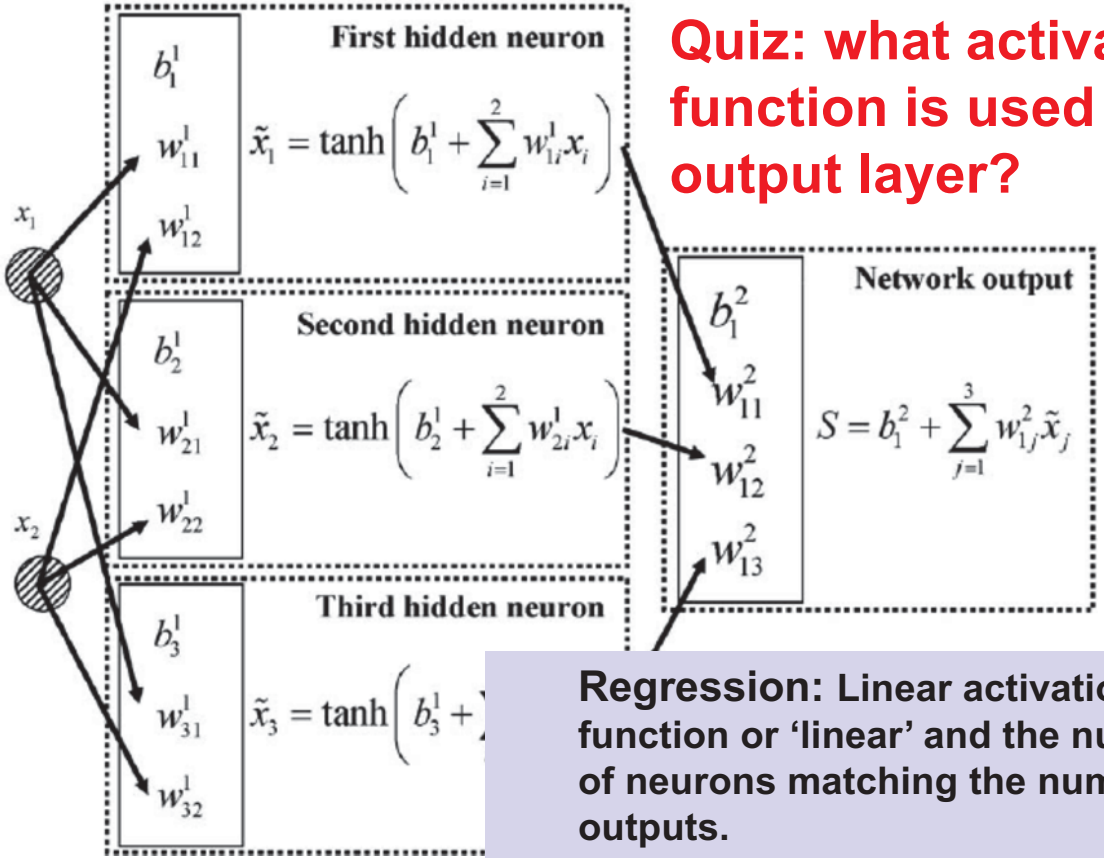
Inputs **hidden layer** **output layer**



Quiz: what activation function is used for the output layer?

Activation functions in NN: 2-3-1

Inputs **hidden layer** **output layer**

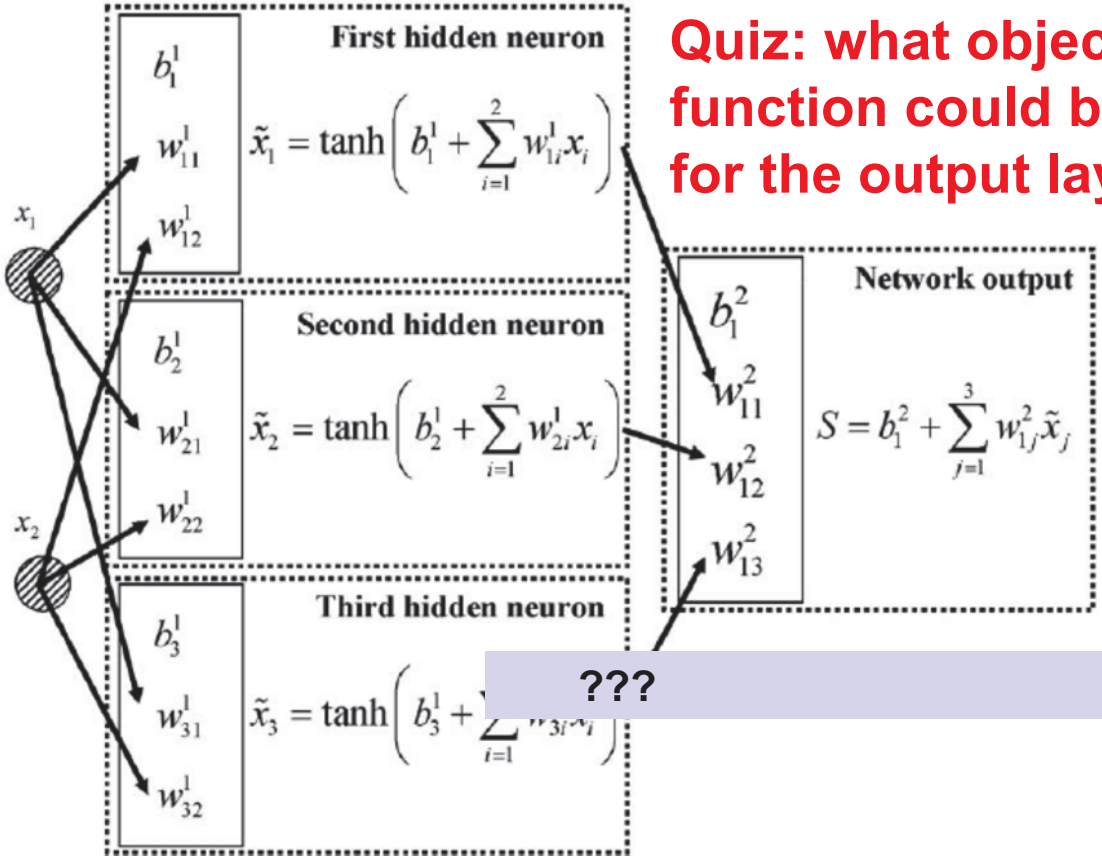


Quiz: what activation function is used for the output layer?

Regression: Linear activation function or 'linear' and the number of neurons matching the number of outputs.

Activation functions in NN: 2-3-1

Inputs **hidden layer** **output layer**

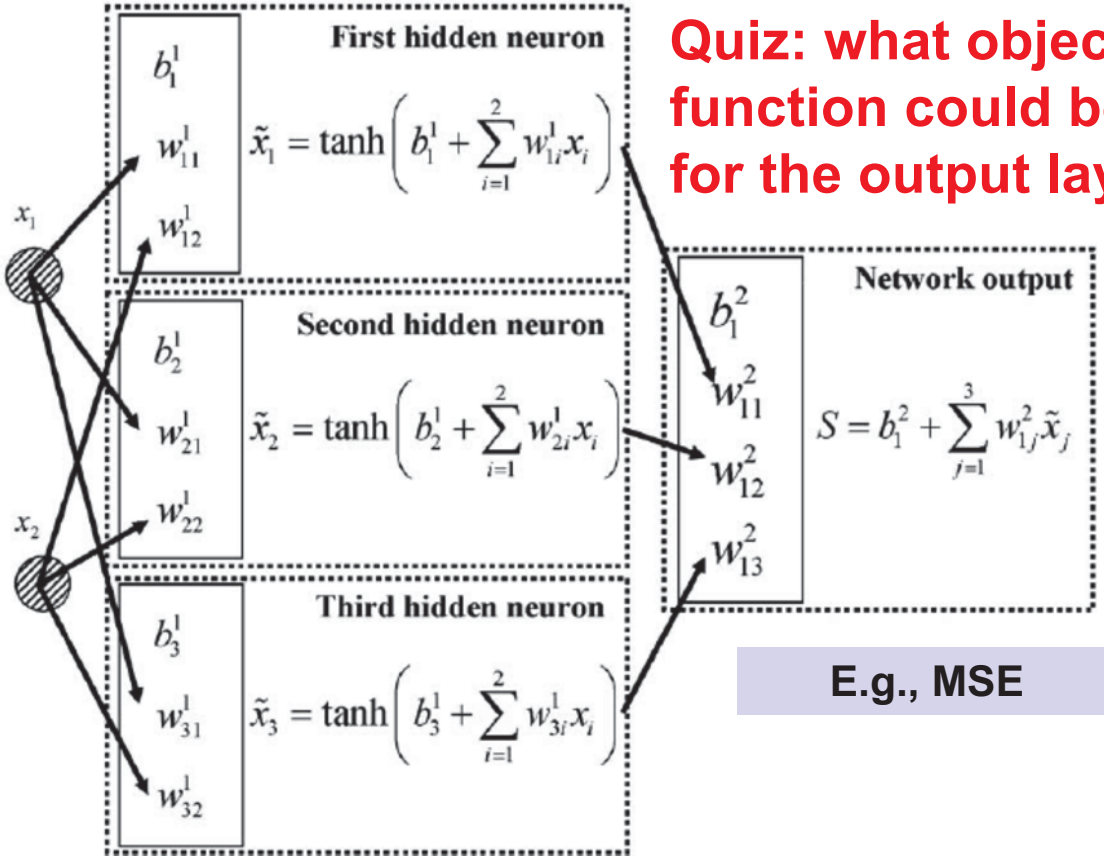


Quiz: what objective function could be used for the output layer?

???

Activation functions in NN: 2-3-1

Inputs **hidden layer** **output layer**

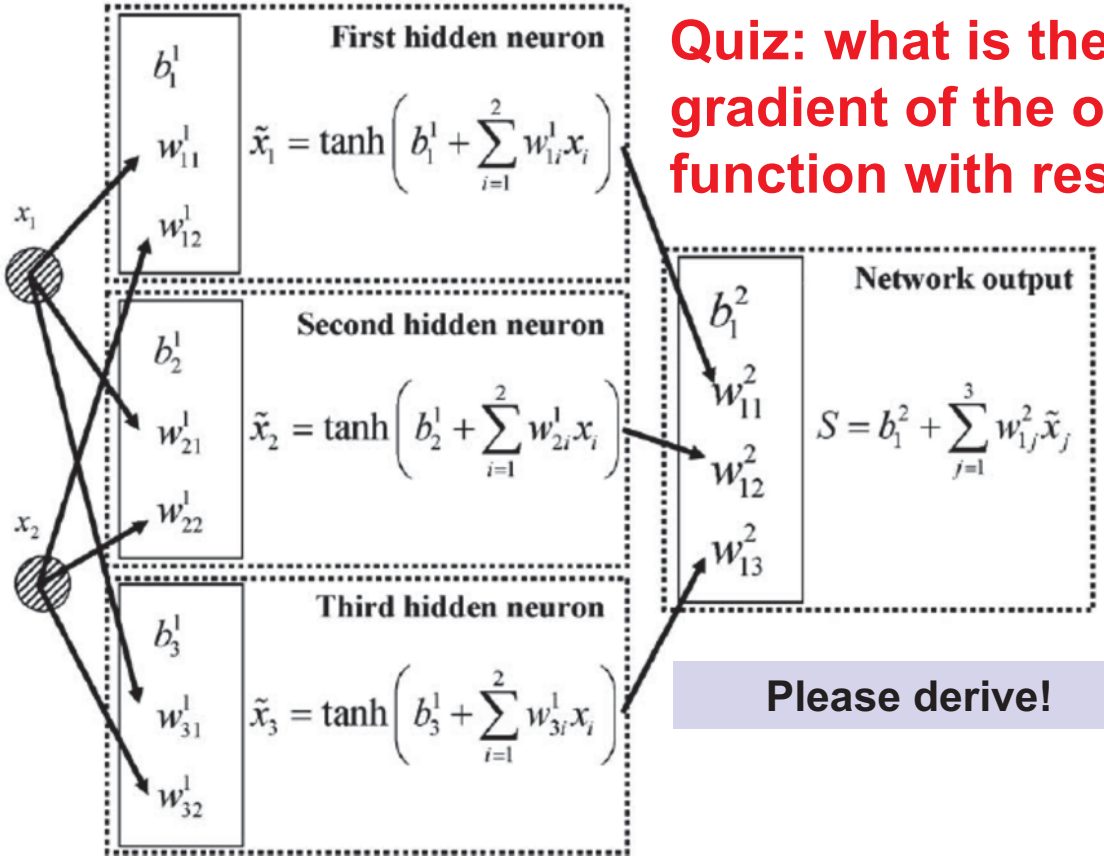


Quiz: what objective function could be used for the output layer?

E.g., MSE

Activation functions in NN: 2-3-1

Inputs **hidden layer** **output layer**

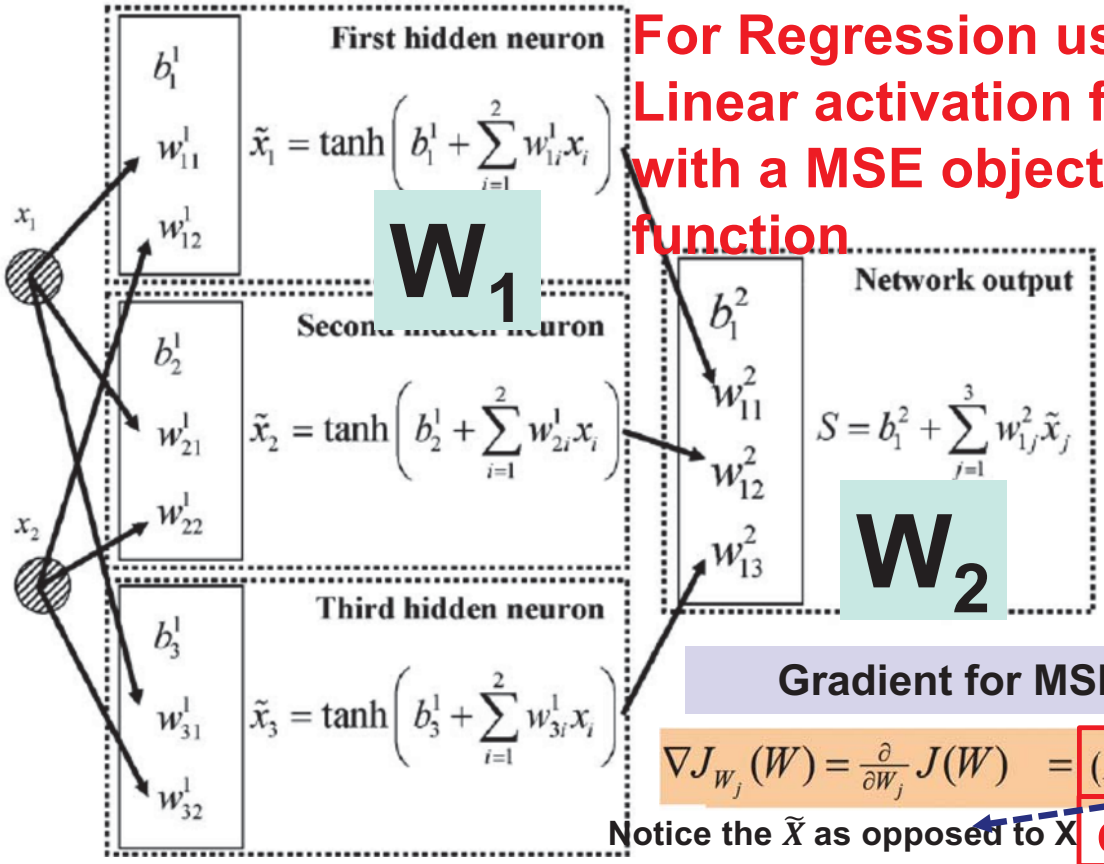


Quiz: what is the gradient of the objective function with respect w_i?

Please derive!

Activation functions in NN: 2-3-1

Inputs **hidden layer** **output layer**



W_1

W_2

For Regression use a Linear activation function with a MSE objective function

NOTE: gradient is NOT fully specified; see later slides (missing activation function)

Gradient for MSE wrt W

$$\nabla J_{W_j}(W) = \frac{\partial}{\partial W_j} J(W) = (\tilde{X}W_2 - Y)^T \tilde{X}$$

Notice the $\tilde{X}$ as opposed to X

error

Gradient Descent for NN

- **Given a set of training data points with target t_k and output layer output O_k we can write the error as**

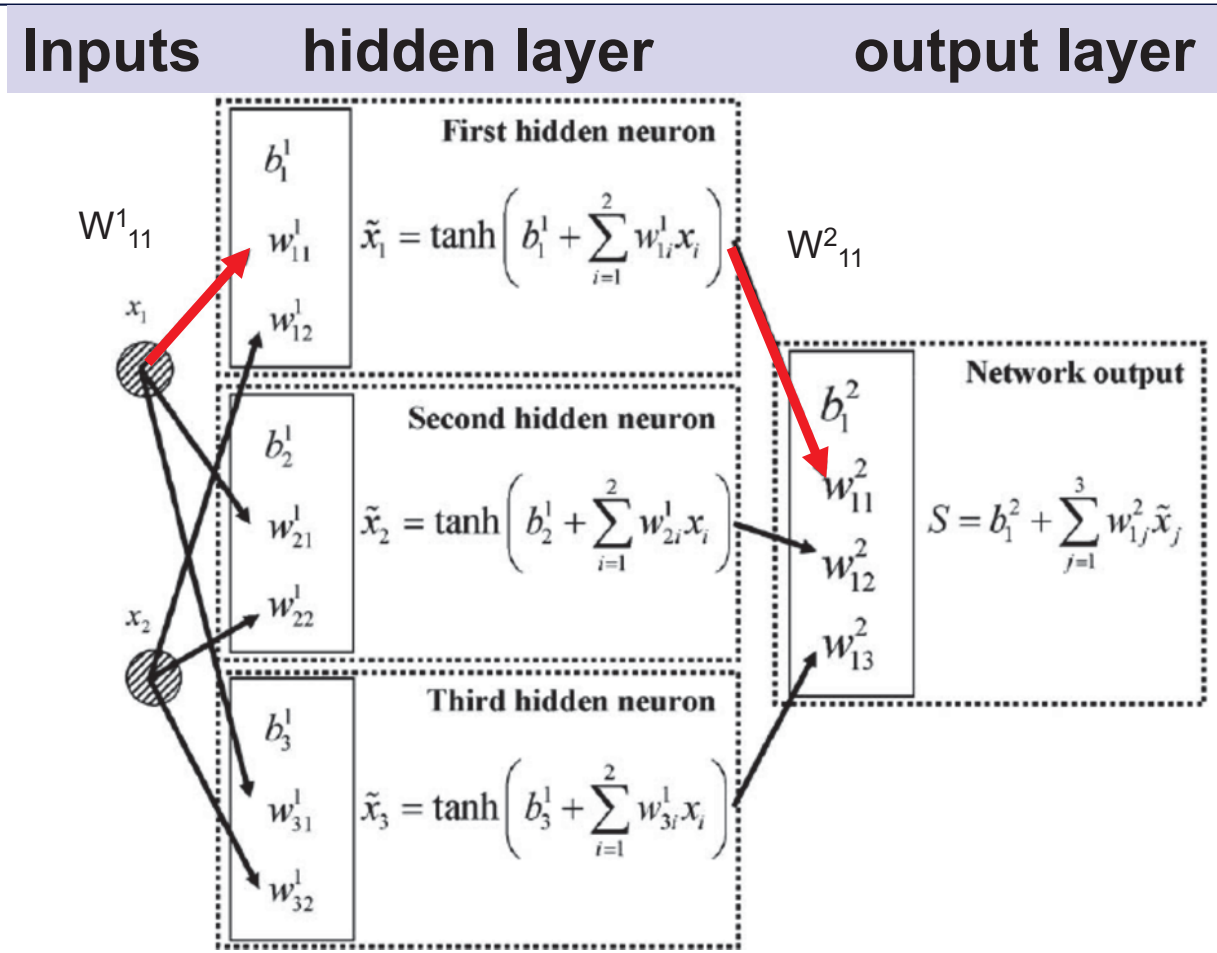
$$E = \frac{1}{2}\sum_{k \in K}(O_k - t_k)^2$$

- **How can we train the network via Gradient descent? We need to calculate the Gradient ∇ (the vector of partial derivatives)**

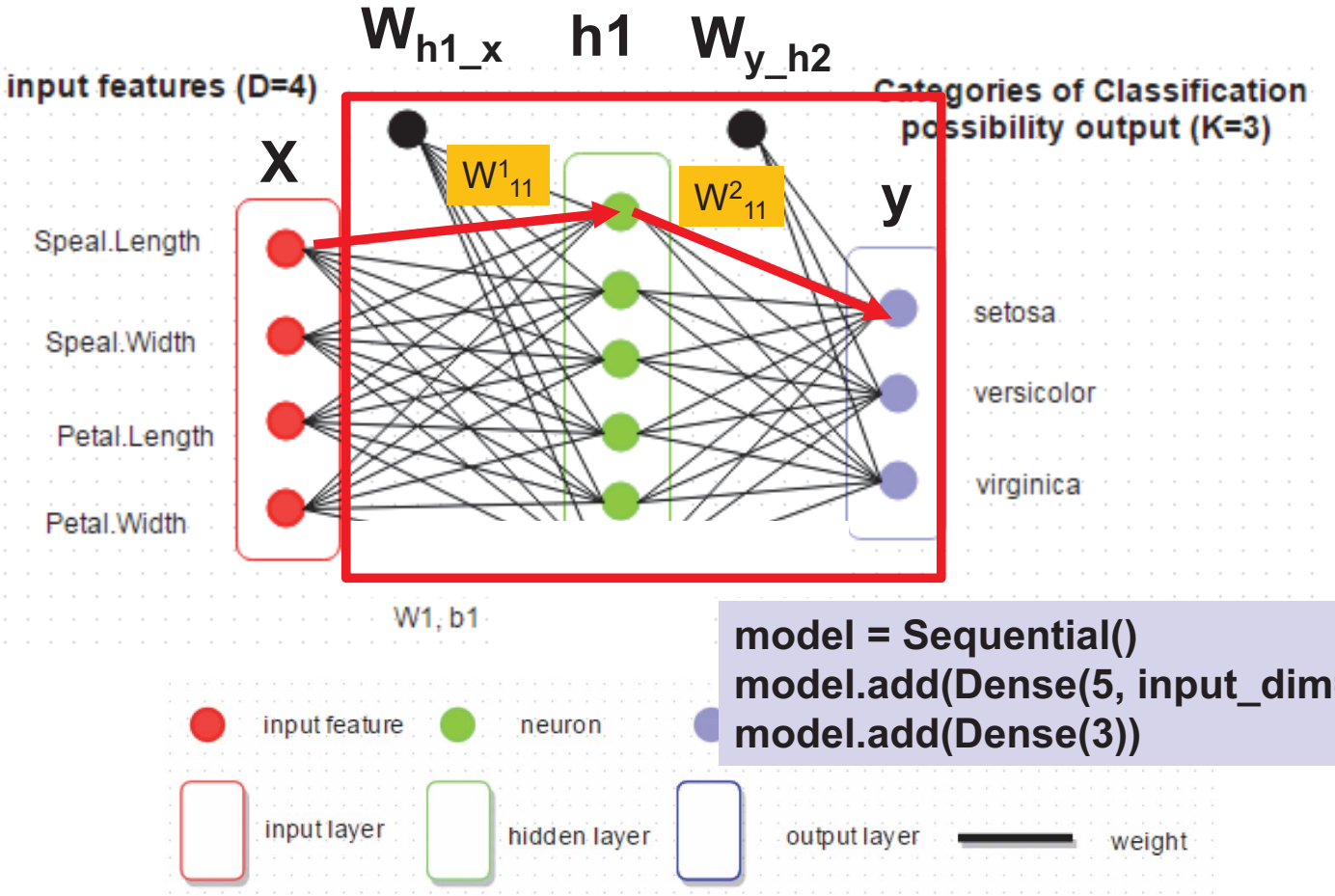
$$\frac{\partial E}{\partial W_{ij}}$$

- **Now we consider two cases:**
  - Weights associated with an output layer node
  - Weights associated with a hidden layer node...

Activation functions in NN: 2-3-1

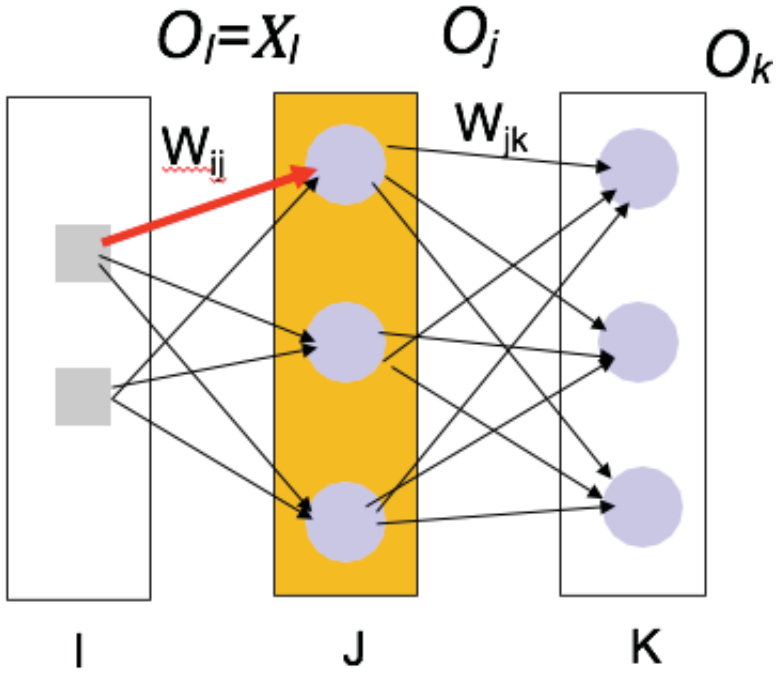


Let's work with a 4 – 5 – 3 MLP

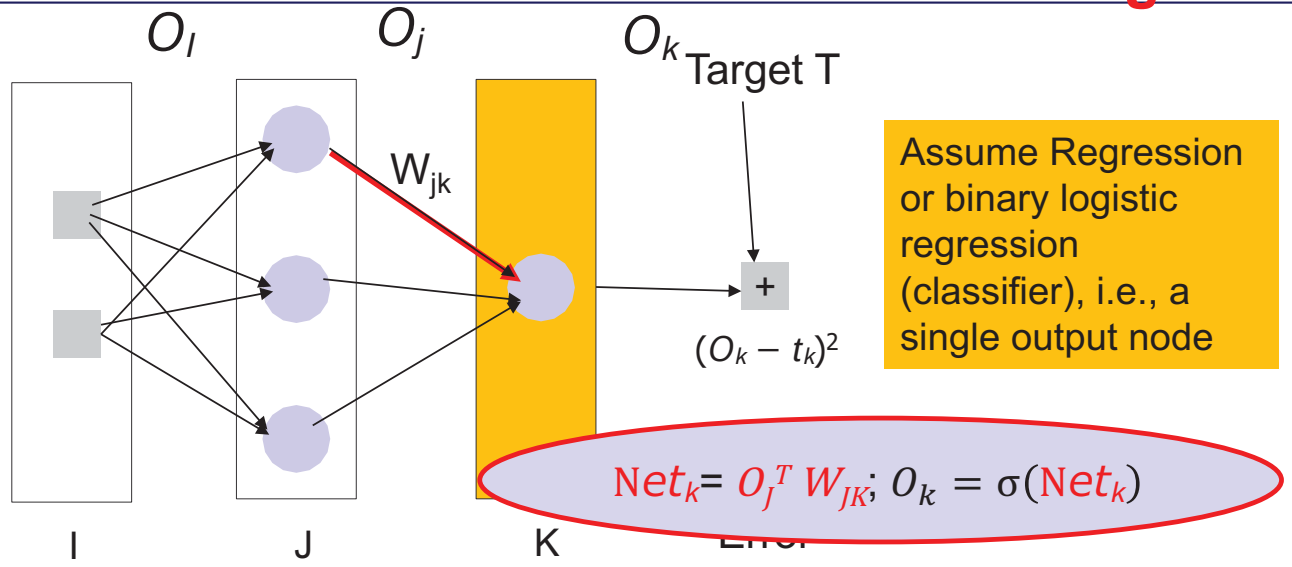



```
model = Sequential()
model.add(Dense(5, input_dim=4))
model.add(Dense(3))
```

network structure



Gradient Descent
Case 1:Outer node weight



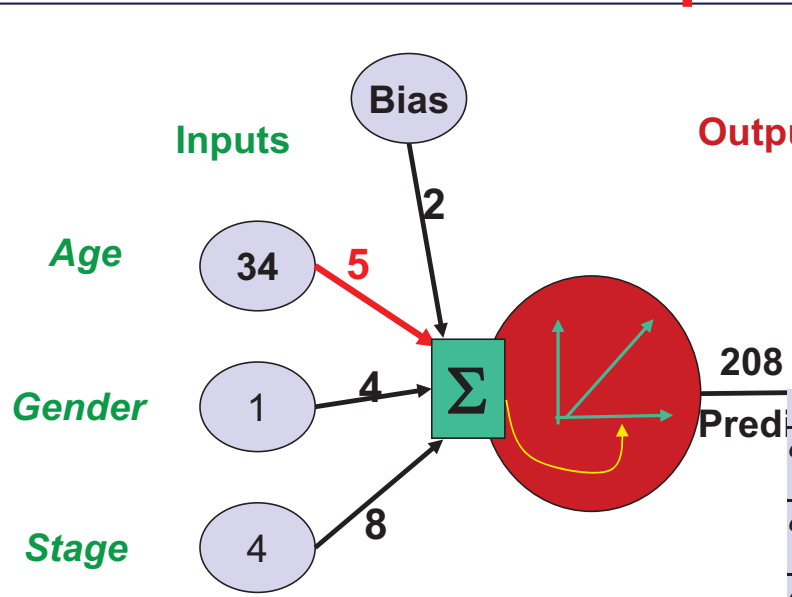
Notation:

Each layer, *I, J,* and *K,* consists of nodes that have in forward prop mode (prediction):

- A Weighted sum is denoted as Net_J (i.e.,$X^T W_J$);
- an activation $\sigma(X_k)$ where the activation func is sigmoid

In BackProp mode: each node has error term, δ_k

Linear Regression Model: Gradient update for one example



Target

206

$$W^T = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix}$$

Error

$$\frac{\partial E}{\partial W_i} = \frac{\partial}{\partial W_i} \sum_{n=1:Train} \frac{1}{2} (X_{in} W^T - t_{in})^2$$

$$\frac{\partial E}{\partial W_i} = \frac{\partial}{\partial W_i} \frac{1}{2} (O_i - t_i)^2$$

$$\frac{\partial E}{\partial W_i} = (O_i - t_i) \frac{\partial}{\partial W_i} (O_i - t_i)$$

$$\frac{\partial E}{\partial W_i} = (O_i - t_i) \frac{\partial}{\partial W_i} (X_i W^T - t_i)$$

$$\frac{\partial E}{\partial W} = (O - t)\, X$$

$$W = W - \alpha \frac{\partial E}{\partial W}$$

$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} 34 \\ 1 \\ 4 \end{bmatrix}$$

$$4.32 = 5 - 0.01 \times (2 \times 34) \text{ for } i = 1$$

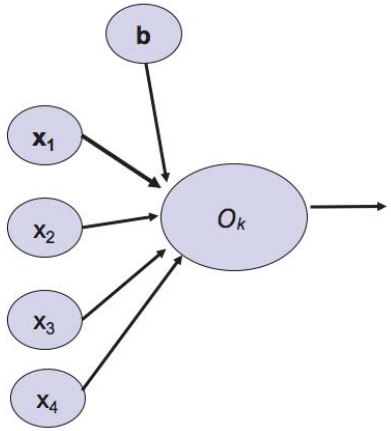
Gradient for this example[34,1,4]

Gradient for b = 2

NOTE: the gradient is a weighted sum of all examples; here the weight is 2 for this example

Assume a learning rate α=0.01

Activation matrices and model matrices for a single target variable



4 1 1 Prediction

150 X 4 W = 150 P

W = W -α ∇_W

∇_W = X E

Weighted Sum

Model

Data

$$\begin{bmatrix}4.32\\3.98\\7.92\end{bmatrix} = \begin{bmatrix}5\\4\\8\end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix}34\\1\\4\end{bmatrix}$$

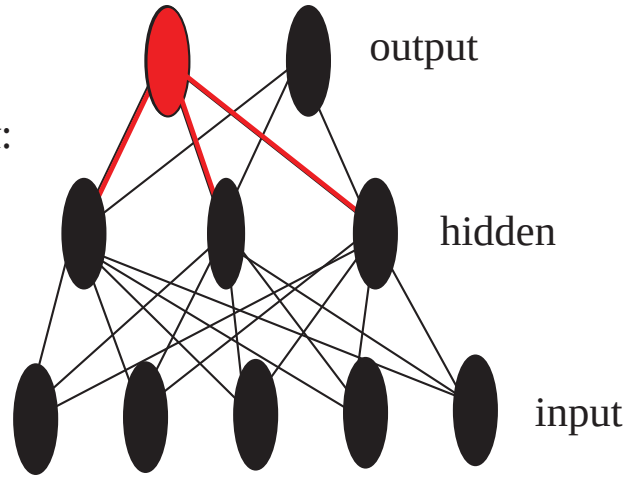
Backpropagation

Calculate the error signal for the output neurons and update the weights between the output and hidden layers

$$\delta_k = (t_k - z_k) f'(net_k)$$

update the weights to k:

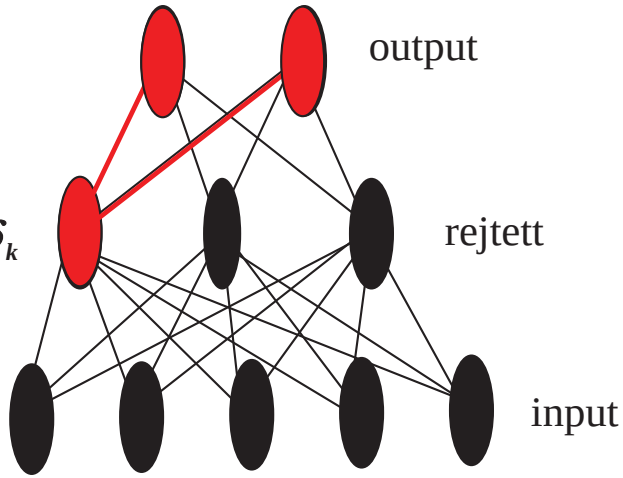
$$\Delta w_{ki} = \eta \delta_k z_i$$



Backpropagation

Calculate the error signal for hidden neurons

$$\delta_j \equiv f'(net_j)\sum_{k=1}^{c} w_{kj}\delta_k$$

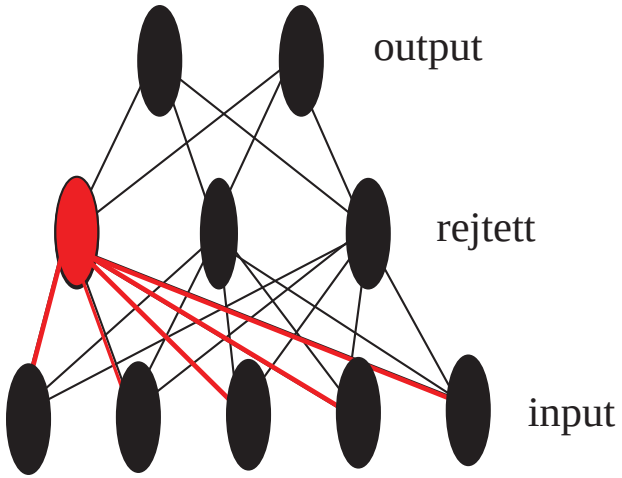


Backpropagation

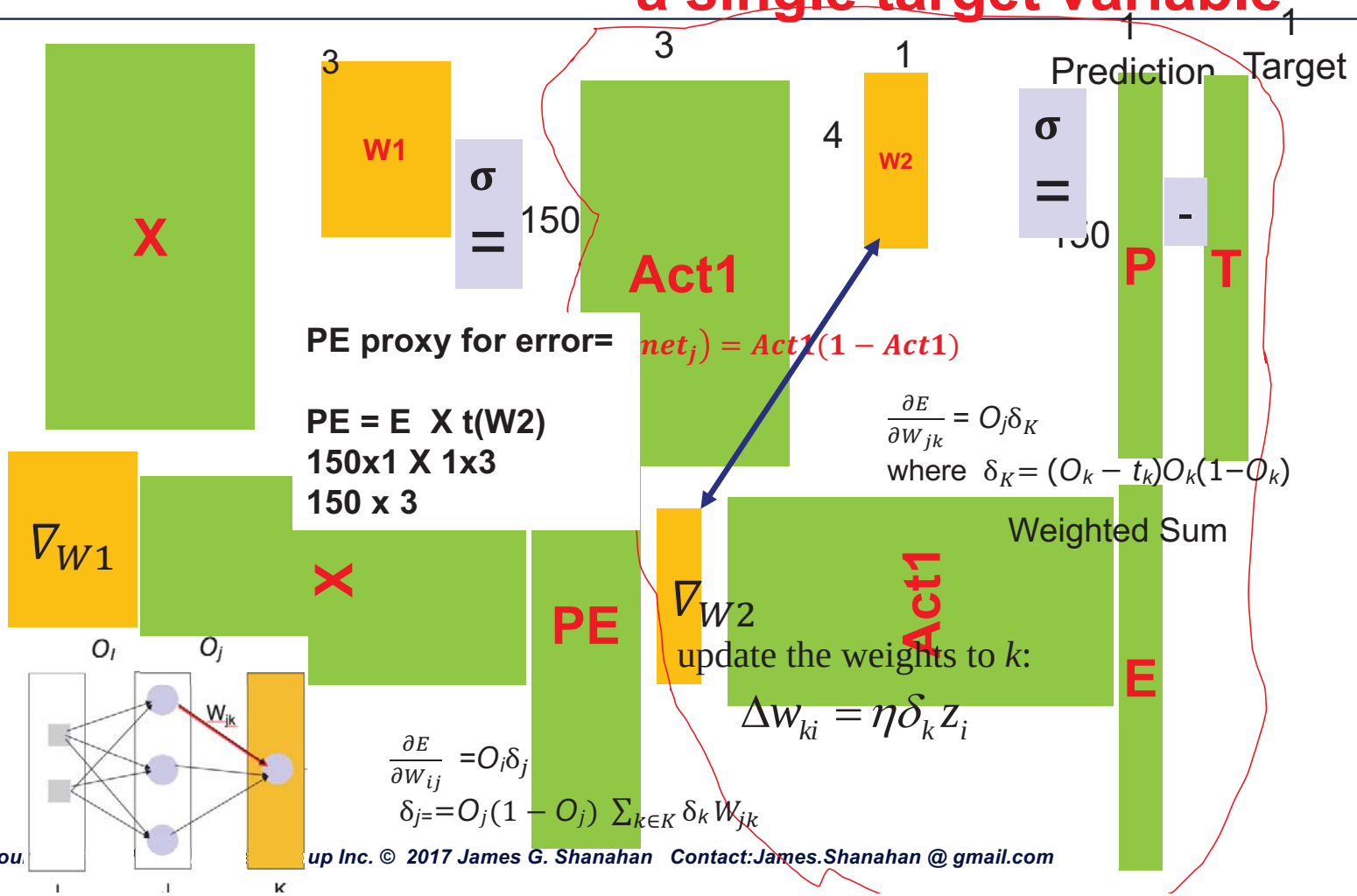
Update the weights between the input and hidden neurons

updating the ones to j

$$\Delta w_{ji} = \eta \delta_j x_i$$



Activation matrices and model matrices for a single target variable



Back propagation algorithm

1. Run the network forward with your input data to get the activations (O)
2. For each output node compute the error

$$\delta_k = O_k(1 - O_k)(O_k - t_k)$$

weight example

Gradient for MSE = $\frac{\partial E}{\partial W} = (O - t)\,X$

3. For each hidden node calulate credit assignment

$$\delta_J = O_j(1 - O_j) \sum_{k \in K} \delta_k W_{jk}$$

4. Update the weights and biases as follows:

For each layer l-1 (Letter elle) (each layer W_l, b_l)

weight example

$$\nabla W_l = -\eta \delta_l O_{l-1}$$

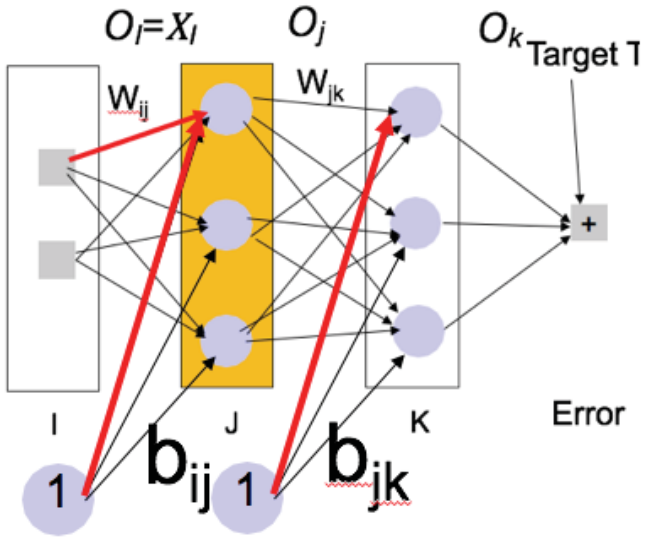
$$\nabla b_l = -\eta \delta_l$$

∇ Nabla

apply

$$W_l = W_l + \nabla W_l$$

$$b_l = b + \nabla b_l$$



▽ the gradient chorus

- **What is the gradient for linear regression?**

$$\frac{\partial E}{\partial W} = \underbrace{(O - t)}_{\text{weight}}\underbrace{X}_{\text{example}}$$

$$W = W - \alpha \times \frac{\partial E}{\partial W}$$

$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} 34 \\ 1 \\ 4 \end{bmatrix}$$

$$4.32 = 5 - 0.01 \times (2 \times 34) \text{ for } i = 1$$

- **Chorus**
  - The gradient is the weighted sum of the training data, where the weights are proportional to the error (for each example) !

Linear Regression

NN – The gradient is the weighted sum of the **input** data, where the weights are proportional to the error **or the error proxy** (for each example) !

Calculate derivatives using cached feed-forward activation values

- Compute activation once during in the forward propagation and then cache this activation matrix
- Use cached version for BackProp
- handy implementation tricks like calculating derivatives using cached feed-forward activation values.
- This is useful because it allows us to compute sigmoid(x once) and re-use its value later on to get the derivative.

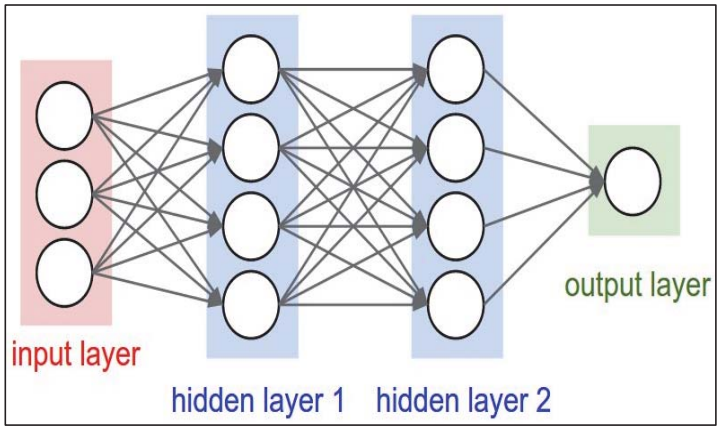
ForwardProp Let's denote the sigmoid function as $\sigma(x) = \frac{1}{1+e^{-x}}$.

BackwardProp The derivative of the sigmoid is $\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$.

Mini-batch SGD

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient



~1700s	• The prologue
1940-1970	• Act 1 Tinker: Hack it up
1970-1995	• Act 2 BackProp: theory to the rescue
1995-2007	• Act 3 Layer by layer learning, a medieval pastime
2007-2015	• Act 4 Introspection: better init. and activation functions
2015 -	• Act 5 Express-laning the gradient: Skip Connections, the SoTA frontier (LSTMs, ResNet, Highway Nets, DenseNets)
	• The epilogue

Deep Challenges: Wrestling with vanishing gradients

- Wrestling with vanishing gradients intermediate progress (sigmoid): train locally

- STEP 1:Train each layer locally (in unsupervised manner; encode decode)
- STEP 2: then bolt all layers together and then do backprop (avoid init problems and disappearing gradients, a big problem with sigmoid act. functions)
- RBM (Restricted Boltzman machine)

Autoencoder: topics covered here

- a simple autoencoder based on a fully-connected layer
- a sparse autoencoder
- a deep fully-connected autoencoder
- a deep convolutional autoencoder
- an image denoising model
- a sequence-to-sequence autoencoder
- a variational autoencoder

Topics in this box are Covered here

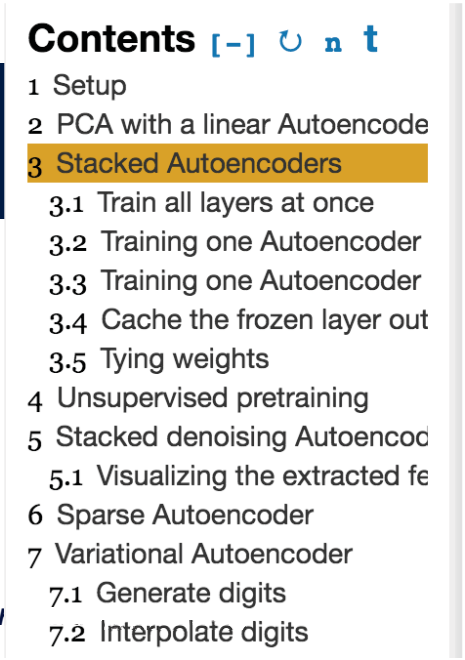
From RBM to Autoencoders to deep learning

- **One of the triggers of the current Deep Learning tsunami is the discovery in 2006 by Geoffrey Hinton et al. that deep neural networks can be pretrained in an unsupervised fashion.**
  - They used restricted Boltzmann machines for this
  - Restricted Boltzmann Machines were introduced by Hinton et al in 1985
- **Then in 2007 Yoshua Bengio et al. showed that autoencoders. worked just as well.**
  - Greedy Layer-Wise Training of Deep Networks
    - http://machinelearning.wustl.edu/mlpapers/paper_files/NIPS2006_739.pdf

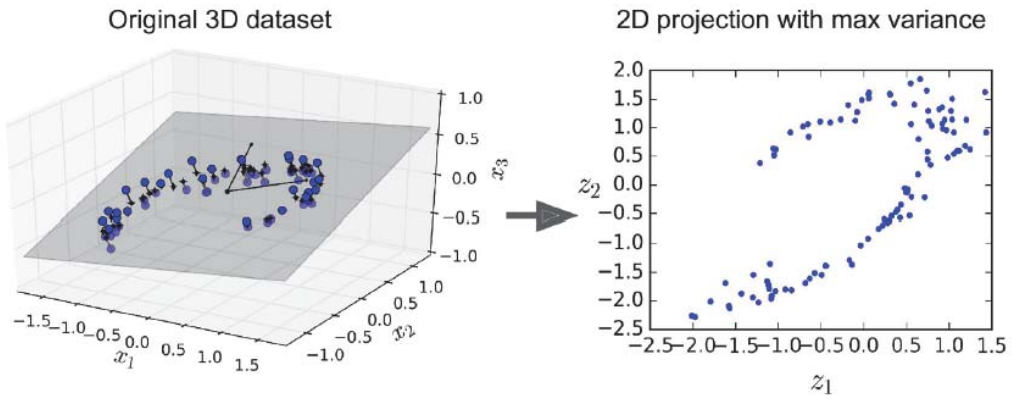
Autoencoder: Reading material and notebooks

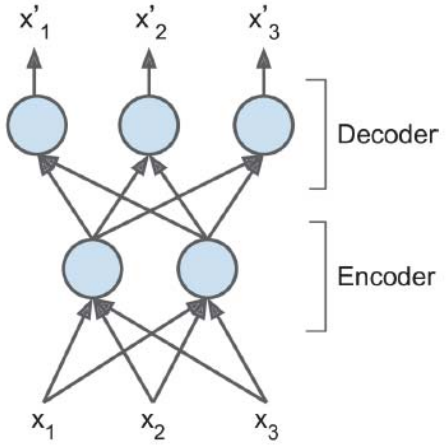
- https://en.wikipedia.org/wiki/Autoencoder
- **Code and examples**
- https://blog.keras.io/building-autoencoders-in-keras.html

http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/7nklebq00l3q7q1/15_autoencoders.ipynb



Linear Autoencoder (PCA)





PCA performed by an undercomplete linear autoencoder

Autoencoder

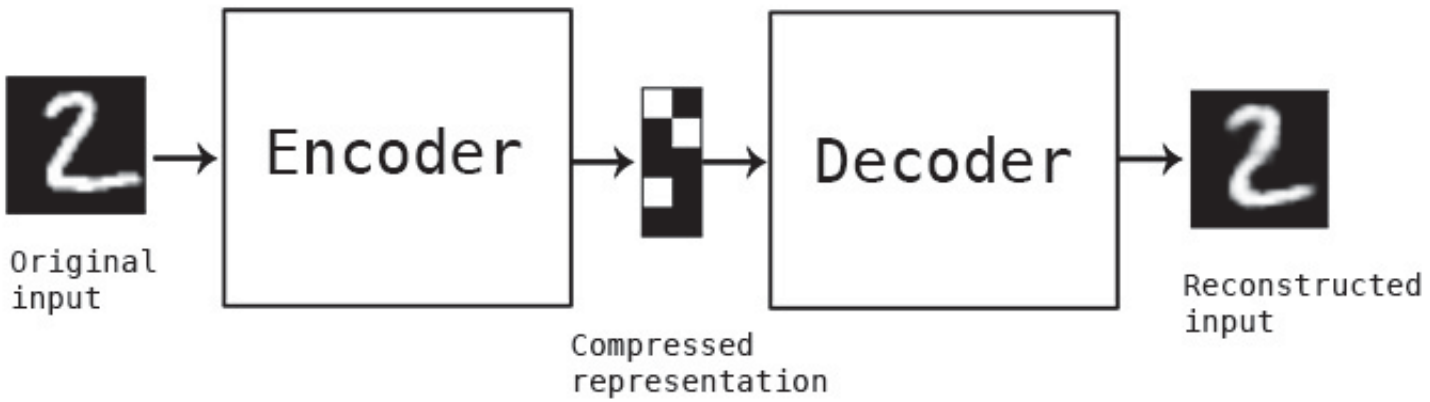
- Autoencoders are artificial neural networks capable of learning efficient representations of the input data, called codings, without any supervision (i.e., the training set is unlabeled).
- These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction.
- More importantly, autoencoders act as powerful feature detectors, and they can be used for unsupervised pretraining of deep neural networks

- https://en.wikipedia.org/wiki/Autoencoder

Autoencoders as data generators

- Lastly, they are capable of randomly generating new data that looks very similar to the training data; this is called a generative model.
- For example, you could train anautoencoder on pictures of faces, and it would then be able to generate new faces.

- **Autoencoders work by simply learning to copy their inputs to their outputs.**



Notebook for a simple autoencoder 100-50-10-50-100

Data

```
X, y = make_blobs(n_samples=2000, n_features=100, centers=2, random_state=42)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, test_size=0.2)
```

Autoencoder

```
inp = Input(shape=(X.shape[1],))
enc = Dense(units=50, activation="relu")(inp)
hid = Dense(units=10, activation="relu")(enc)
dec = Dense(units=50, activation="relu")(hid)
out = Dense(units=X.shape[1], activation="linear")(dec)
```

```
autoencoder = Model(inputs=[inp], outputs=[out])
autoencoder.compile(loss="mse", optimizer="adam")
autoencoder.summary()
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100)	0
dense_1 (Dense)	(None, 50)	5050
dense_2 (Dense)	(None, 10)	510
dense_3 (Dense)	(None, 50)	550
dense_4 (Dense)	(None, 100)	5100

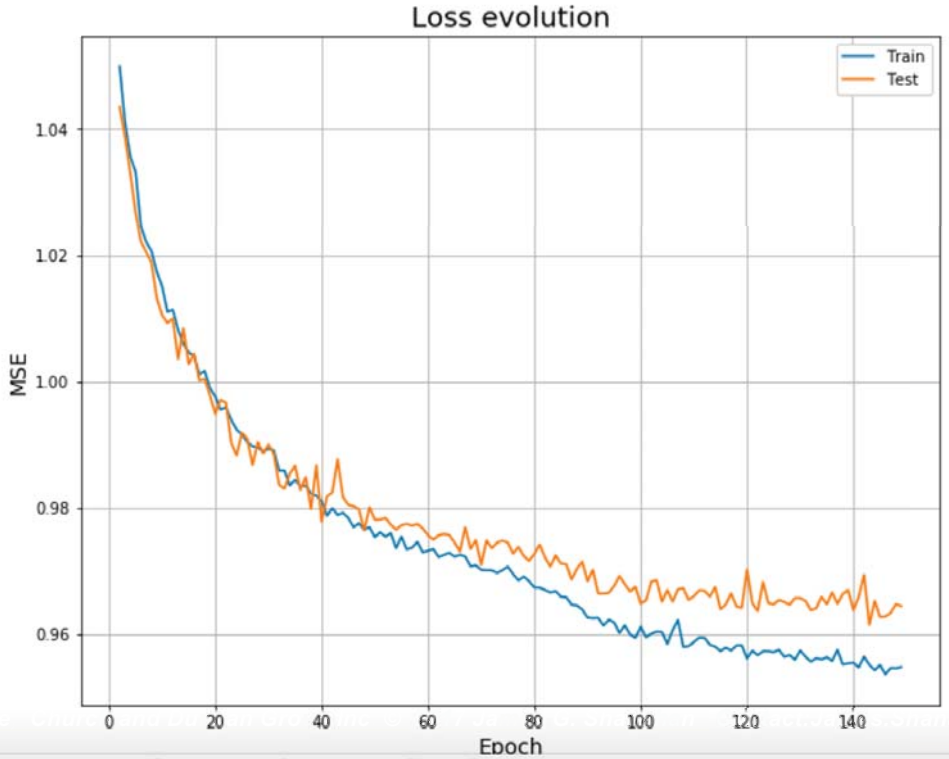
Total params: 11,210
Trainable params: 11,210
Non-trainable params: 0

http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/zbufi92yzlfv9at/AutoencoderBasic.ipynb

```
history = autoencoder.fit(X_train, X_train, epochs=150, validation_data=(X_test, X_test), verbose=0)
```

```
/opt/conda/lib/python3.6/site-packages/keras/backend/tensorflow_backend.py:2252: UserWarning: Expected no kwargs,
kwargs passed to function are ignored with Tensorflow backend
  warnings.warn('\n'.join(msg))
```

```
plt.figure(figsize=(10, 8))
plt.plot(np.arange(2, 150), history.history["loss"][2:], label="Train")
plt.plot(np.arange(2, 150), history.history["val_loss"][2:], label="Test")
plt.legend(frameon=True, framealpha=1)
plt.grid("on")
plt.xlabel("Epoch", fontsize=14)
plt.ylabel("MSE", fontsize=14)
plt.title("Loss evolution", fontsize=18);
```



Encoder and decoder in terms of layer variables in Keras

Encoder part

```
encoder = Model(inputs=[inp], outputs=[hid])
```

```
encoded_test = encoder.predict(X_test)
```

```
/opt/conda/lib/python3.6/site-packages/keras/backend/tensorflow_backend.py:2252: UserWarning: Expected no kwargs, you passed 1
kwargs passed to function are ignored with Tensorflow backend
  warnings.warn('\n'.join(msg))
```

```
encoded_test.shape
```

```
(400, 10)
```

encoder = Model(inputs=[inp], outputs=[hid])
encoded_test = encoder.predict(X_test)

```
inp = Input(shape=(X.shape[1],))
enc = Dense(units=50, activation="relu")(inp)
hid = Dense(units=10, activation="relu")(enc)
dec = Dense(units=50, activation="relu")(hid)
out = Dense(units=X.shape[1], activation="linear")(dec)
```

```
autoencoder = Model(inputs=[inp], outputs=[out])
autoencoder.compile(loss="mse", optimizer="adam")
autoencoder.summary()
```

Decoder part

```
encoded_inp = Input(shape=(10,))
decoder_out = autoencoder.layers[-1](autoencoder.layers[-2](encoded_inp))
```

```
decoder = Model(inputs=[encoded_inp], outputs=[decoder_out])
```

```
decoder_test = decoder.predict(encoded_test)
```

```
/opt/conda/lib/python3.6/site-packages/keras/backend/tensorflow_backend.py:2252: UserWarning: Expected no kwargs, you passed 1
kwargs passed to function are ignored with Tensorflow backend
  warnings.warn('\n'.join(msg))
```

```
decoder_test.shape
```

```
(400, 100)
```

Decoder: neat one liner

```
inp = Input(shape=(X.shape[1],))
enc = Dense(units=50, activation="relu")(inp)
hid = Dense(units=10, activation="relu")(enc)
dec = Dense(units=50, activation="relu")(hid)
out = Dense(units=X.shape[1], activation="linear")(dec)
```

```
autoencoder = Model(inputs=[inp], outputs=[out])
autoencoder.compile(loss="mse", optimizer="adam")
autoencoder.summary()
```

Decoder part

decoder_out = autoencoder.layers[-1](autoencoder.layers[-2](encoded_inp))

:
```
encoded_inp = Input(shape=(10,))
decoder_out = autoencoder.layers[-1](autoencoder.layers[-2](encoded_inp))
```

:
```
decoder = Model(inputs=[encoded_inp], outputs=[decoder_out])
```

:
```
decoder_test = decoder.predict(encoded_test)
```

Autoencoder

Architecturally, the simplest form of an autoencoder is a feedforward, non-recurrent neural network very similar to the multilayer perceptron (MLP) – having an input layer, an output layer and one or more hidden layers connecting them –, but with the output layer having the same number of nodes as the input layer, and with the purpose of *reconstructing* its own inputs (instead of predicting the target value Y given inputs X). Therefore, autoencoders are unsupervised learning models.

An autoencoder always consists of two parts, the encoder and the decoder, which can be defined as transitions ϕ and ψ, such that:

$$\phi : \mathcal{X} \to \mathcal{F}$$
$$\psi : \mathcal{F} \to \mathcal{X}$$
$$\phi, \psi = \arg\min_{\phi,\psi} \|X - (\psi \circ \phi)X\|^2$$

In the simplest case, where there is one hidden layer, the encoder stage of an autoencoder takes the input $\mathbf{x} \in \mathbb{R}^d = \mathcal{X}$ and maps it to $\mathbf{z} \in \mathbb{R}^p = \mathcal{F}$:

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

This image $\mathbf{z}$ is usually referred to as *code*, *latent variables*, or *latent representation*. Here, σ is an element-wise activation function such as a sigmoid function or a rectified linear unit. $\mathbf{W}$ is a weight matrix and $\mathbf{b}$ is a bias vector. After that, the decoder stage of the autoencoder maps $\mathbf{z}$ to the *reconstruction* $\mathbf{x}'$ of the same shape as $\mathbf{x}$:

$$\mathbf{x}' = \sigma'(\mathbf{W}'\mathbf{z} + \mathbf{b}')$$

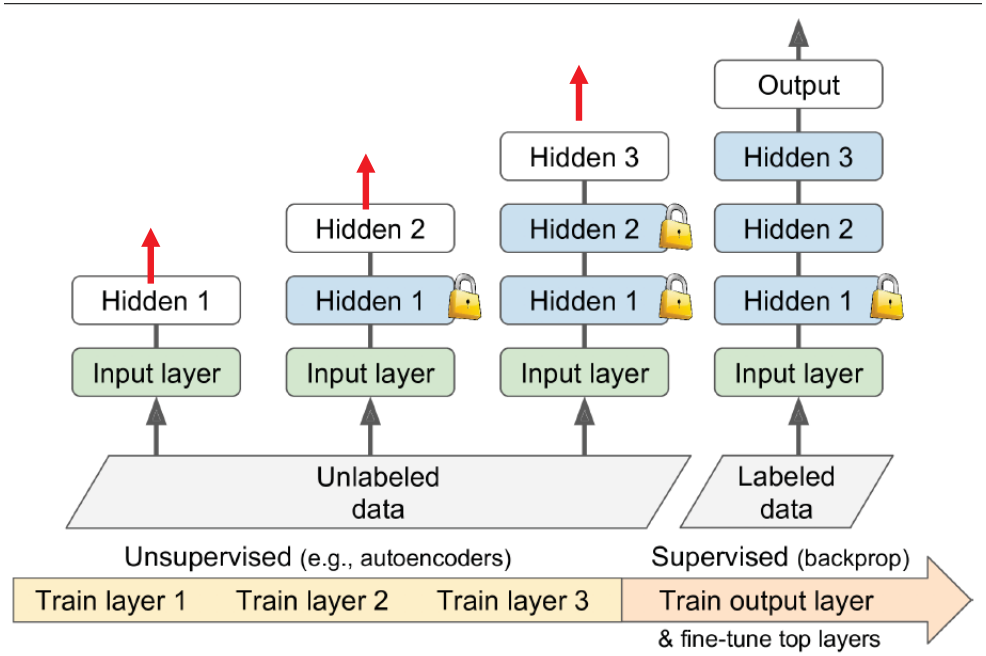
where σ', $\mathbf{W}'$, and $\mathbf{b}'$ for the decoder may differ in general from the corresponding σ, $\mathbf{W}$, and $\mathbf{b}$ for the encoder, depending on the design of the autoencoder.

Autoencoders are also trained to minimise reconstruction errors (such as squared errors):

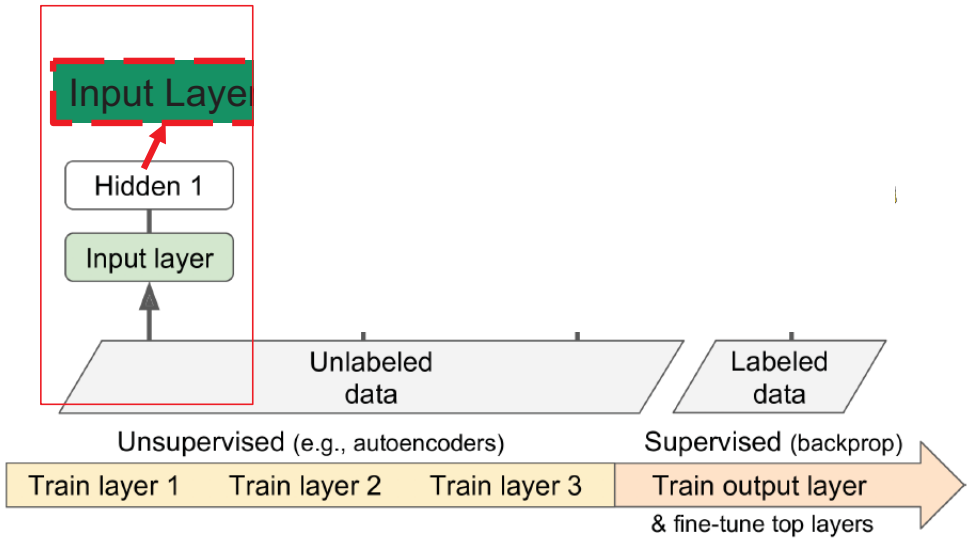
$$\mathcal{L}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2 = \|\mathbf{x} - \sigma'(\mathbf{W}'(\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})) + \mathbf{b}')\|^2$$

where $\mathbf{x}$ is usually averaged over some input training set.

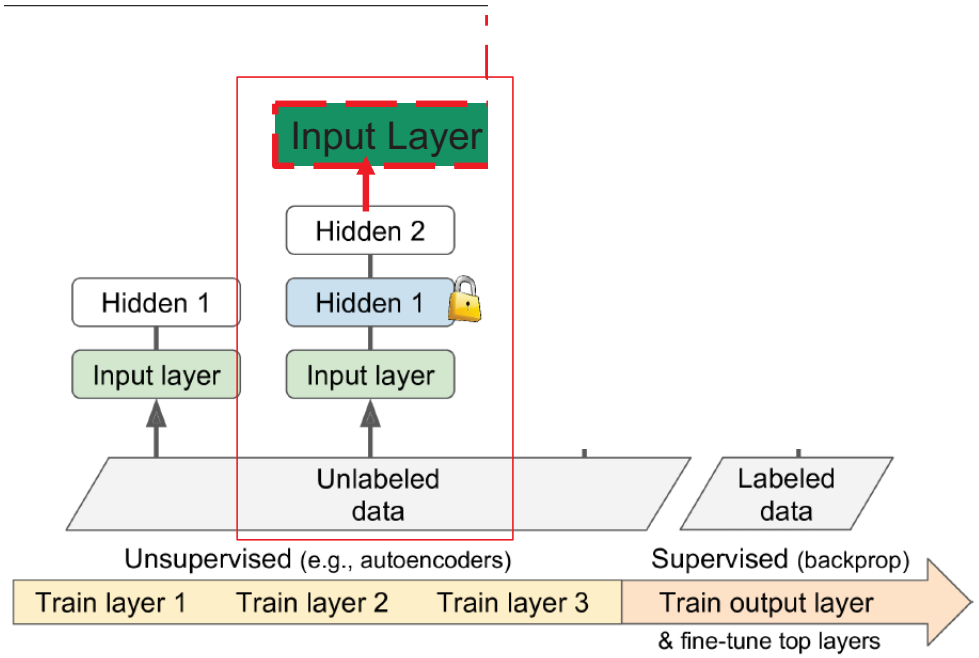
Schematic str
connected hid


Output
Hidden 3
Hidden 3
Hidden 2
Hidden 2
Hidden 2
Hidden 1
Hidden 1
Hidden 1
Hidden 1
Input layer
Input layer
Input layer
Input layer
Unlabeled data
Labeled data
Unsupervised (e.g., autoencoders)
Supervised (backprop)
Train layer 1
Train layer 2
Train layer 3
Train output layer
& fine-tune top layers

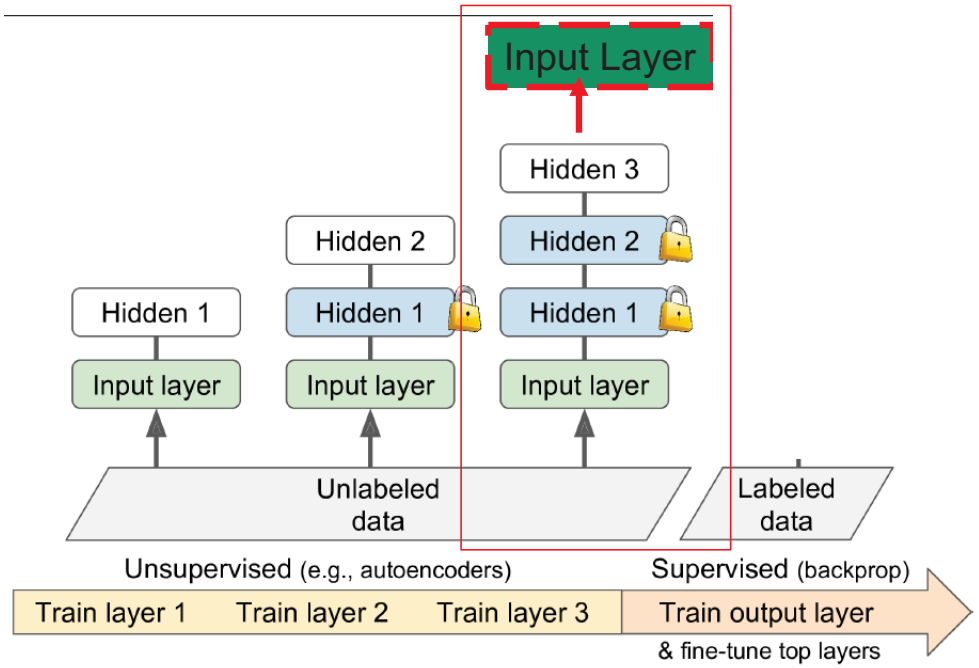
Train hidden layer 1



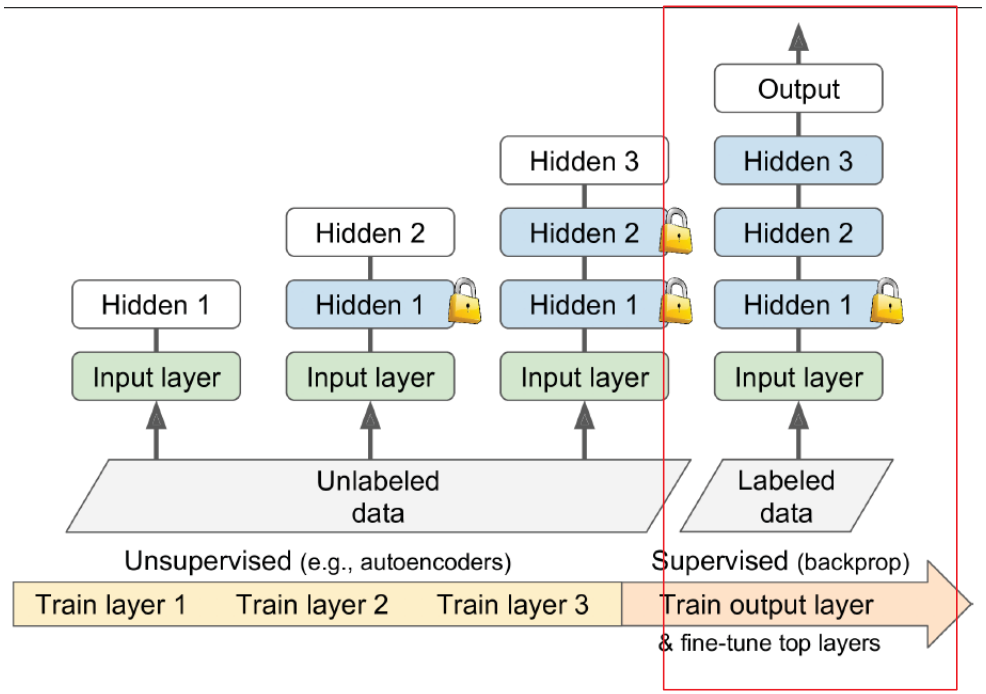
Train hidden layer 2



Train hidden layer 3



Fine tune top layers using backprop



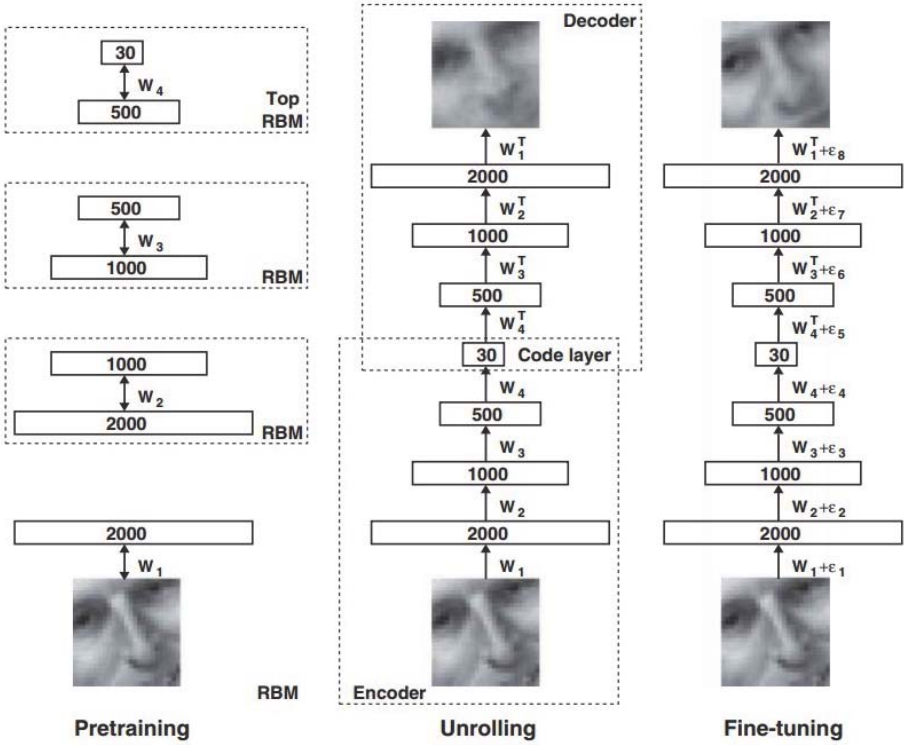
Back propagation works but

- One needs to be careful in the way one uses it
- E.g., Sigmoid activation functions with BP don't work well

Layer by layer learning

[Hinton and Salakhutdinov 2006]

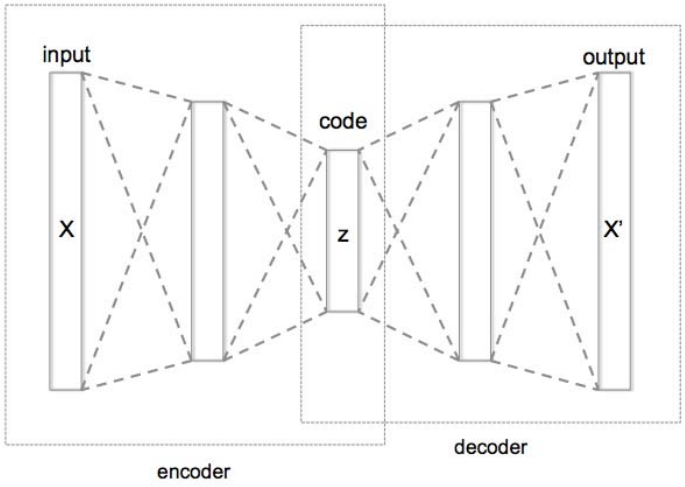
Reinvigorated research in Deep Learning



13
5

Autoencoders

- Architecturally, the simplest form of an autoencoder is a feedforward, non-recurrent neural network very similar to the multilayer perceptron (MLP) – having an input layer, an output layer and one or more hidden layers connecting them –, but with the output layer having the same number of nodes as the input layer, and with the purpose of *reconstructing* its own inputs



V • T • E

Architecturally, the simplest form of an autoencoder is a feedforward, non-recurrent neural network very similar to the multilayer perceptron (MLP) – having an input layer, an output layer and one or more hidden layers connecting them –, but with the output layer having the same number of nodes as the input layer, and with the purpose of *reconstructing* its own inputs (instead of predicting the target value Y given inputs X). Therefore, autoencoders are unsupervised learning models.

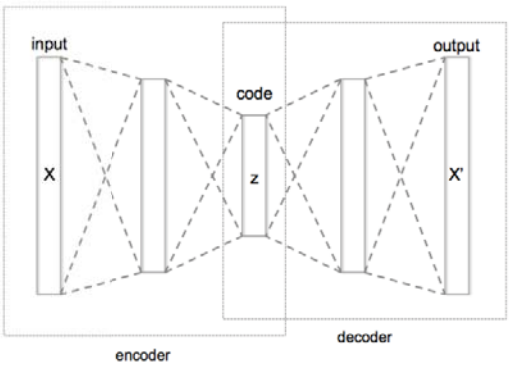
An autoencoder always consists of two parts, the encoder and the decoder, which can be defined as transitions ϕ and ψ, such that:

$$\phi : \mathcal{X} \to \mathcal{F}$$
$$\psi : \mathcal{F} \to \mathcal{X}$$
$$\phi, \psi = \arg\min_{\phi,\psi} \|X - (\psi \circ \phi)X\|^2$$

In the simplest case, where there is one hidden layer, an autoencoder takes the input $\mathbf{x} \in \mathbb{R}^d = \mathcal{X}$ and maps it onto $\mathbf{z} \in \mathbb{R}^p = \mathcal{F}$:

$$\mathbf{z} = \sigma_1(\mathbf{W}\mathbf{x} + \mathbf{b})$$

This is usually referred to as *code* or *latent variables* (latent representation). Here,



Schematic structure of an autoencoder with 3 fully-connected hidden layers.

Autoencoders in TensorFlow

Contents [-]

1 Setup
2 PCA with a linear Autoencode
3 Stacked Autoencoders
3.1 Train all layers at once
3.2 Training one Autoencoder
3.3 Training one Autoencoder
3.4 Cache the frozen layer out
3.5 Tying weights
4 Unsupervised pretraining
5 Stacked denoising Autoencod
5.1 Visualizing the extracted fe
6 Sparse Autoencoder
7 Variational Autoencoder
7.1 Generate digits
7.2 Interpolate digits

Break-Through (2006, and 2007)

Another recent Break-Through (2014)

Unsupervised pretraining

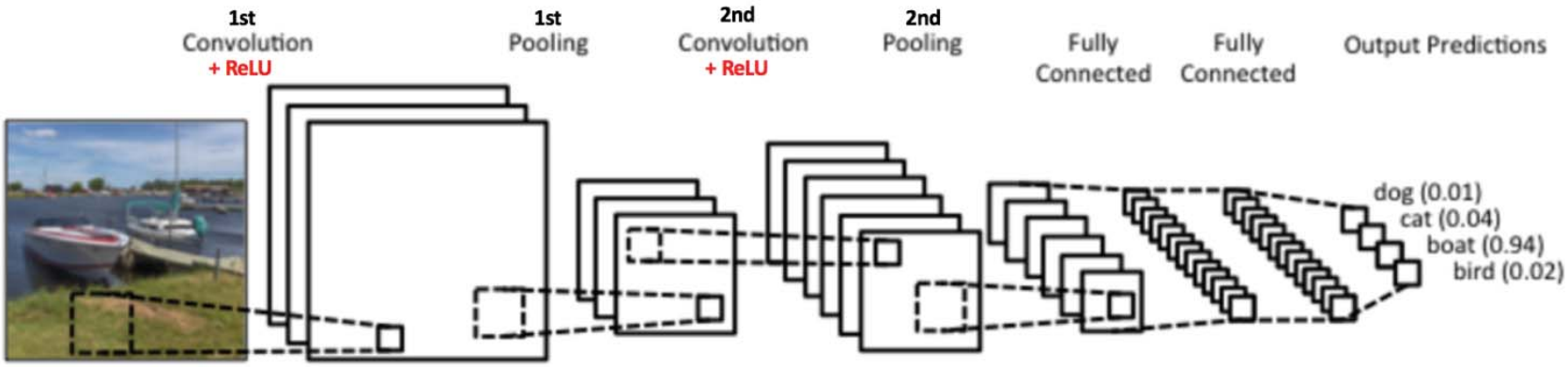
```
tf.reset_default_graph()

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 150
n_outputs = 10

learning_rate = 0.01
l2_reg = 0.0005

activation = tf.nn.elu
regularizer = tf.contrib.layers.l2_regular
initializer = tf.contrib.layers.variance_s

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
y = tf.placeholder(tf.int32, shape=[None])

weights1_init = initializer([n_inputs, n_hidden1])
weights2_init = initializer([n_hidden1, n_hidden2])
weights3_init = initializer([n_hidden2, n_hidden3])

weights1 = tf.Variable(weights1_init, dtype=tf.float32, name="weights1")
weights2 = tf.Variable(weights2_init, dtype=tf.float32, name="weights2")
weights3 = tf.Variable(weights3_init, dtype=tf.float32, name="weights3")

biases1 = tf.Variable(tf.zeros(n_hidden1), name="biases1")
biases2 = tf.Variable(tf.zeros(n_hidden2), name="biases2")
biases3 = tf.Variable(tf.zeros(n_hidden3), name="biases3")

hidden1 = activation(tf.matmul(X, weights1) + biases1)
hidden2 = activation(tf.matmul(hidden1, weights2) + biases2)
logits = tf.matmul(hidden2, weights3) + biases3

cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=log
reg_loss = regularizer(weights1) + regularizer(weights2) + regularizer(weights3)
loss = cross_entropy + reg_loss
optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
pretrain_saver = tf.train.Saver([weights1, weights2, biases1, biases2])
saver = tf.train.Saver()
```

http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/7nklebq00l3q7q1/15_autoencoders.ipynb

1 Setup
2 PCA with a linear Autoencode
3 Stacked Autoencoders
3.1 Train all layers at once
3.2 Training one Autoencoder
3.3 Training one Autoencoder
3.4 Cache the frozen layer out
3.5 Tying weights
4 Unsupervised pretraining
5 Stacked denoising Autoencod
5.1 Visualizing the extracted fe
6 Sparse Autoencoder
7 Variational Autoencoder
7.1 Generate digits
7.2 Interpolate digits

CNN + Autoencoder-based training in Vision

- **Learn feature detectors which are shared among all locations in an image, because features which capture useful information in one part of an image can pick up the same information elsewhere**



Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations
Honglak Lee, Roger Grosse, Rajesh Ranganath, Andrew Y. Ng, ICML 2009
http://web.eecs.umich.edu/~honglak/icml09-ConvolutionalDeepBeliefNetworks.pdf

~1700s	• The prologue
1940-1970	• Act 1 Tinker: Hack it up
1970-1995	• Act 2 BackProp: theory to the rescue
1995-2007	• Act 3 Layer by layer learning, a medieval pastime
2007-2015	• Act 4 Introspection: better init. and activation functions
2015 -	• Act 5 Express-laning the gradient: Skip Connections, the SoTA frontier (LSTMs, ResNet, Highway Nets, DenseNets)
	• The epilogue

Activations should be unit Gaussian everywhere!

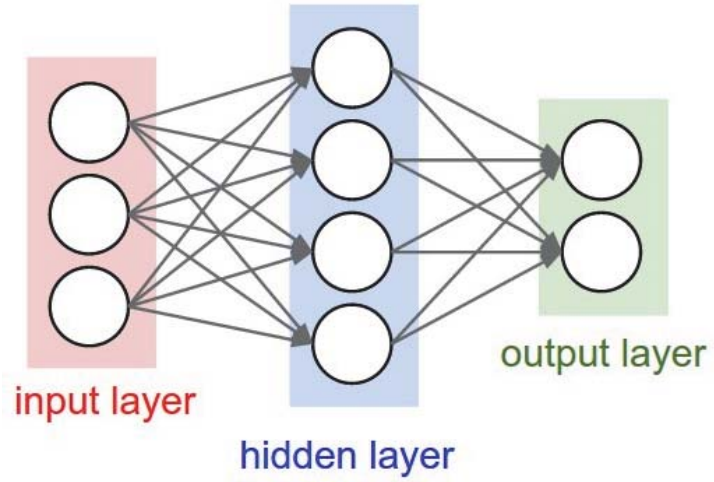
“you want unit Gaussian activations in all parts of your network? just make them so.”

Who comes up with this stuff?

EASY to say

Please do NOT try this at home!

- Q: what happens when W=0 init is used?



All neurons will compute the same outputs

if initial wgts too small activations go to ZERO. Let's explore this and overcome this using two very different approaches

- **Use "Xavier initialization"**
- **Batch Normalization**

Init with small random numbers

- **CASE 1: Small random numbers: init W_i**
  - (gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

  - Activations go to ZERO, weights are close to zero so gradient goes to zero
- **CASE 2: bigger small random numbers**

```
W =      np.random.randn(D,H)
```

  - Network get saturated as the activations get pegged at 1 or -1 Tanh so gradient goes to zero ($d(\tanh)/dx = 1 - \tanh(x)^2 = 0$
- **CASE 3: between 0.1 and 1…but where?**

Init with small random numbers

- First idea: **Small random numbers**

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.

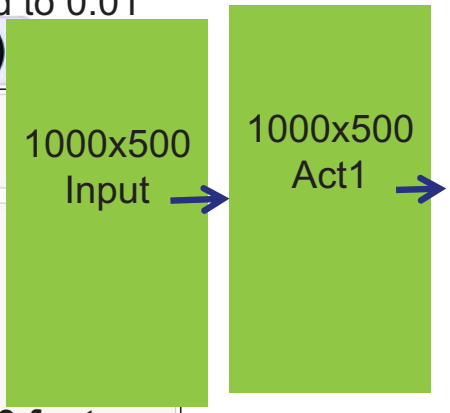
10-layer NN, 500 units each with tanh

1000-500-500-500-……..

Lets look at some activation statistics

Init each W_i with Unit gaussian and scaled to 0.01

```
W = 0.01* np.random.randn(D,H)
```




```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)

act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

1,000 data points with 500 features

Zero mean, std= 1

10 layers of 500 hidden nodes

```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

Take mean(Act_i)

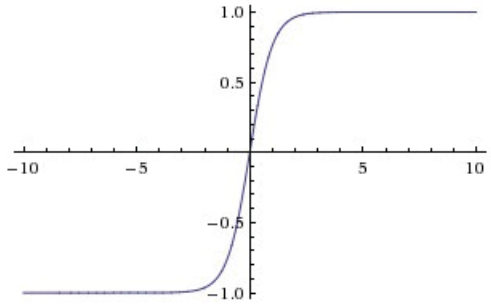
Zero mean, std = 0.01

Compare two act functions: ReLU or Tanh

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

tanh

Activation Functions



tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

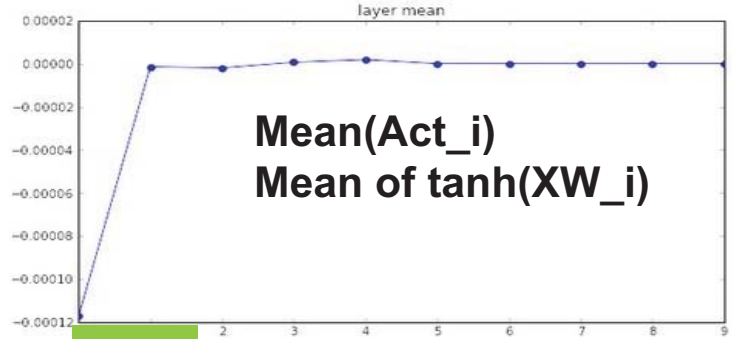
[LeCun et al., 1991]

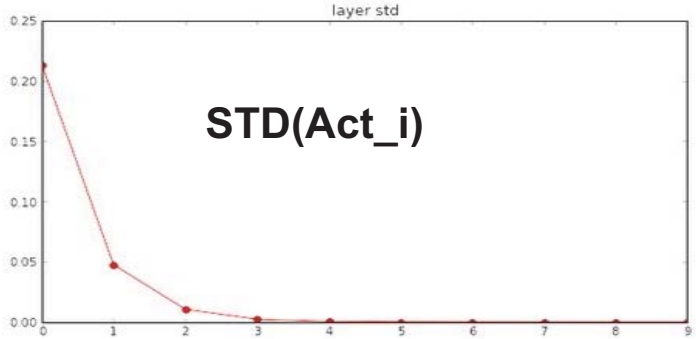
Init = N(mean=0, STD=0.01)
Tanh-based Activations : Mean is zero along with STD

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```

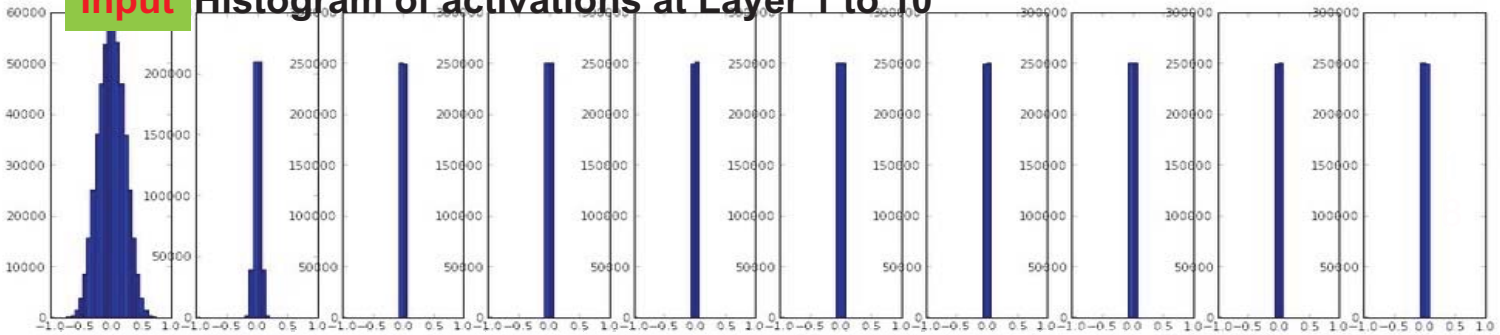
Activations for 500 neurons on each layer for the 1,000 training data

Using TANH so Mean remains ~0.0, but STD collapses → 0





Input Histogram of activations at Layer 1 to 10

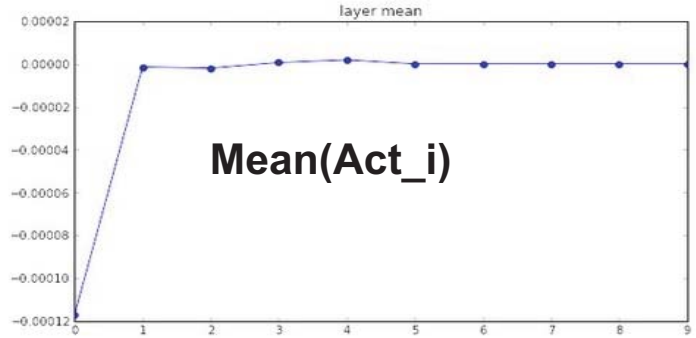


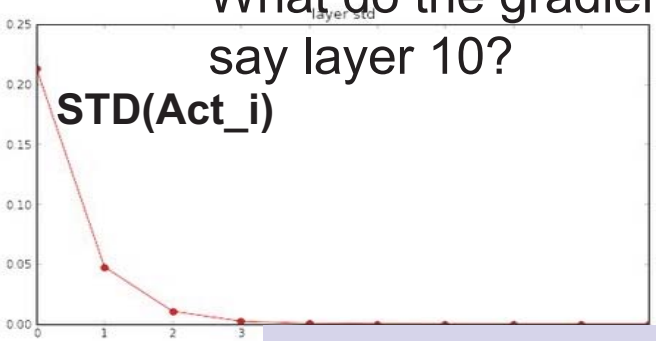
All activations become zero
Gradients disappear also? Why?

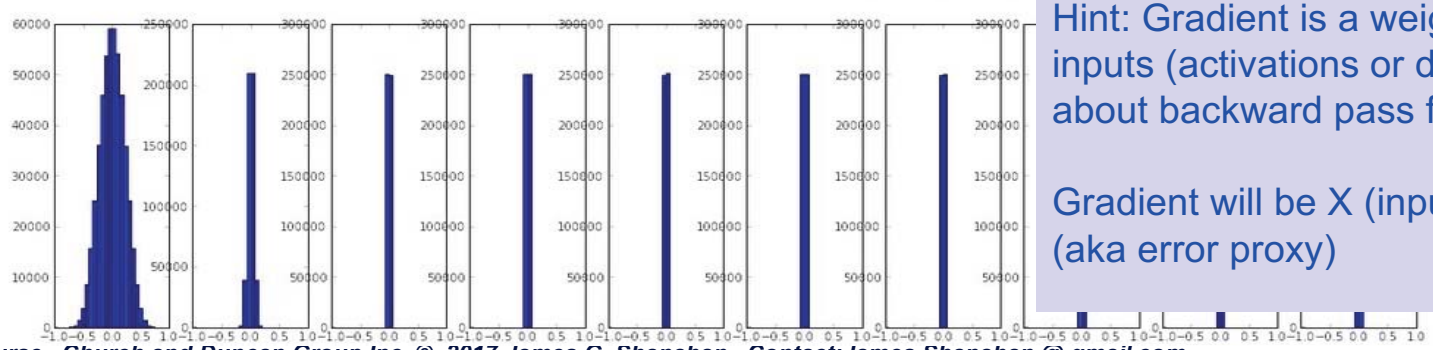
```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```

All activations become zero!

Q: think about the backward pass. What do the gradients look like in say layer 10?







Hint: Gradient is a weighted sum of its inputs (activations or data inputs) think about backward pass for a W*X gate.

Gradient will be X (input) times error (aka error proxy)

▽ the gradient chorus

- **What is the gradient for linear regression?**

- **Chorus**

Linear Regression – The gradient is the weighted sum of the training data, where the weights are proportional to the error (for each example) !

NN – The gradient is the weighted sum of the **input** data, where the weights are proportional to the error **or the error proxy** (for each example) !

weight *example*

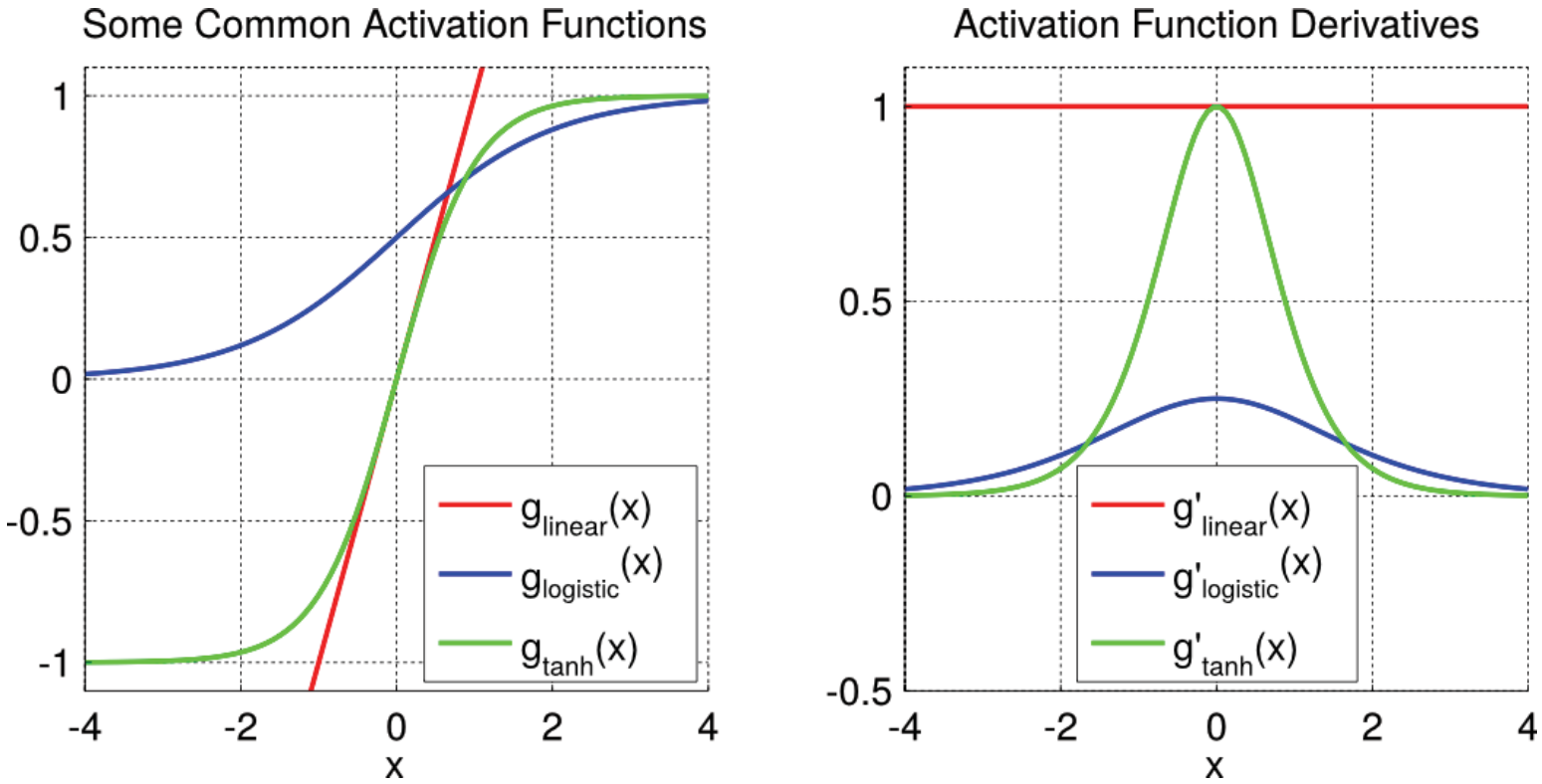
$$\frac{\partial E}{\partial W} = (O - t)\,\boldsymbol{X}$$

$$W = W - \alpha \times \frac{\partial E}{\partial W}$$

$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} \mathbf{34} \\ \mathbf{1} \\ \mathbf{4} \end{bmatrix}$$

$$4.32 = 5 - 0.01 \times (2 \times 34) \text{ for } i = 1$$

Derivatives of Activation Functions



https://theclevermachine.wordpress.com/2014/09/08/derivation-derivatives-for-common-neural-network-activation-functions/

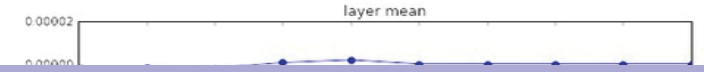
All activations become zero
Gradients disappear also? Why?

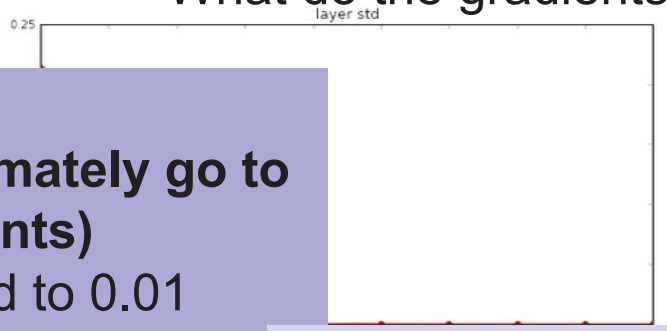
```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```

Using Tanh

All activations become zero!

Q: think about the backward pass. What do the gradients look like?

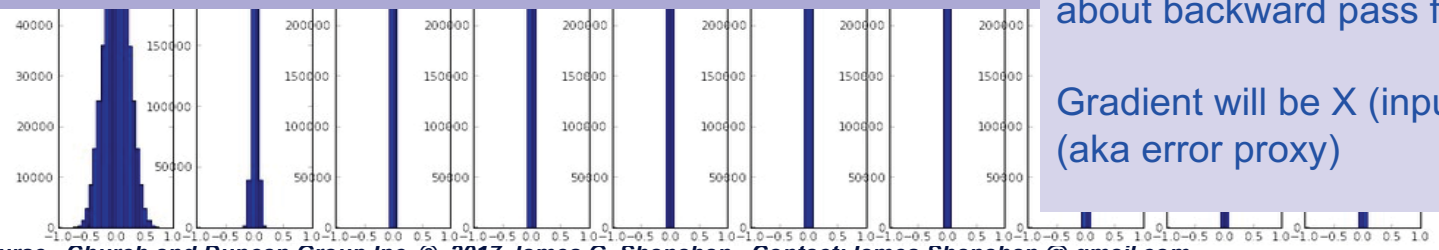




By the time you get back to input layer Gradient goes to ZEROs or Ones but ultimately go to ZERO (tiny gradients or vanishing gradients)

Init each W_i with Unit Gaussian and scaled to 0.01

W is used both ways so if W is small Gradients will be small



Hint: Gradient is a weighted sum of its inputs (activations or data inputs) think about backward pass for a W*X gate.

Gradient will be X (input) times error (aka error proxy)

Gradient updates

- Propagate signal from output layer backwards
- But Activations and Weights…. So gradient goes to zero very quickly
- Sigmoids have other challenges

Init with small random numbers

- **CASE 1: Small random numbers: init W_i**
  - (gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

- **CASE 2: bigger small random numbers**

```
W =      np.random.randn(D,H)
```

- **CASE 3: between 0.1 and 1…but where?**

Try init W_i with N(0, 1): Activations become bimodal activations: +1; -1

```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

*1.0 instead of *0.01

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000[illegible] and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.[illegible] and std 0.981[illegible]
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

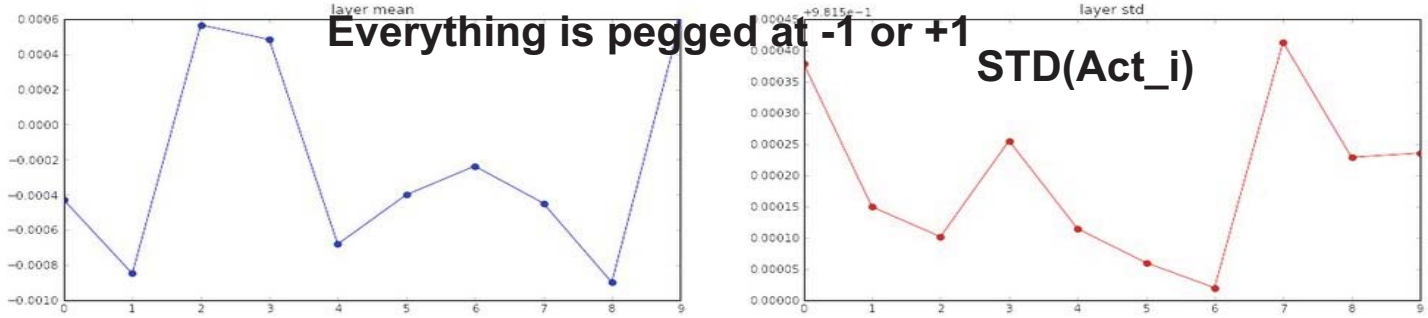
Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero. WHY?

Mean(Act_i)

Mean of tanh(XW_i)

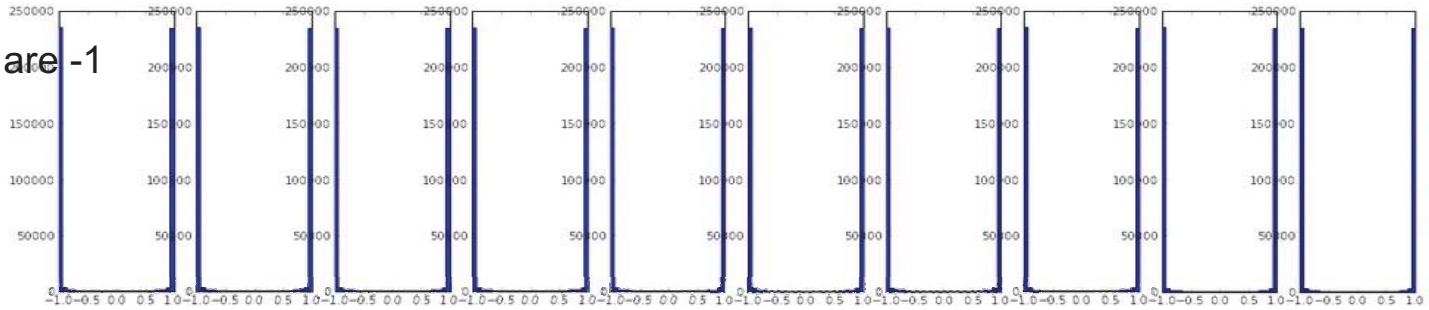
But look at the distributions below.

Everything is pegged at -1 or +1



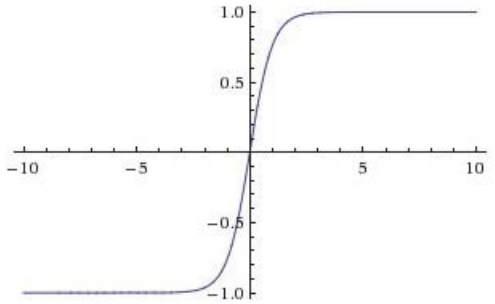
STD(Act_i)

Tanh are -1



Tanh gets saturated as the weights are too large Leading to large absolute weighted sums

Activation Functions



tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]
proposed tanh

Weights are large N(0, 1) so the scores going thru the activation function (tanh) are large to they saturate at 1

Gradient: $d(\tanh)/d(x) = 1 - \tanh(x)^2$

Try init W_i with N(0, 1): Activations become bimodal activations: +1; -1

```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

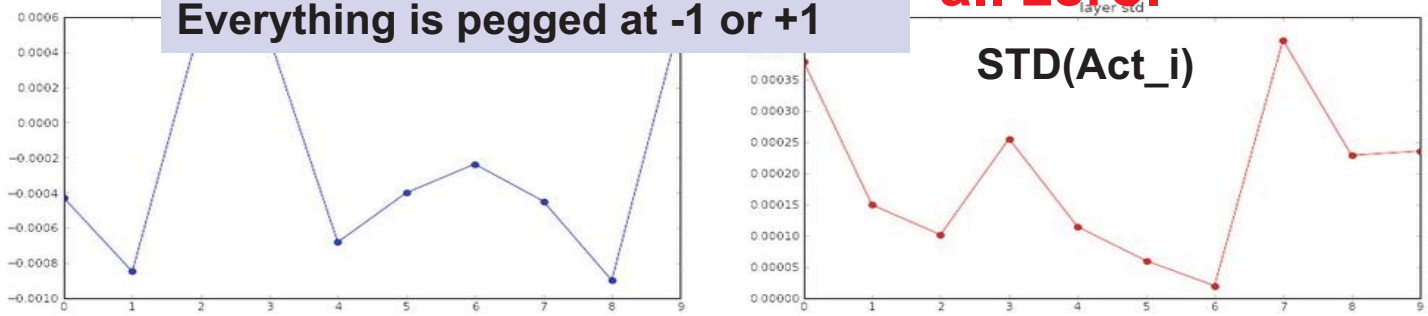
*1.0 instead of *0.01

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer
hidden layer
hidden layer
hidden layer
hidden layer
hidden layer
hidden layer
```

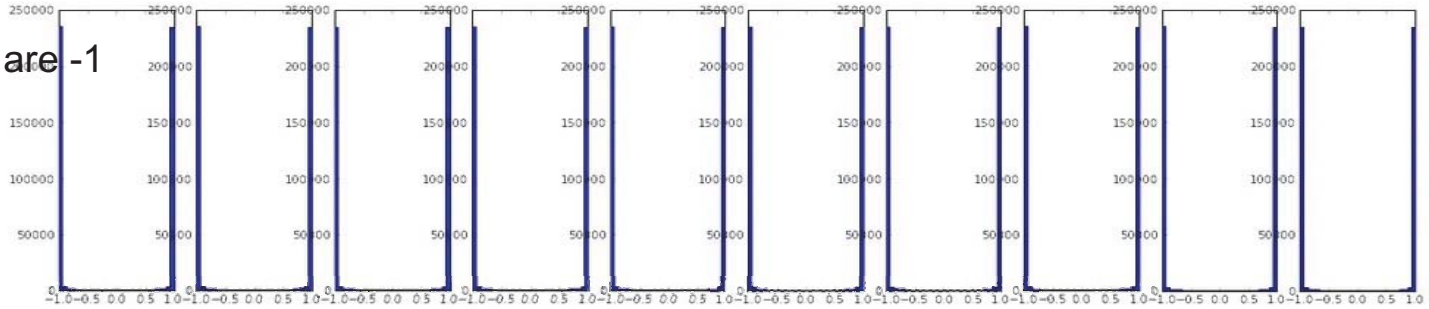
Mean(Act_i)
Mean of tanh(XW_i)
But look at the distributions below.
Everything is pegged at -1 or +1

Almost all neurons completely saturated, either -1 and 1.

Gradients will be all zero.



STD(Act_i)



Tanh are -1

Init with small random numbers

- **CASE 1: Small random numbers: init W_i**
  - (gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

  - Activations go to ZERO, weights are close to zero so gradient goes to zero
- **CASE 2: bigger small random numbers**

```
W =      np.random.randn(D,H)
```

  - Network get saturated as the activations get pegged at 1 or -1 Tanh so gradient goes to zero ($d(\tanh)/dx = 1 - \tanh(x)^2 = 0$
- **CASE 3: between 0.1 and 1…but where?**
  - Can we get more principled? **Calibrating the variances with 1/sqrt(n)….→ 1/sqrt(n)/2 for ReLUs**

"Xavier initialization" for Tanh

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

- **"Xavier initialization" for Tanh**
  - [Xavier Glorot et Bengio, 2010]
  - Variance for neurons
    - Inputs for each neurons should be N(0, 1)
  - One problem with the previous N(0,1) suggestion is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs.
- **"Xavier initialization": It turns out that we can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its *fan-in* (i.e. its number of inputs).**

For details see: http://cs231n.github.io/neural-networks-2/

Derivation of Xavier init

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

The sketch of the derivation is as follows: Consider the inner product $s = \sum_i^n w_i x_i$ between the weights w and input x, which gives the raw activation of a neuron before the non-linearity. We can examine the variance of s:

$$\text{Var}(s) = \text{Var}(\sum_i^n w_i x_i)$$

$$= \sum_i^n \text{Var}(w_i x_i)$$

$$= \sum_i^n [E(w_i)]^2 \text{Var}(x_i) + E[(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i)\text{Var}(w_i)$$

$$= \sum_i^n \text{Var}(x_i)\text{Var}(w_i)$$

$$= (n\text{Var}(w))\ \text{Var}(x)$$

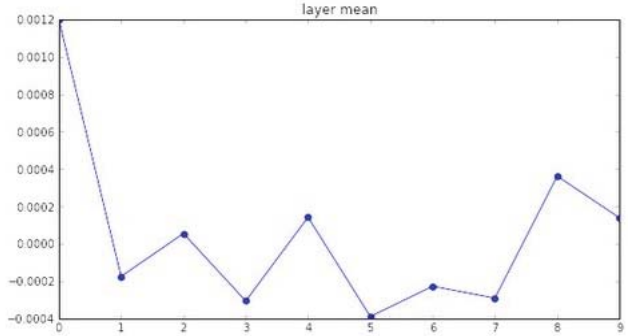
“Xavier initialization” for Tanh

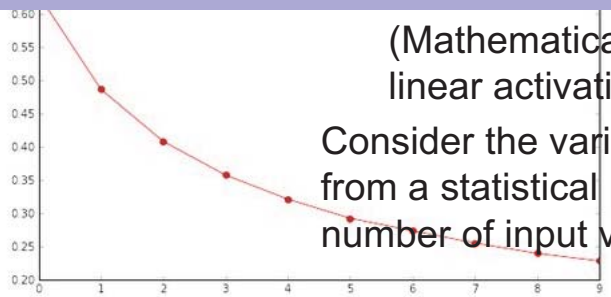
```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

itialization”
t al., 2010]

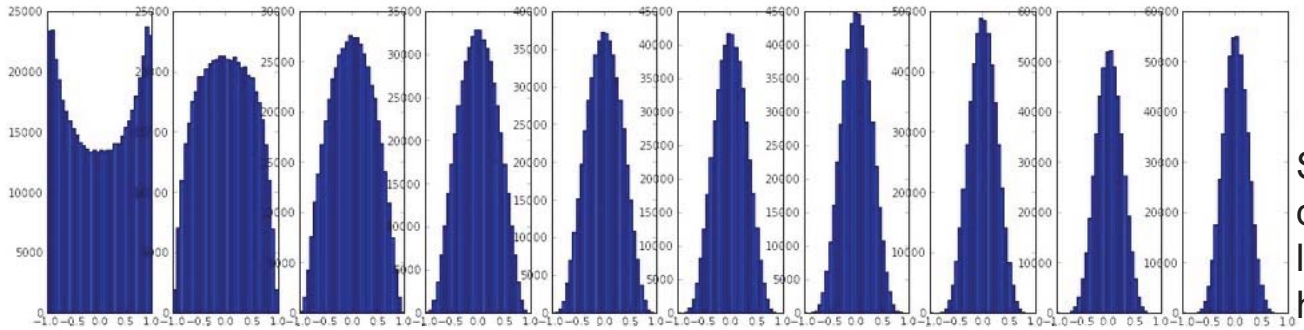
- **Num of weights: variance**
- **> sd(rnorm(500*500, sqrt(2/(500 + 500))))[1]**
- **1.001566**
- **mean(rnorm(500*500, sqrt(2/(500 + 500))))[1]**
- **0.04508544**





(Mathematical derivation assumes linear activations)

Consider the variance of the activations from a statistical perspective; consider the number of input variables



STD drops; lot better here

“Xavier initialization” for Tanh

```
input layer had mean 0.001800 and std 1.
hidden layer 1 had mean 0.001198 and std
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

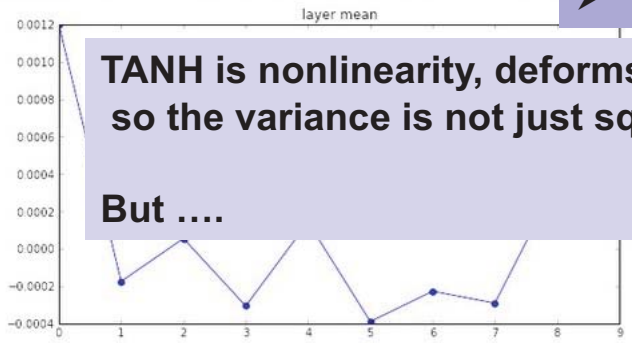
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

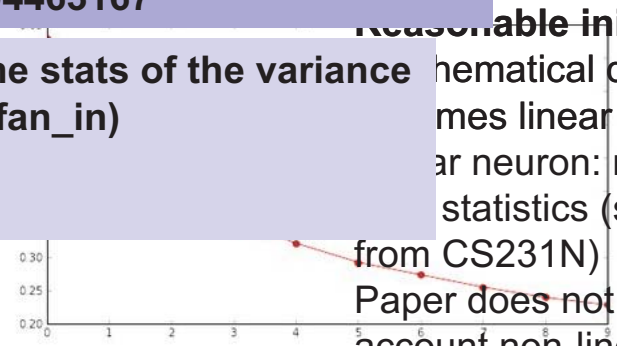
“Xavier initialization”
[Glorot et al., 2010]

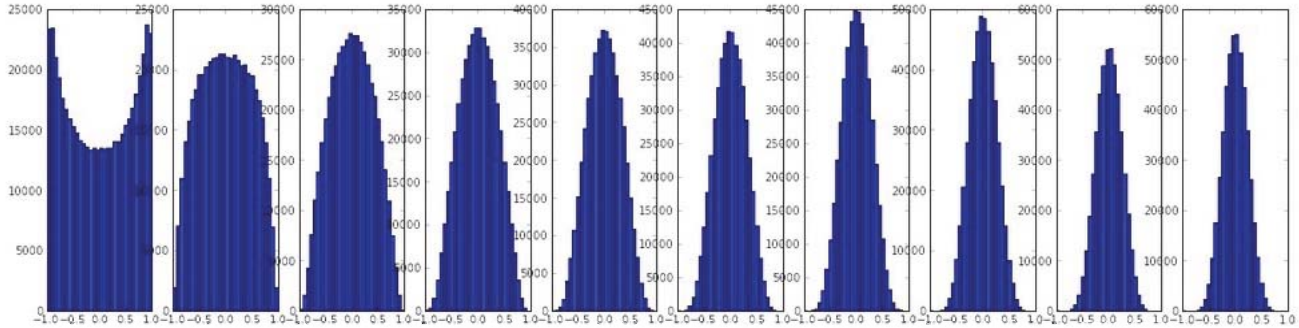
- **Num of weights: variance**
- **sd(rnorm(10000))/sqrt(500)[1]**
- **0.04463167**

TANH is nonlinearity, deforms the stats of the variance so the variance is not just sqrt(fan_in)

But ….







Reasonable initialization.
hematical derivation
mes linear activations)
ar neuron: motivated
statistics (see notes from CS231N)
Paper does not take into account non-linear neurons

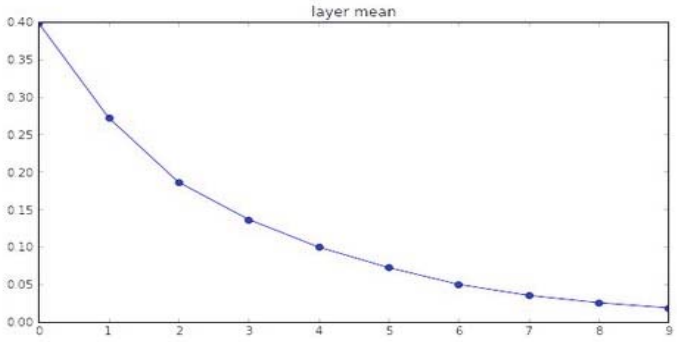
Within Active region of tanh

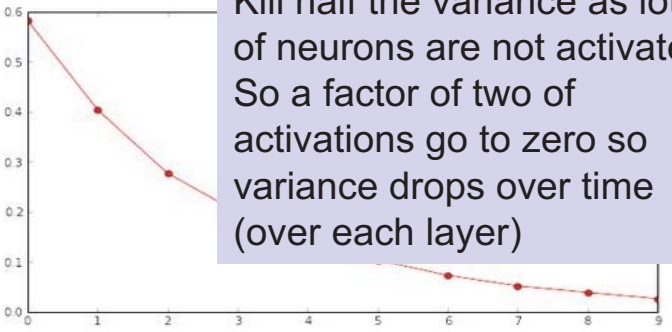
“Xavier initialization” for ReLU Neuron

```
input layer had mean 0.000501 and s
hidden layer 1 had mean 0.398623 an
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

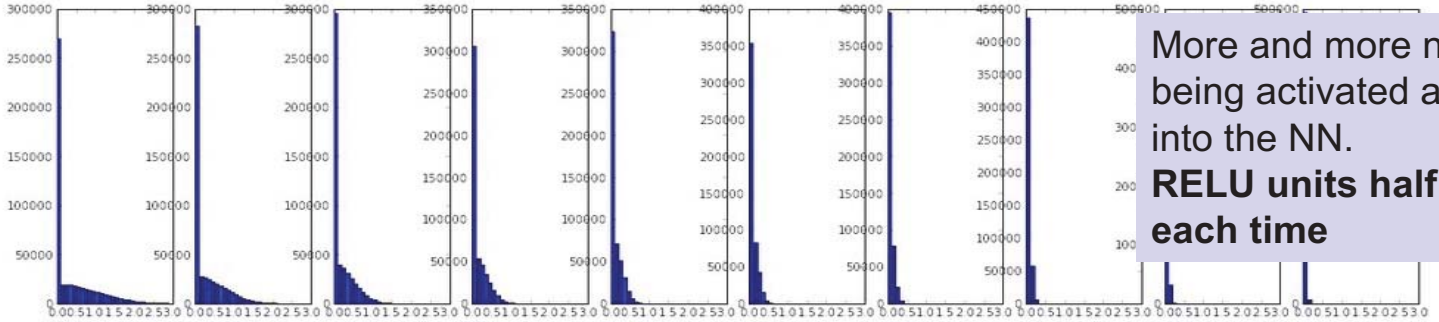
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

but when using the ReLU nonlinearity it breaks.





Kill half the variance as lots of neurons are not activated. So a factor of two of activations go to zero so variance drops over time (over each layer)



More and more neurons are not being activated as we get deep into the NN.
RELU units half your variance each time

Standard Error of the mean decreases

Standard deviations of averages are smaller than standard deviations of individual observations.

It's a consequence of the simple fact that the standard deviation of the sum of two random variables is smaller than the sum of the standard deviations (it can only be equal when the two variables are perfectly correlated).

In fact, when you're dealing with uncorrelated random variables, we can say something more specific: the variance of a sum of variates is the sum of their variances.

This means that with n independent (or even just uncorrelated) variates with the same distribution, the variance of the mean is the variance of an individual divided by the sample size.

http://en.wikipedia.org/wiki/Variance#Basic_properties

$$\sigma_{\bar{X}} = \sigma/\sqrt{n}.$$

Correspondingly with n independent (or even just uncorrelated) variates with the same distribution, the standard deviation of their mean is the standard deviation of an individual divided by the square root of the sample size:

$$\sigma_{\bar{X}} = \sigma/\sqrt{n}.$$

http://stats.stackexchange.com/questions/129885/why-does-increasing-the-sample-size-lower-the-variance

So as you add more data, you ffect applies in regression problems.

Since we can get more precise estimates of averages by increasing the sample size, we are more easily able to tell apart means which are close together -- even though the distributions overlap quite a bit, by taking a large sample size we can still estimate their population means accurately enough to tell that they're not the same.

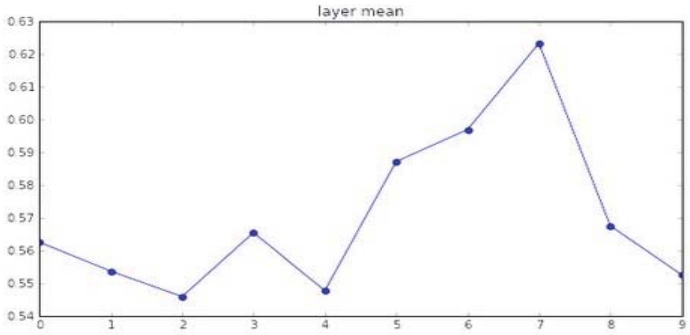
Standard error about the mean needs to consider that the number of activation nodes is not 500 but 250

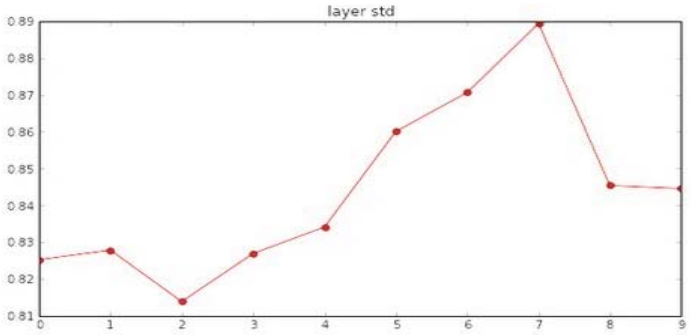
```
input layer had mean 0.000501 and
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

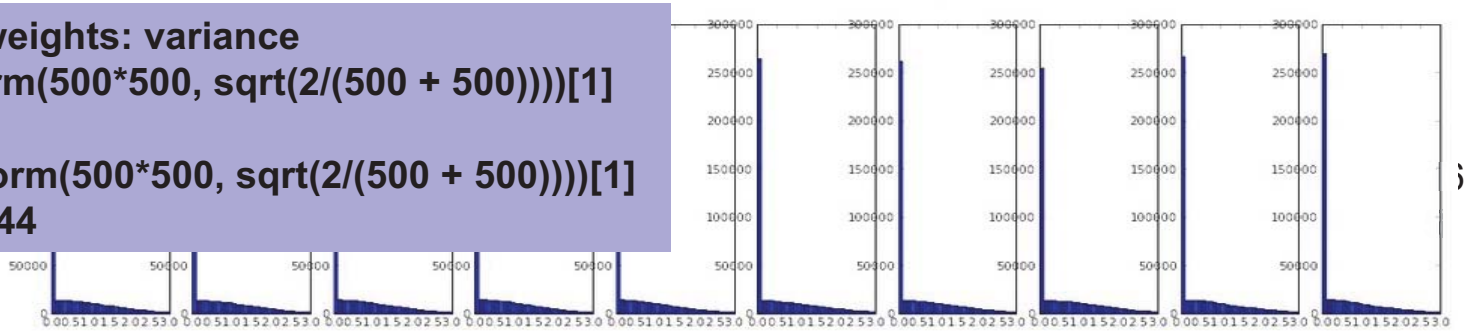
ReLU is aggressive kills half the activations (if your inputs are unit gaussian)
So it will HALF you variance
…so take this into account

He et al., 2015
(note additional /2)





- Num of weights: variance
- > sd(rnorm(500*500, sqrt(2/(500 + 500))))[1]
- 1.001566
- mean(rnorm(500*500, sqrt(2/(500 + 500))))[1]
- 0.04508544

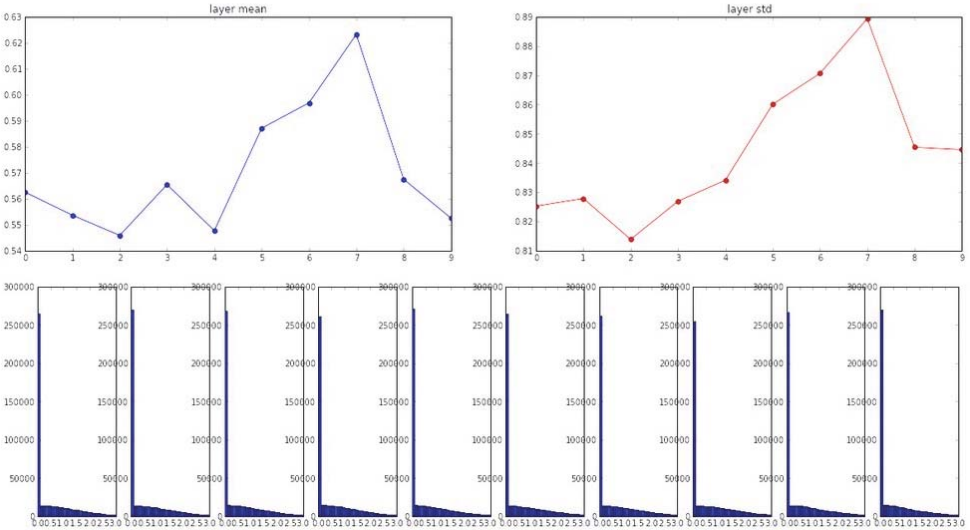


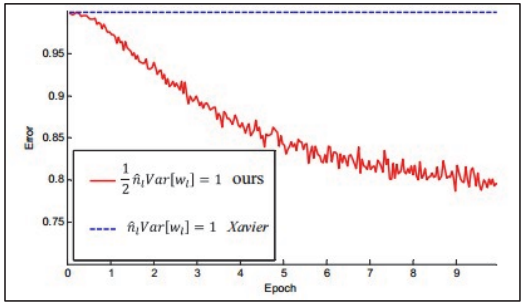
Now everything works ok

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

He et al., 2015
(note additional /2)





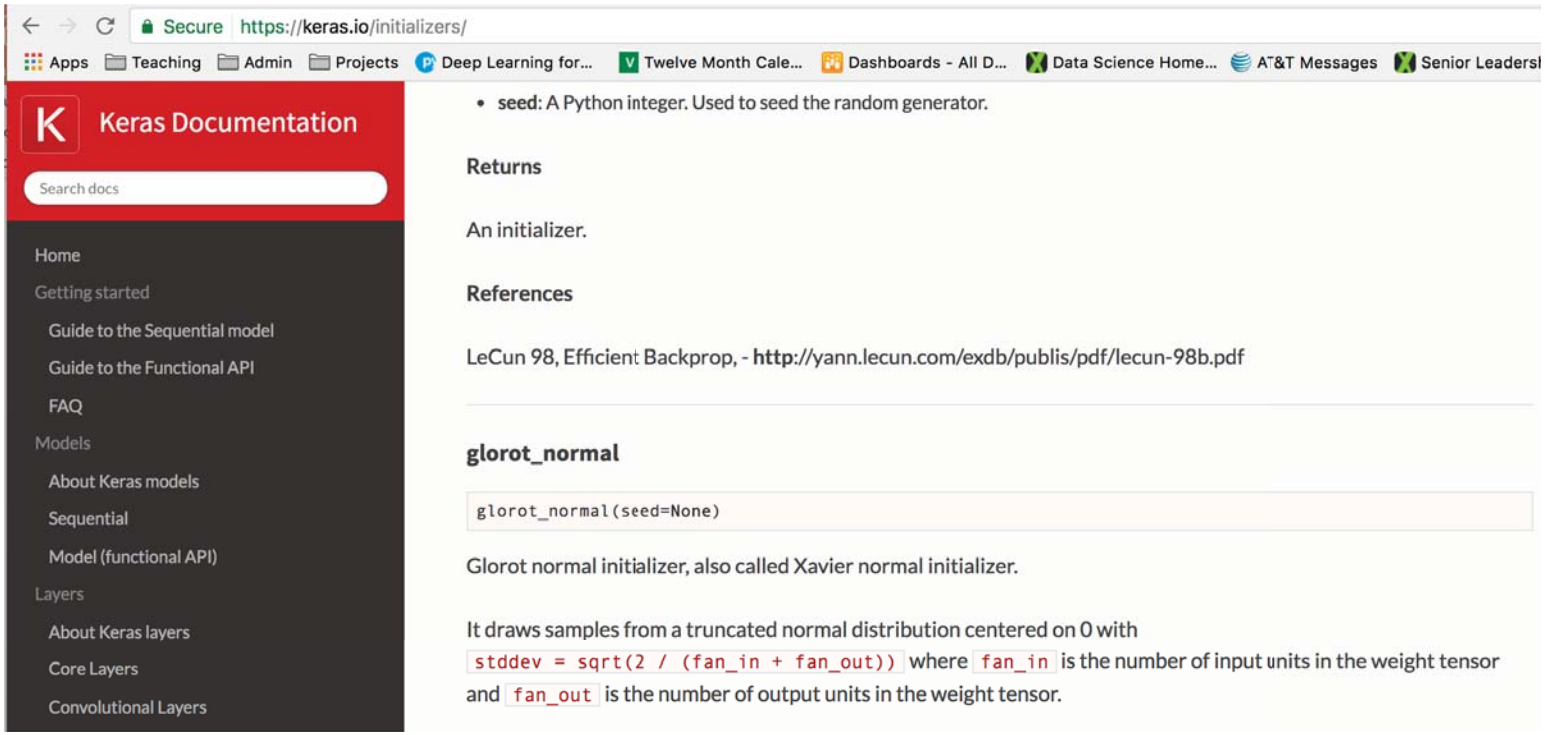
16

Experiment Converges in RED

Initialization of weights

- **Factor of 2 matters**
- **May not converge…. (see paper from He et al. 2015)**

Keras has Glorot init (with the factor of two)



Proper initialization is an active area of research

Understanding the difficulty of training deep feedforward neural networks by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

All you need is a good init, Mishkin and Matas, 2015

…

Data dependent: unit Gaussian; scale weights to unit Gaussian

Init with small random numbers

- **CASE 1: Small random numbers: init W_i**
  - (gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

  - Activations go to ZERO, weights are close to zero so gradient goes to zero
- **CASE 2: bigger small random numbers**

```
W =      np.random.randn(D,H)
```

  - Network get saturated as the activations get pegged at 1 or -1 Tanh so gradient goes to zero ($d(\tanh)/dx = 1 - \tanh(x)^2 = 0$
- **CASE 3: between 0.1 and 1…but where?**
  - Can we get more principled? **Calibrating the variances with 1/sqrt(n)….→ 1/sqrt(n)/2 for ReLUs**

Initialization parameters for each type of activation function

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{inputs} + n_{outputs}}}$	$\sigma = \sqrt{\frac{2}{n_{inputs} + n_{outputs}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{inputs} + n_{outputs}}}$	$\sigma = 4\sqrt{\frac{2}{n_{inputs} + n_{outputs}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{inputs} + n_{outputs}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{inputs} + n_{outputs}}}$

Using the Xavier initialization strategy can speed up training considerably, and it is one of the tricks that led to the current success of Deep Learning. Some recent papers have provided similar strategies for different activation functions, as shown here. The initialization strategy for the ReLU activation function (and its variants, including the ELU activation described shortly) is sometimes called He initialization (Kamming He).

Mini-Batch Normalization

- Want unit Gaussian activations in each layer
- Careful initialization can help
- BUT…. Batch Normalization is a good alternative to bad initialization
- Mini-Batch Normalization of the weighted sum matrix (or activation) on a per neuron basis (per column basis)

Mini-Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations in all parts of your network? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

Incorporate the calculation into the forward prop (so then backprop will work naturally)
Add it as a BN layer
For each single feature do this normalization (standardization)

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

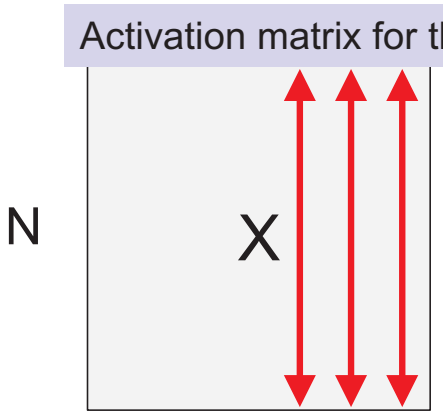
Mini-Batch Normalization

- A recently developed technique by Ioffe and Szegedy called Batch Normalization alleviates a lot of headaches with properly initializing neural networks by explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training.
- The core observation is that this is possible because normalization is a simple differentiable operation.
- In the implementation, applying this technique usually amounts to insert the BatchNorm layer immediately after fully connected layers (or convolutional layers, as we'll soon see), and before non-linearities.
- See paper for more details
- NOTE: that it has become a very common practice to use Batch Normalization in neural networks.

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations?
just make them so."



1. Compute the empirical mean and variance independently for each dimension.

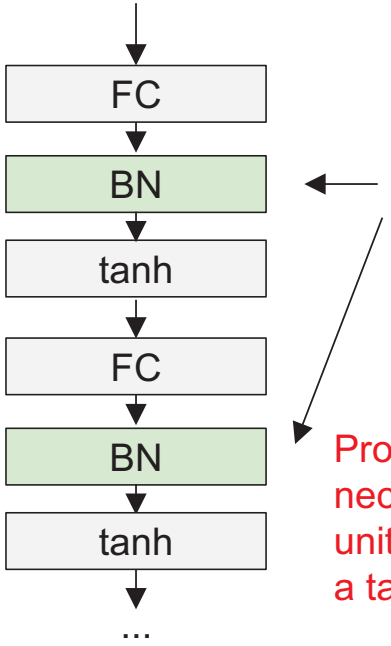
2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Should be called MINI-Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected / (or Convolutional, as we'll see soon) layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

Does tanh want to receive N(0, 1)

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

17
6

Batch Normalization Part 2
Scale or shift the activations

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

Shift by gamma abd B

Shift and scale activations…more peaky or saturated; BP can finetune

17
7

Batch Normalization: Nice properties

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

[Ioffe and Szegedy, 2015]

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Learning with a batch so it helps regularize …

17
8

Batch Normalization at run/test time

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

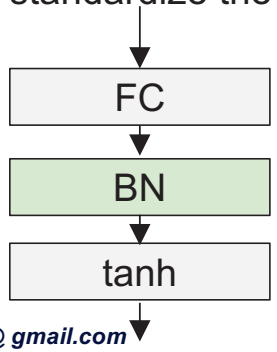
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Note: at test time BatchNorm layer functions differently:
The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

30% more expensive at run time as we have to standardize the weighted sums



Keras

BatchNormalization [source]

```
keras.layers.normalization.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scale
```

Batch normalization layer (Ioffe and Szegedy, 2014).

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

Arguments

- **axis**: Integer, the axis that should be normalized (typically the features axis). For instance, after a `Conv2D` layer with `data_format="channels_first"`, set `axis=1` in `BatchNormalization`.
- **momentum**: Momentum for the moving average.
- **epsilon**: Small float added to variance to avoid dividing by zero.
- **center**: If True, add offset of `beta` to normalized tensor. If False, `beta` is ignored.
- **scale**: If True, multiply by `gamma`. If False, `gamma` is not used. When the next layer is linear (also e.g. `nn.r`
  disabled since the scaling will be done by the next layer.
- **beta_initializer**: Initializer for the beta weight.
- **gamma_initializer**: Initializer for the gamma weight.
- **moving_mean_initializer**: Initializer for the moving mean.
- **moving_variance_initializer**: Initializer for the moving variance.
- **beta_regularizer**: Optional regularizer for the beta weight.
- **gamma_regularizer**: Optional regularizer for the gamma weight.
- **beta_constraint**: Optional constraint for the beta weight.
- **gamma_constraint**: Optional constraint for the gamma weight.

Input shape

Arbitrary. Use the keyword argument `input_shape` (tuple of integers, does not include the sample
using this layer as the first layer in a model.

Output shape

Same shape as input.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Activations should be unit Gaussian everywhere!

“you want unit Gaussian activations in all parts of your network? just make them so.”

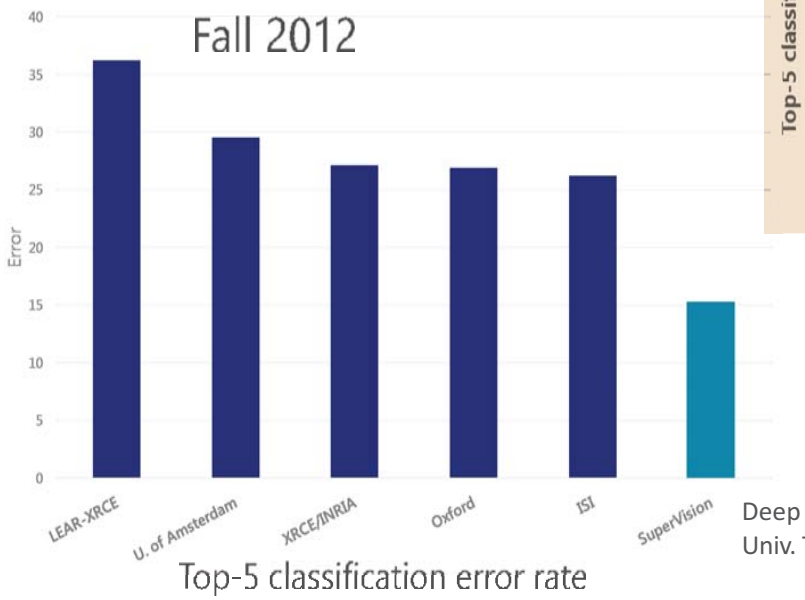
- **Glorot-He Intialization**
- **(mini) Batch normalization**

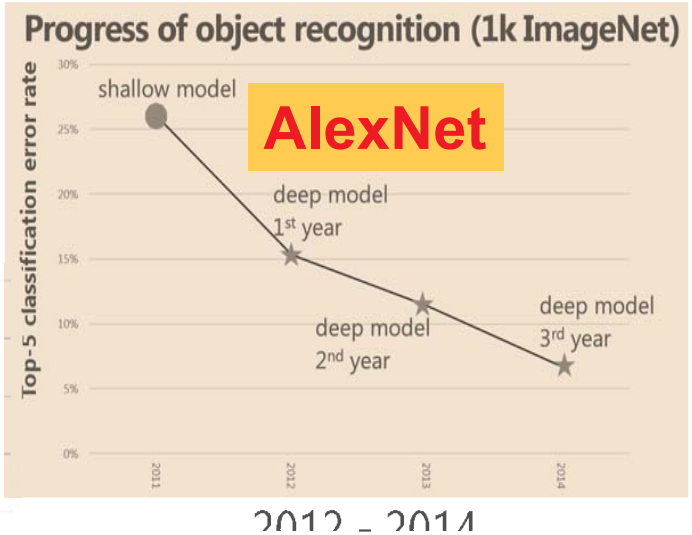
ImageNet

ImageNet dataset that was first collected by Deng et al. 2009, and has been organized by a lab at Stanford since 2010

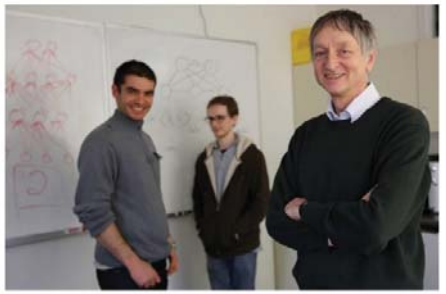
ImageNet 1K Competition

Krizhevsky, Sutskever, Hinton, "ImageNet Classification with Deep Convolutional Neural Networks." *NIPS*, Dec. 2012





Deep CNN !!!
Univ. Toronto team

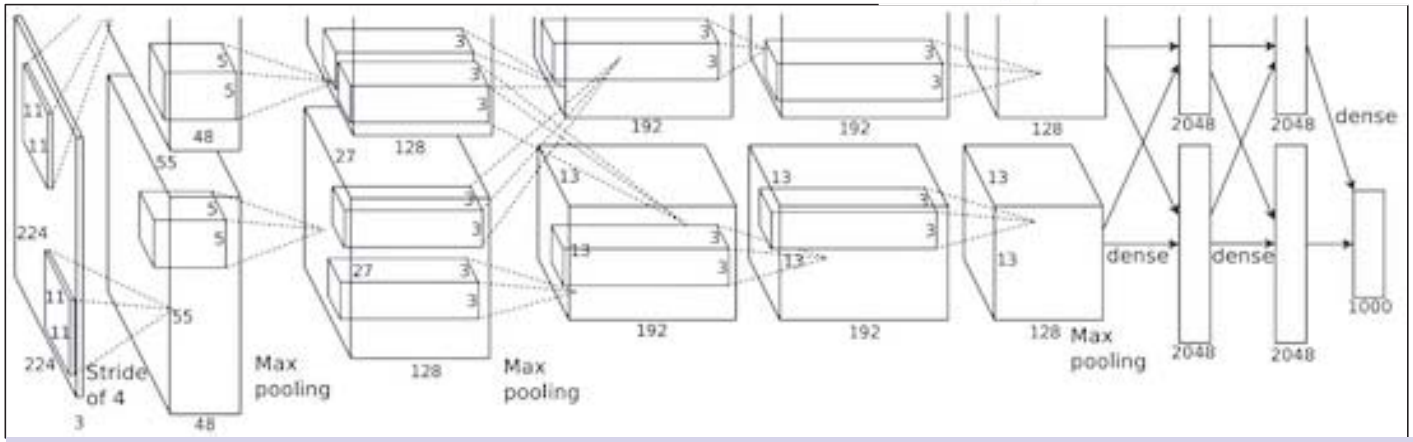


Ilya Sutskever, Alex Krizhevsky, Geoffrey Hinton

ImageNet

ImageNet Classification with Deep Convolutional Neural Networks
[Krizhevsky, Sutskever, Hinton, 2012]

IMAGENET



"AlexNet": Architecture similar to LeNet but bigger, ReLU, GPU, more training data. LeNet used bigger kernels (5x5) versus 3x3

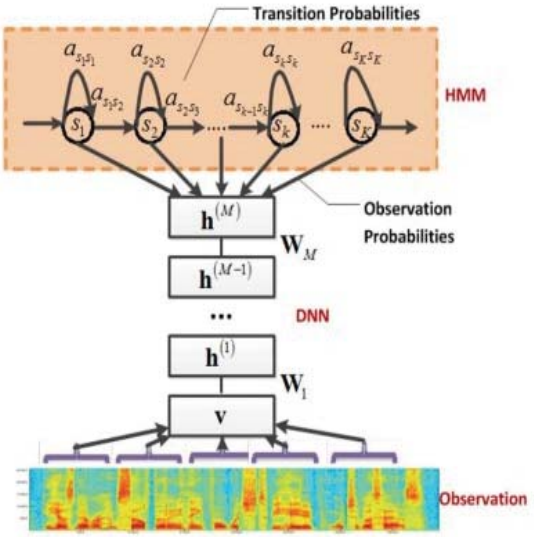
60 Million parameters

VGGNet

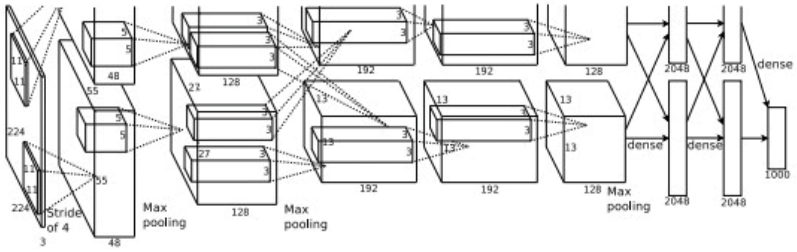
2010 Speech + 2012 Image Breakthrough

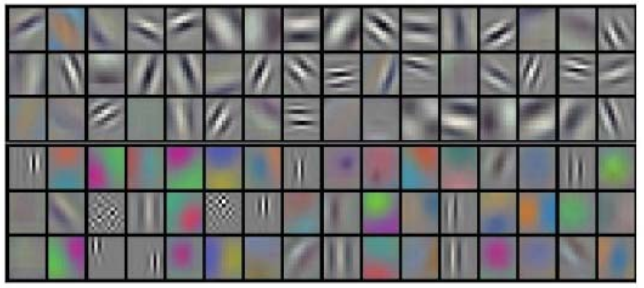
Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition
George Dahl, Dong Yu, Li Deng, Alex Acero, 2010

Spectacular December 2012 **live demonstration** of instant **English-to-Chinese voice recognition and translation** by Microsoft Research chief Rick Rashid



Imagenet classification with deep convolutional neural networks
Alex Krizhevsky, Ilya Sutskever, Geoffrey E Hinton, 2012





~1700s	• The prologue
1940-1970	• Act 1 Tinker: Hack it up
1970-1995	• Act 2 BackProp: theory to the rescue
1995-2007	• Act 3 Layer by layer learning, a medieval pastime
2007-2015	• Act 4 Introspection: better init. and activation functions
2015 -	• Act 5 Express-laning the gradient: Skip Connections, the SoTA frontier (LSTMs, ResNet, Highway Nets, DenseNets)
	• The epilogue

Gating memory/history

- **Controllable gating mechanisms is the basis of the LSTM and the GRU architectures**
- **At each time step, differentiable gating mechanisms decide which parts of the inputs will be written to memory, and which parts of memory will be overwritten (forgotten).**

- **LSTMs has a state vector:**
  - Memory and working memory

View DL training as a time series

Predict the next time step using:

- **Naive:** $F_{t+1} = A_t$
  - The forecast is equal to the actual value observed during the last period – good for level patterns
- **Simple Mean:** $F_{t+1} = \sum A_t / n$
  - The average of all available data - good for level patterns
- **Moving Average:** $F_{t+1} = \sum A_t / n$
  - The average value over a set time period (e.g.: the last four weeks)
  - Each new forecast drops the oldest data point & adds a new observation
  - More responsive to a trend but still lags behind actual data

Time Series Models: Exponential Smoothing

Predict the next time step using:

- Exponential Smoothing:

$$F_{t+1} = \alpha A_t + (1-\alpha) F_t$$

Most frequently used time series method because of ease of use and minimal amount of data needed

- **Need just three pieces of data to start:**
  - Last period's forecast (F_t)
  - Last periods actual value (A_t)
  - Select value of smoothing coefficient, α, between 0 and 1.0
- **If no last period forecast is available, average the last few periods or use naive method**
- **Higher α values (e.g. .7 or .8) may place too much weight on last period's random variation**

Gated memory: binary gate here

$$\mathbf{F}_{t+1} = \alpha \mathbf{A}_t + (1-\alpha)\mathbf{F}_t$$

$$\begin{bmatrix} 8 \\ 11 \\ 3 \\ 7 \\ 5 \\ 15 \end{bmatrix} \leftarrow \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \odot \begin{bmatrix} 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \odot \begin{bmatrix} 8 \\ 9 \\ 3 \\ 7 \\ 5 \\ 8 \end{bmatrix}$$

$\mathbf{s}'$ $\mathbf{g}$ $\mathbf{x}$ $(\mathbf{1}-\mathbf{g})$ $\mathbf{s}$

Using binary gate vector g (AKA α) to control access to memory s' .

Gated memory: binary gate here

$$\mathbf{F}_{t+1} = \alpha \mathbf{A}_t + (1-\alpha)\mathbf{F}_t$$

$$\begin{bmatrix} 8 \\ 11 \\ 3 \\ 7 \\ 5 \\ 15 \end{bmatrix} \leftarrow \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \odot \begin{bmatrix} 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \odot \begin{bmatrix} 8 \\ 9 \\ 3 \\ 7 \\ 5 \\ 8 \end{bmatrix}$$

$\mathbf{s}'$ $\mathbf{g}$ $\mathbf{x}$ $(\mathbf{1}-\mathbf{g})$ $\mathbf{s}$

Using binary gate vector g (AKA α) to control access to memory s' .

Gates should not static but be controlled by the current memory state and the input, and their behavior should be learned.
This introduced an obstacle, as learning in our framework entails being differentiable (because of the backpropagation algorithm) and the binary 0-1 values used in the gates are not differentiable

Gating memory based on memory aand current value

- **A solution to the above problem is to approximate the hard gating mechanism with a soft—but differentiable—gating mechanism.**
- **To achieve these differentiable gates, we replace the requirement that $g \in \{0, 1\}^n$ and allow arbitrary real numbers, $g \in R^n$, which are then passed through a sigmoid function $\sigma(g')$.**
  - This bounds the value in the range (0, 1}, with most values near the borders.
  - When using the gate $\sigma(g') \odot x$, indices in x corresponding to near-one values in $\sigma(g')$ are allowed to pass, while those corresponding to near-zero values are blocked.
- **The gate values can then be conditioned on the input and and the current memory and trained using a gradient-based method to perform a desired behavior**

LSTM: Input, Forget, and Output

$$s_j = R_{\text{LSTM}}(s_{j-1}, x_j) = [c_j; h_j]$$

$$c_j = f \odot c_{j-1} + i \odot z$$

$$h_j = o \odot \tanh(c_j)$$

$$i = \sigma(x_j W^{xi} + h_{j-1} W^{hi})$$

$$f = \sigma(x_j W^{xf} + h_{j-1} W^{hf})$$

$$o = \sigma(x_j W^{xo} + h_{j-1} W^{ho})$$

$$z = \tanh(x_j W^{xz} + h_{j-1} W^{hz})$$

$$y_j = O_{\text{LSTM}}(s_j) = h_j$$

$$s_j \in \mathbb{R}^{2 \cdot d_h}, \; x_i \in \mathbb{R}^{d_x}, \; c_j, h_j, i, f, o, z \in \mathbb{R}^{d_h}, \; W^{xo} \in \mathbb{R}^{d_x \times d_h}, \; W^{ho} \in \mathbb{R}^{d_h \times d_h}.$$

c_j is the memory component: weighted sum of memory and current state
z combo of input and previous state

LSTM has three gates, i , f , and o, controlling for input, forget, and output.

The state at time j is composed of two vectors, cj and hj ,

- where cj is the memory component
- and hj is the hidden state component.

LSTMs equations explained

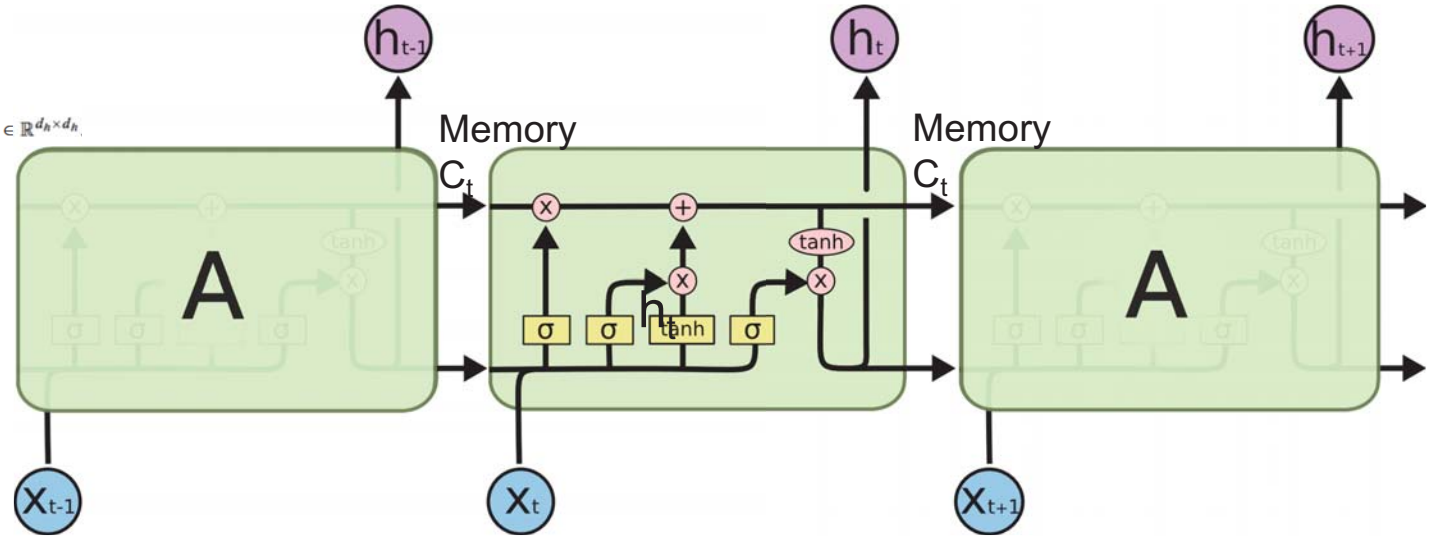
The state at time j is composed of two vectors, c_j and h_j, where c_j is the memory component and h_j is the hidden state component. There are three gates, i, f, and o, controlling for input, forget, and output. The gate values are computed based on linear combinations of the current input x_j and the previous state h_{j-1}, passed through a sigmoid activation function. An update candidate z is computed as a linear combination of x_j and h_{j-1}, passed through a tanh activation function. The memory c_j is then updated: the forget gate controls how much of the previous memory to keep ($f \odot c_{j-1}$), and the input gate controls how much of the proposed update to keep ($i \odot z$). Finally, the value of h_j (which is also the output y_j) is determined based on the content of the memory c_j, passed through a tanh nonlinearity and controlled by the output gate. The gating mechanisms allow for gradients related to the memory part c_j to stay high across very long time ranges.

LSTMs graphically: State (h_t) and memory (c_t)

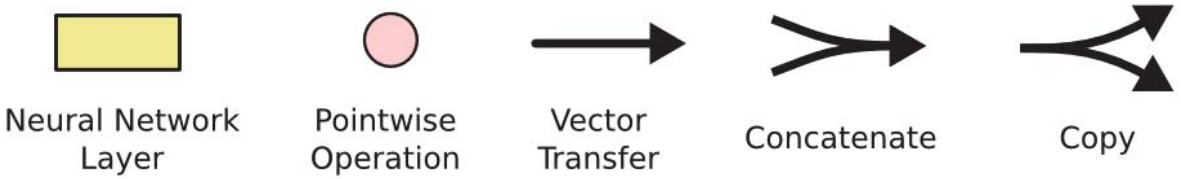
$$s_j = R_{\text{LSTM}}(s_{j-1}, x_j) = [c_j; h_j]$$
$$c_j = f \odot c_{j-1} + i \odot z$$
$$h_j = o \odot \tanh(c_j)$$
$$i = \sigma(x_j W^{xi} + h_{j-1} W^{hi})$$
$$f = \sigma(x_j W^{xf} + h_{j-1} W^{hf})$$
$$o = \sigma(x_j W^{xo} + h_{j-1} W^{ho})$$
$$z = \tanh(x_j W^{xz} + h_{j-1} W^{hz})$$

$$y_j = O_{\text{LSTM}}(s_j) = h_j$$

$s_j \in \mathbb{R}^{2 \cdot d_h}$, $x_i \in \mathbb{R}^{d_x}$, $c_j, h_j, i, f, o, z \in \mathbb{R}^{d_h}$, $W^{xo} \in \mathbb{R}^{d_x \times d_h}$, $W^{ho} \in \mathbb{R}^{d_h \times d_h}$

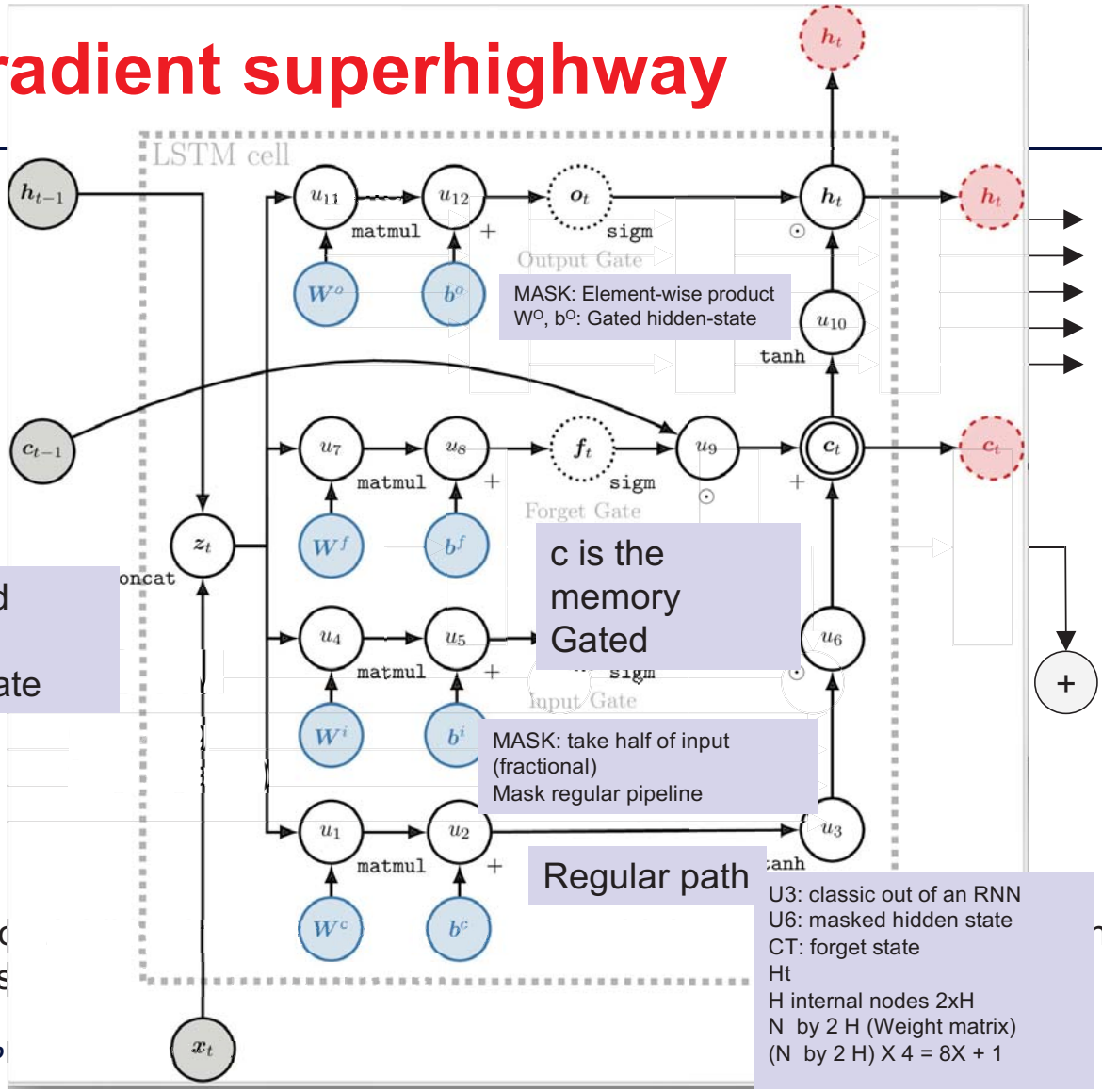


http://colah.github.io/posts/2015-08-Understanding-LSTMs/



LSTM: gradient superhighway

See Slide 110



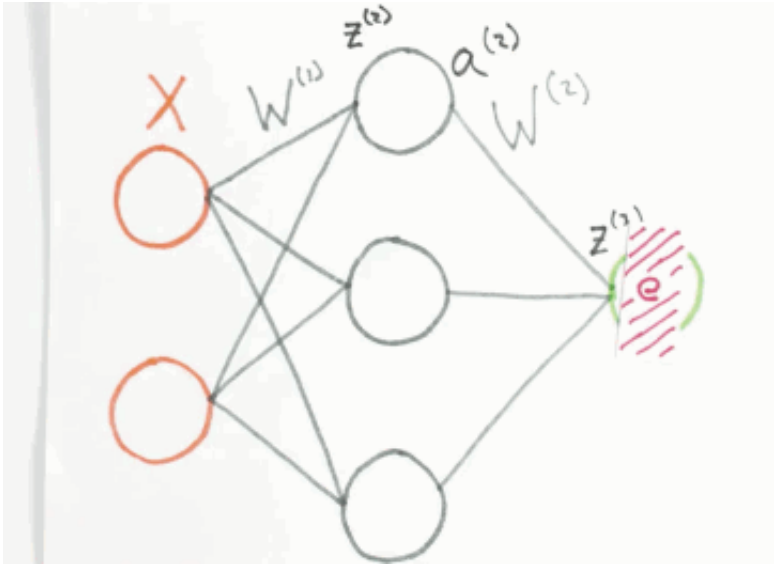
Zt is Xt and output from previous state

In BackPr
Can bypas
nts

Long Short-Term Memory (LSTMs), GRUs

- **Long Short-Term Memory (LSTM) architecture [Hochreiter and Schmidhuber, 1997]**
- **LSTM were designed to solve the vanishing gradients (think long term history) problem, and is the first to introduce the gating mechanism.**
- **GRU (gated recurrent unit)**
  - The gated recurrent unit (GRU) was recently introduced by Cho et al. [2014b] as an alternative to the LSTM. It was subsequently shown by Chung et al. [2014] to perform comparably to the LSTM on several (non textual) datasets.

Backpropagation



$$\frac{\partial J_{MSE}}{\partial W_{jk}} = \frac{\partial}{\partial W_{jk}} \frac{1}{2}(O_k - t_k)^2$$

weight example

$$\frac{\partial E}{\partial W} = (O - t)\,\boldsymbol{X}$$

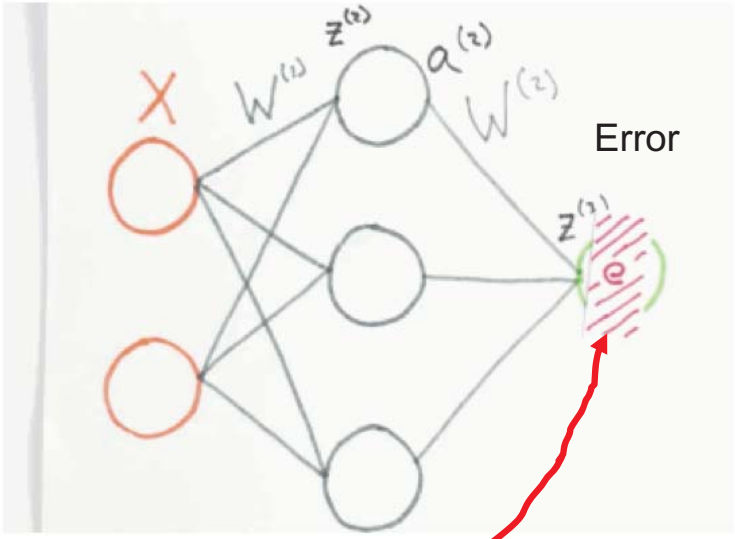
$$W = W - \alpha \times \frac{\partial E}{\partial W}$$

$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} \mathbf{34} \\ \mathbf{1} \\ \mathbf{4} \end{bmatrix}$$

$$4.32 = 5 - 0.01 \times (2 \times 34) \text{ for } i = 1$$

http://www.cbcity.de/tutorial-neuronale-netze-einfach-erklaert

Backpropagation with skip connection



$$\frac{\partial J_{MSE}}{\partial W_{jk}} = \frac{\partial}{\partial W_{jk}} \frac{1}{2}(O_k - t_k)^2$$

weight example

$$\frac{\partial E}{\partial W} = (O - t)\,\boldsymbol{X}$$

$$W = W - \alpha \times \frac{\partial E}{\partial W}$$

$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} 34 \\ 1 \\ 4 \end{bmatrix}$$

$$4.32 = 5 - 0.01 \times (2 \times 34) \text{ for } i = 1$$

Error

F(X) = H(X) + X

Express-lane the gradient thru summation

E.g, when H(X) ==0 then W can be viewed as Identity weight matrix (because of the +X factor)

residual

http://www.cbcity.de/tutorial-neuronale-netze-einfach-erklaert

Skip connections:

By stacking these layers, the gradient could theoretically "skip" over all the intermediate layers and reach the bottom without being diminished.

Oh my: a Non-vanishing Gradient

While this is the intuition, the actual implementation is a little more complex.

Back propagation algorithm

1. Run the network forward with your input data to get the activations (O)
2. For each output node compute the error

$$\delta_k = O_k(1 - O_k)(O_k - t_k)$$

3. For each hidden node calulate credit assignment

$$\delta_J = O_j(1 - O_j) \sum_{k \in K} \delta_k W_{jk}$$

4. Update the weights and biases as follows:

For each layer l-1 (Letter elle) (each layer W_l, b_l)

$$\nabla W_l = -\eta \delta_l O_{l-1}$$

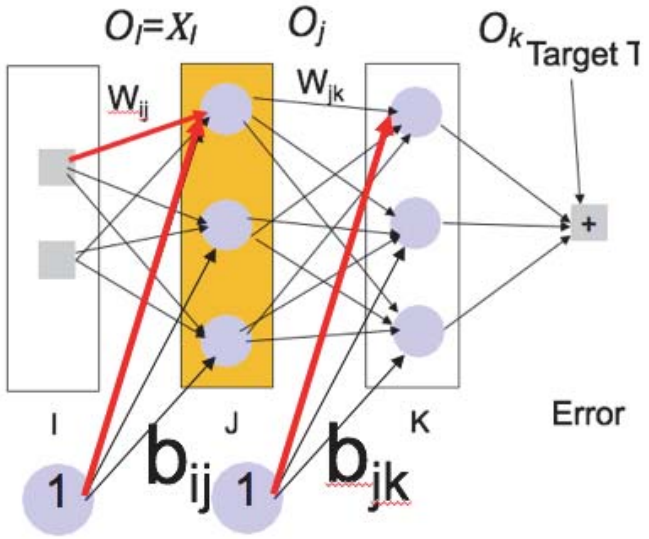
$$\nabla b_l = -\eta \delta_l$$

∇ Nabla

apply

$$W_l = W_l + \nabla W_l$$

$$b_l = b + \nabla b_l$$



Network Architecture: I-H-O
CX Loss + regularization

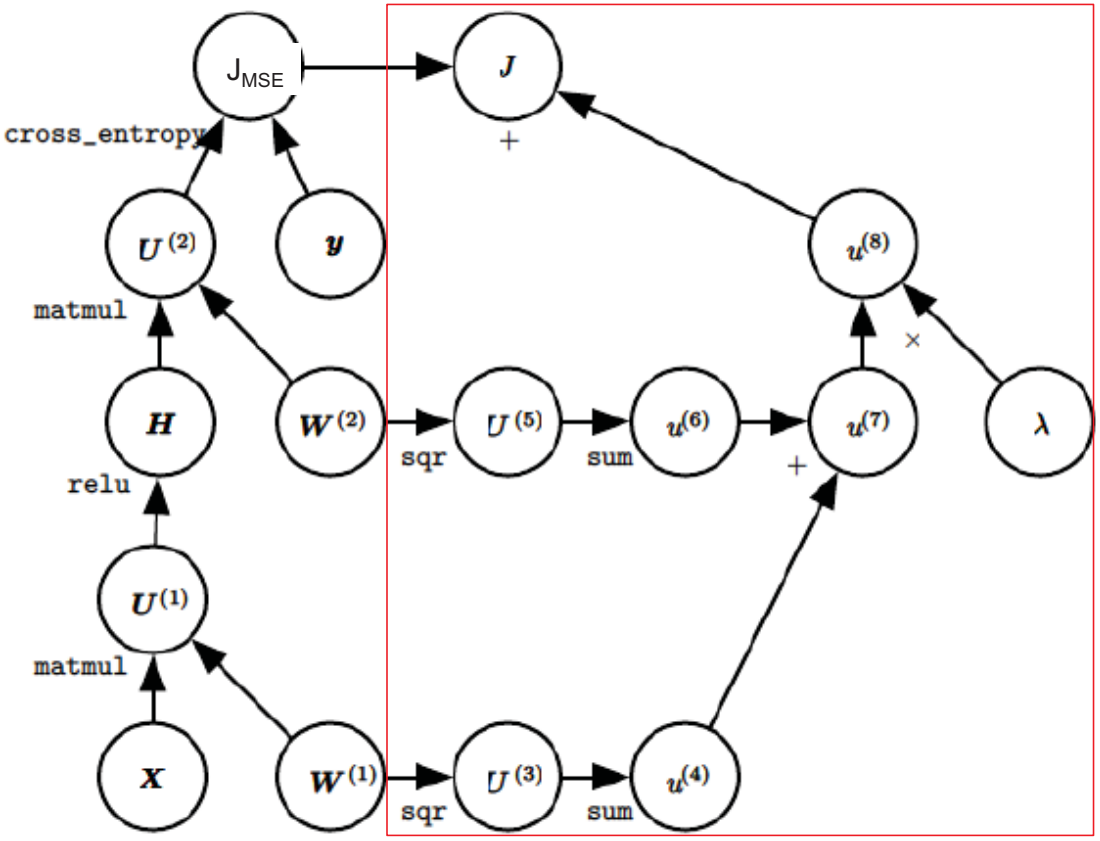


Figure 6.11: The computational graph used to compute the cost used to train our example of a single-layer MLP using the cross-entropy loss and weight decay.

$$\frac{\partial J_{MSE}}{\partial W_{jk}} = \frac{\partial}{\partial W_{jk}} \frac{1}{2}(O_k - t_k)^2$$

1. Run the network forward with your input data to get the activations (O)
2. For each output node compute the error
   $\delta_k = O_k(1-O_k)(O_k - t_k)$
3. For each hidden node calulate credit assignment
   $\delta_J = O_j(1 - O_j) \sum_{k \in K} \delta_k W_{jk}$
4. Update the weights and biases as follows:

For each layer l-1 (Letter elle) (each layer W_l, b_l)

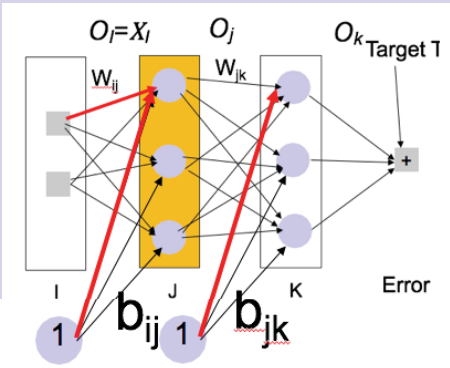
$$\nabla W_l = -\eta \delta_l O_{l-1}$$

$$\nabla b_l = -\eta \delta_l$$

apply

$$W_l = W_l + \nabla W_l$$

$$b_l = b + \nabla b_l$$



▽ the gradient chorus

- **What is the gradient for linear regression?**

- **Chorus**

Linear Regression
  - The gradient is the weighted sum of the training data, where the weights are proportional to the error (for each example) !

NN
  - The gradient is the weighted sum of the **input** data, where the weights are proportional to the error **or the error proxy** (for each example) !

weight *example*

$$\frac{\partial E}{\partial W} = (O - t)\,\boldsymbol{X}$$

$$W = W - \alpha \times \frac{\partial E}{\partial W}$$

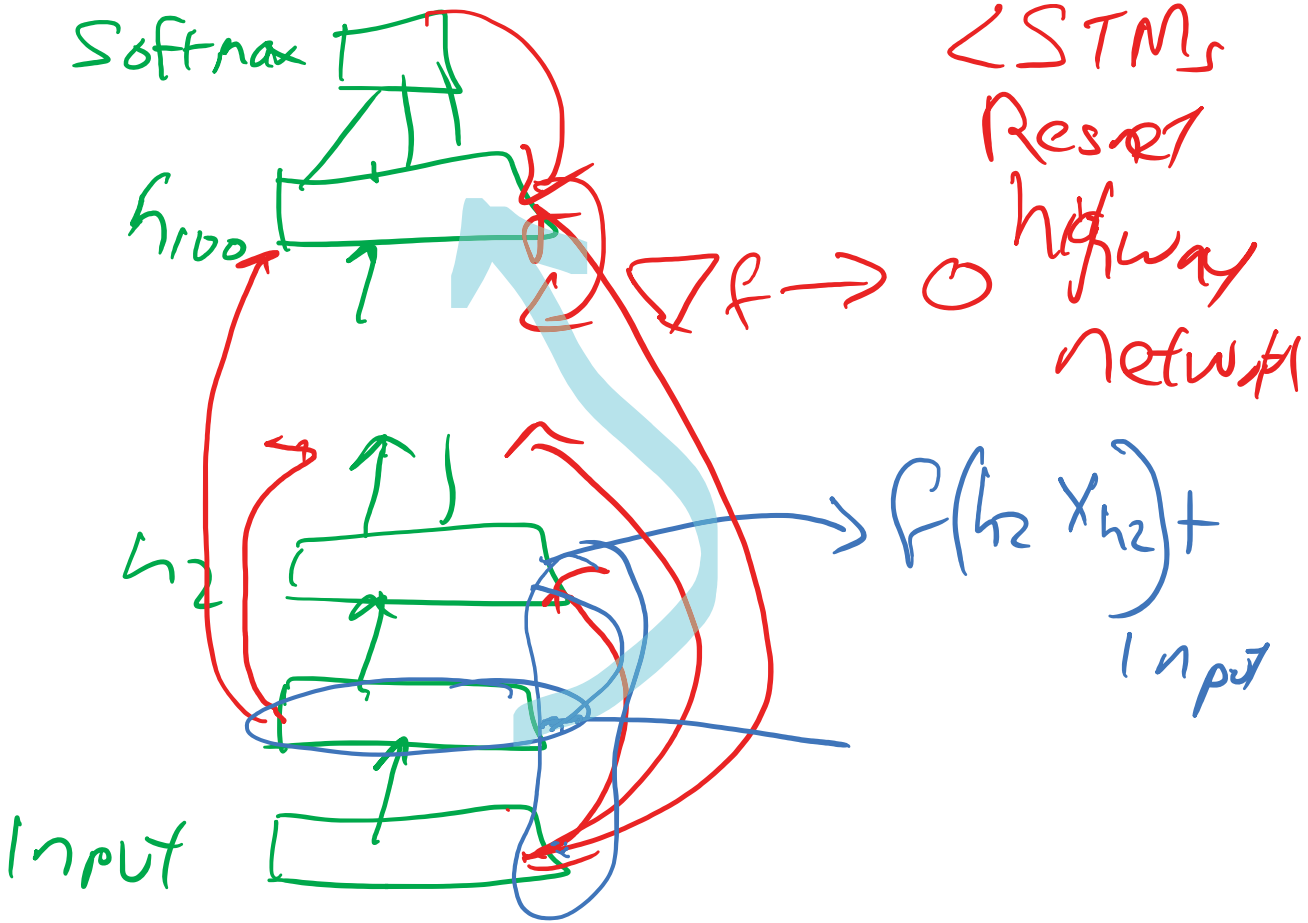
$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} \mathbf{34} \\ \mathbf{1} \\ \mathbf{4} \end{bmatrix}$$

$$4.32 = 5 - 0.01 \times (2 \times 34) \text{ for } i = 1$$

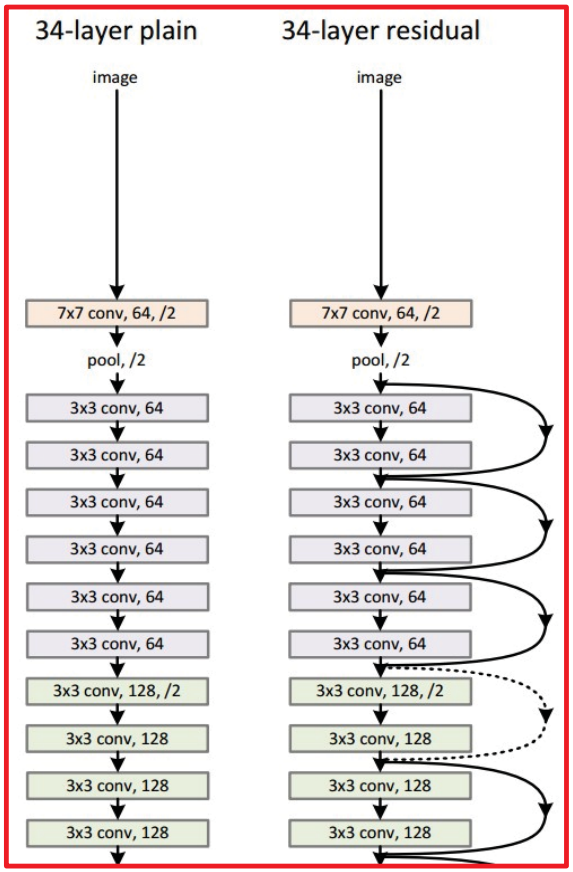
Skip connections as means to combat vanishing gradients

-
- By stacking these layers, the gradient could theoretically “skip” over all the intermediate layers and reach the bottom without being diminished.
- Oh my: a Non-vanishing Gradient
- While this is the intuition, the actual implementation is a little more complex.

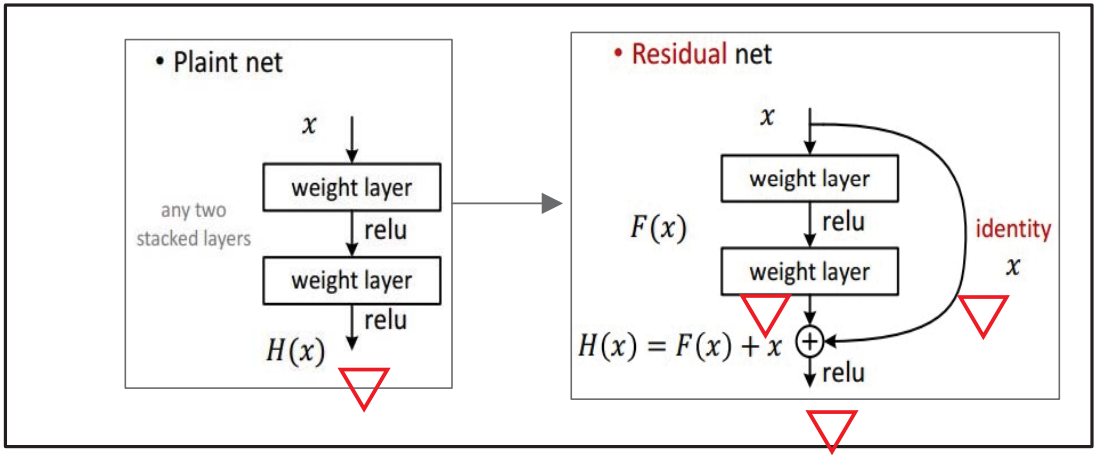
Resnet and highway networks



PlainNets" vs. ResNets



***ResNet is to PlainNet what LSTM is to RNN*, kind of.**

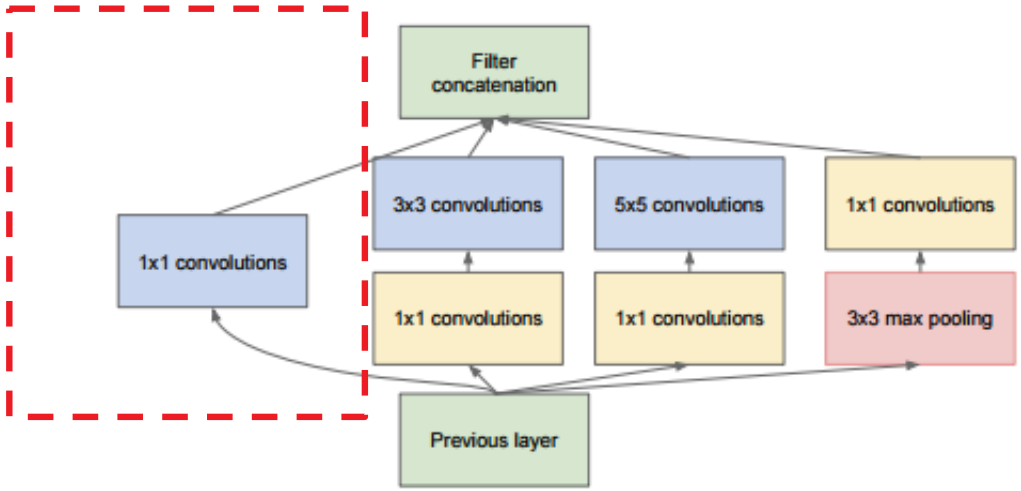


Shortcut Connections has a history

- "highway networks" present shortcut connections with gating functions (like in LSTMs).
- These gates are data-dependent and have parameters, in contrast to our identity shortcuts that are parameter-free.
- When a gated shortcut is "closed" (approaching zero), the layers in highway networks represent non-residual functions.

Inception v1 Module

an “inception” layer is composed of a shortcut branch and a few deeper branches.



GoogLeNet → Inception v1, V2, V3

- **In the paper Batch Normalization,Sergey et al,2015. proposed Inception-v1 architecture which is a variant of the GoogleNet in the paper Going deeper with convolutions, and in the meanwhile they introduced Batch Normalization to Inception(BN-Inception).**
  - The main difference to the network described in (Szegedy et al.,2014) is that the 5x5 convolutional layers are replaced by two consecutive layer of 3x3 convolutions with up to 128 filters.
- As for **Inception-v3**, it is a variant of Inception-v2 which adds BN-auxiliary.

Highway networks have encouraging results but..

- **Highway networks have not demonstrated accuracy gains with extremely increased depth (e.g., over 100 layers).**

method			error (%)
Maxout [10]			9.38
NIN [25]			8.81
DSN [24]			8.22
	# layers	# params	
FitNet [35]	19	2.5M	8.39
Highway [42, 43]	19	2.3M	7.54 (7.72±0.16)
Highway [42, 43]	32	1.25M	8.80
ResNet	20	0.27M	8.75
ResNet	32	0.46M	7.51
ResNet	44	0.66M	7.17
ResNet	56	0.85M	6.97
ResNet	110	1.7M	**6.43** (6.61±0.16)
ResNet	1202	19.4M	7.93

Table 6. Classification error on the **CIFAR-10** test set. All methods are with data augmentation. For ResNet-110, we run it 5 times and show "best (mean±std)" as in [43].

Gradients

- **ResNets, HighwayNets, and DenseNets, Oh My!**
  - Implementing really deep neural networks in Tensorflow
  - https://chatbotslife.com/resnets-highwaynets-and-densenets-oh-my-9bb15918ee32
- **Resnet**
  - http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf

Highway networks

Training Very Deep Networks

Rupesh Kumar Srivastava Klaus Greff Jürgen Schmidhuber

The Swiss AI Lab IDSIA / USI / SUPSI
{rupesh, klaus, juergen}@idsia.ch

Abstract

Theoretical and empirical evidence indicates that the depth of neural networks is crucial for their success. However, training becomes more difficult as depth increases, and training of very deep networks remains an open problem. Here we introduce a new architecture designed to overcome this. Our so-called highway networks allow unimpeded information flow across many layers on *information highways*. They are inspired by Long Short-Term Memory recurrent networks and use adaptive gating units to regulate the information flow. Even with hundreds of layers, highway networks can be trained directly through simple gradient descent. This enables the study of extremely deep and efficient architectures.

Highway Network: more general resNet
~~$H(x, W_H) + X$~~

- Highway Network **builds on the ResNet in a pretty intuitive way. The Highway Network preserves the shortcuts introduced in the ResNet, but augments them with a learnable parameter to determine to what extent each layer should be a skip connection or a nonlinear connection.**
- **Layers in a Highway Network are defined as follows:**

$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}) \cdot T(\mathbf{x}, \mathbf{W_T}) + \mathbf{x} \cdot (1 - T(\mathbf{x}, \mathbf{W_T})). \quad (3)$$

- **In this equation we can see an outline of the previous two kinds of layers discussed: *y = H(x,Wh)* mirrors our traditional layer, and *y = H(x,Wh) + x* mirrors our residual unit. What is new is the *T(x,Wt)* function.**
  - This serves at the switch to determine to what extent information should be sent through the primary pathway or the skip pathway.
  - By using *T* and *(1-T)* for each of the two pathways, the activation must always sum to 1.

```python
def highway(x, size, activation, carry_bias=-1.0):
  W_T = tf.Variable(tf.truncated_normal([size, size], stddev=0.1), name="weight_transform")
  b_T = tf.Variable(tf.constant(carry_bias, shape=[size]), name="bias_transform")

  W = tf.Variable(tf.truncated_normal([size, size], stddev=0.1), name="weight")
  b = tf.Variable(tf.constant(0.1, shape=[size]), name="bias")

  T = tf.sigmoid(tf.matmul(x, W_T) + b_T, name="transform_gate")
  H = activation(tf.matmul(x, W) + b, name="activation")
  C = tf.sub(1.0, T, name="carry_gate")

  y = tf.add(tf.mul(H, T), tf.mul(x, C), "y")
  return y
```

Highway networks

- Highway networks allow unimpeded information flow across many layers on information highways.
- They are inspired by Long Short-Term Memory recurrent networks and use adaptive gating units to regulate the information flow.
- Even with hundreds of layers, highway networks can be trained directly through simple gradient descent.

```python
import tensorflow as tf
import numpy as np
import tensorflow.contrib.slim as slim

total_layers = 25 #Specify how deep we want our network
units_between_stride = total_layers / 5

def highwayUnit(input_layer,i):
    with tf.variable_scope("highway_unit"+str(i)):
        H = slim.conv2d(input_layer,64,[3,3])
        T = slim.conv2d(input_layer,64,[3,3], #We initialize with a negative bias to push the netw
            biases_initializer=tf.constant_initializer(-1.0),activation_fn=tf.nn.sigmoid)
        output = H*T + input_layer*(1.0-T)
        return output

tf.reset_default_graph()

input_layer = tf.placeholder(shape=[None,32,32,3],dtype=tf.float32,name='input')
label_layer = tf.placeholder(shape=[None],dtype=tf.int32)
label_oh = slim.layers.one_hot_encoding(label_layer,10)

layer1 = slim.conv2d(input_layer,64,[3,3],normalizer_fn=slim.batch_norm,scope='conv_'+str(0))
for i in range(5):
    for j in range(units_between_stride):
        layer1 = highwayUnit(layer1,j + (i*units_between_stride))
    layer1 = slim.conv2d(layer1,64,[3,3],stride=[2,2],normalizer_fn=slim.batch_norm,scope='conv_s_

top = slim.conv2d(layer1,10,[3,3],normalizer_fn=slim.batch_norm,activation_fn=None,scope='conv_top

output = slim.layers.softmax(slim.layers.flatten(top))

loss = tf.reduce_mean(-tf.reduce_sum(label_oh * tf.log(output) + 1e-10, reduction_indices=[1]))
trainer = tf.train.AdamOptimizer(learning_rate=0.001)
update = trainer.minimize(loss)
```

~1700s • The prologue

1940-1970 • Act 1 Tinker: Hack it up

1970-1995 • Act 2 BackProp: theory to the rescue

1995-2007 • Act 3 Layer by layer learning, a medieval pastime

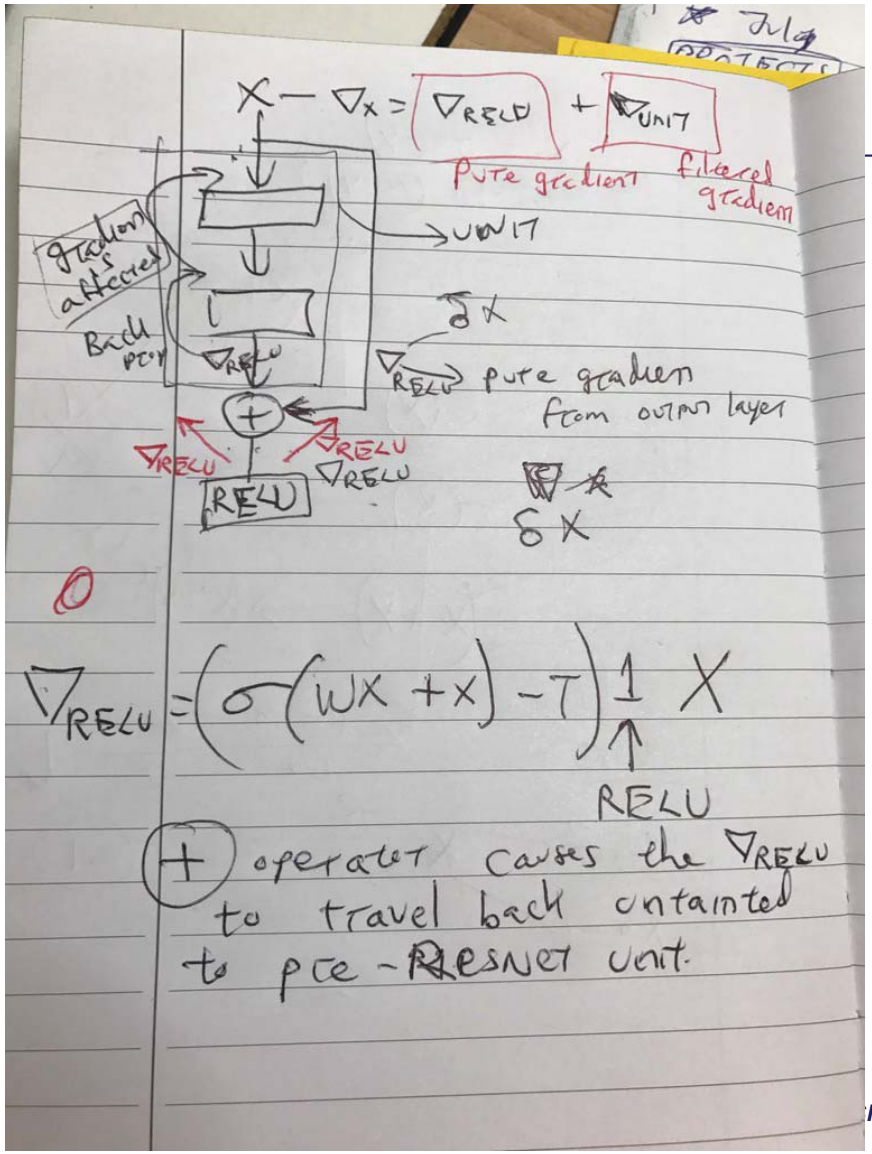
2007-2015 • Act 4 Introspection: better init. and activation functions

2015 - • Act 5 Express-laning the gradient: Skip Connections, the SoTA frontier (LSTMs, ResNet, Highway Nets, DenseNets)

• The epilogue

ResNet: Networks

Gradient of relu function

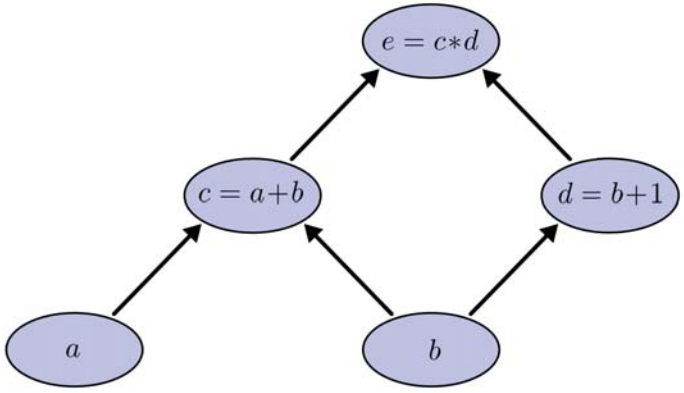


Backpropagation: aka "reverse-mode differentiation."

- Beyond its use in deep learning, backpropagation is a powerful computational tool in many other areas, ranging from weather forecasting to analyzing numerical stability – it just goes by different names. In fact, the algorithm has been reinvented at least dozens of times in different fields (see Griewank (2010)).
- The general, application independent, name is "reverse-mode differentiation."

Computational Graphs

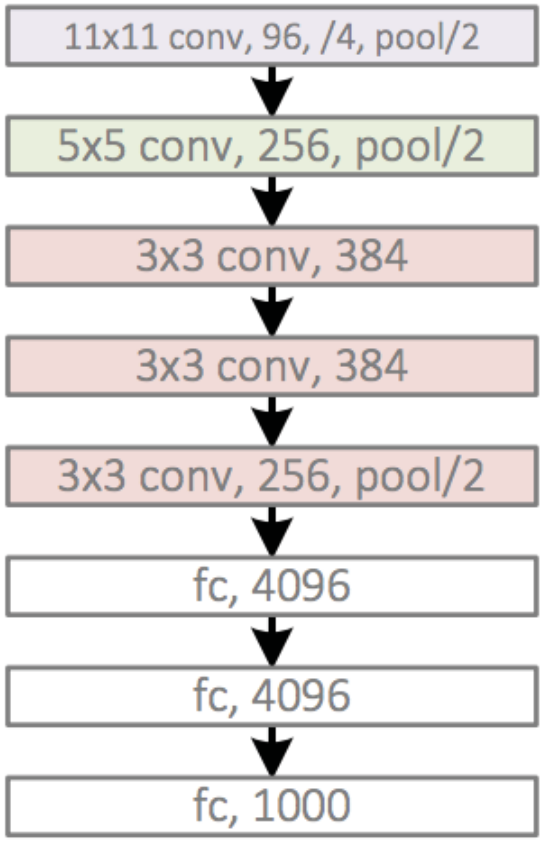
- Computational graphs are a nice way to think about mathematical expressions.
- For example, consider the expression e=(a+b)∗(b+1).
  - There are three operations: two additions and one multiplication.
  - To help us talk about this, let's introduce two intermediary variables, c and d so that every function's output has a variable.
  - Leading to :
    - c=a+b
    - d=b+1
    - e=c∗d



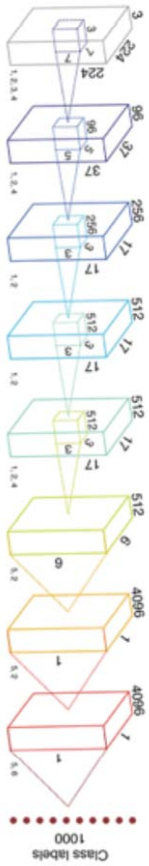
- When one node's value is the input to another node, an arrow goes from one to another.

AlexNet

AlexNet, 8 layers
(ILSVRC 2012)

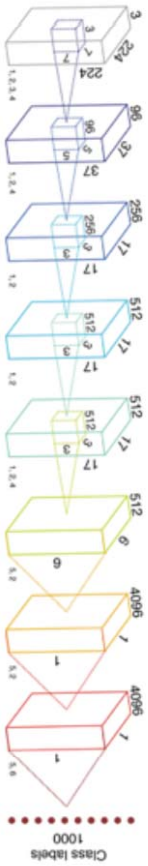


2012



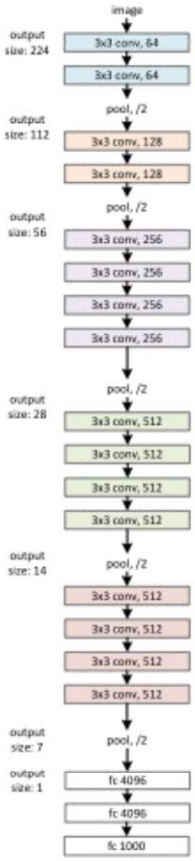
8 layers
15.31% error

2013



9 layers, 2x params
11.74% error

2014

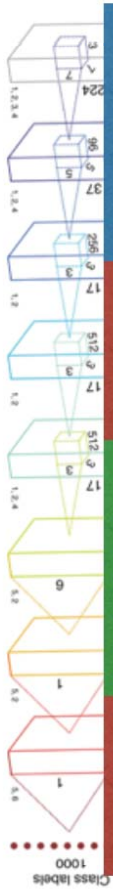


19 layers
7.41% error

2015

?

2012 2013 2014 2015



Is learning **better networks** as easy as stacking **more layers**?

Vanishing / exploding gradients

Normalized initialization & intermediate normalization

Degradation problem

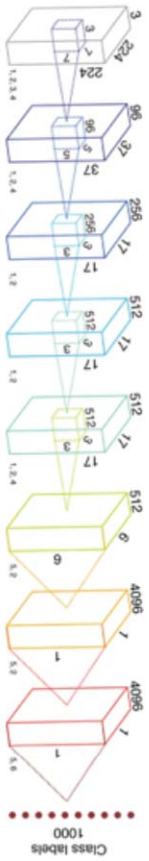
?

8 layers
15.31% error

9 layers, 2x params
11.74% error

19 layers
7.41% error

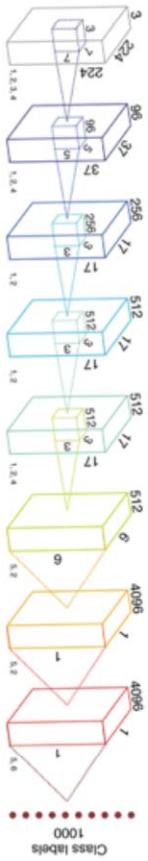
2012



8 layers

15.31% error

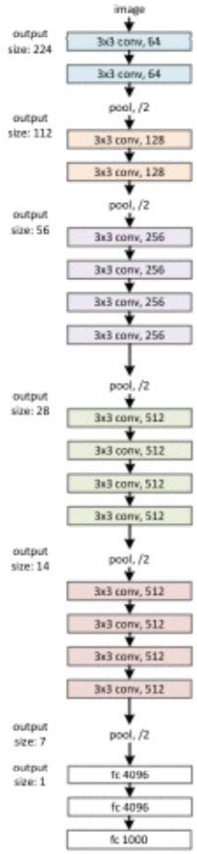
2013



9 layers, 2x params

11.74% error

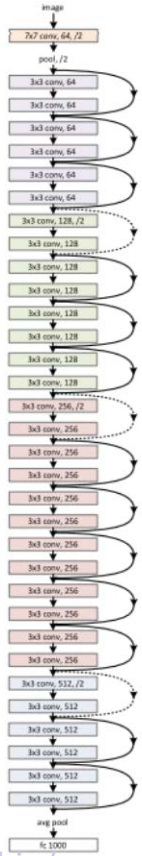
2014



19 layers

7.41% error

2015

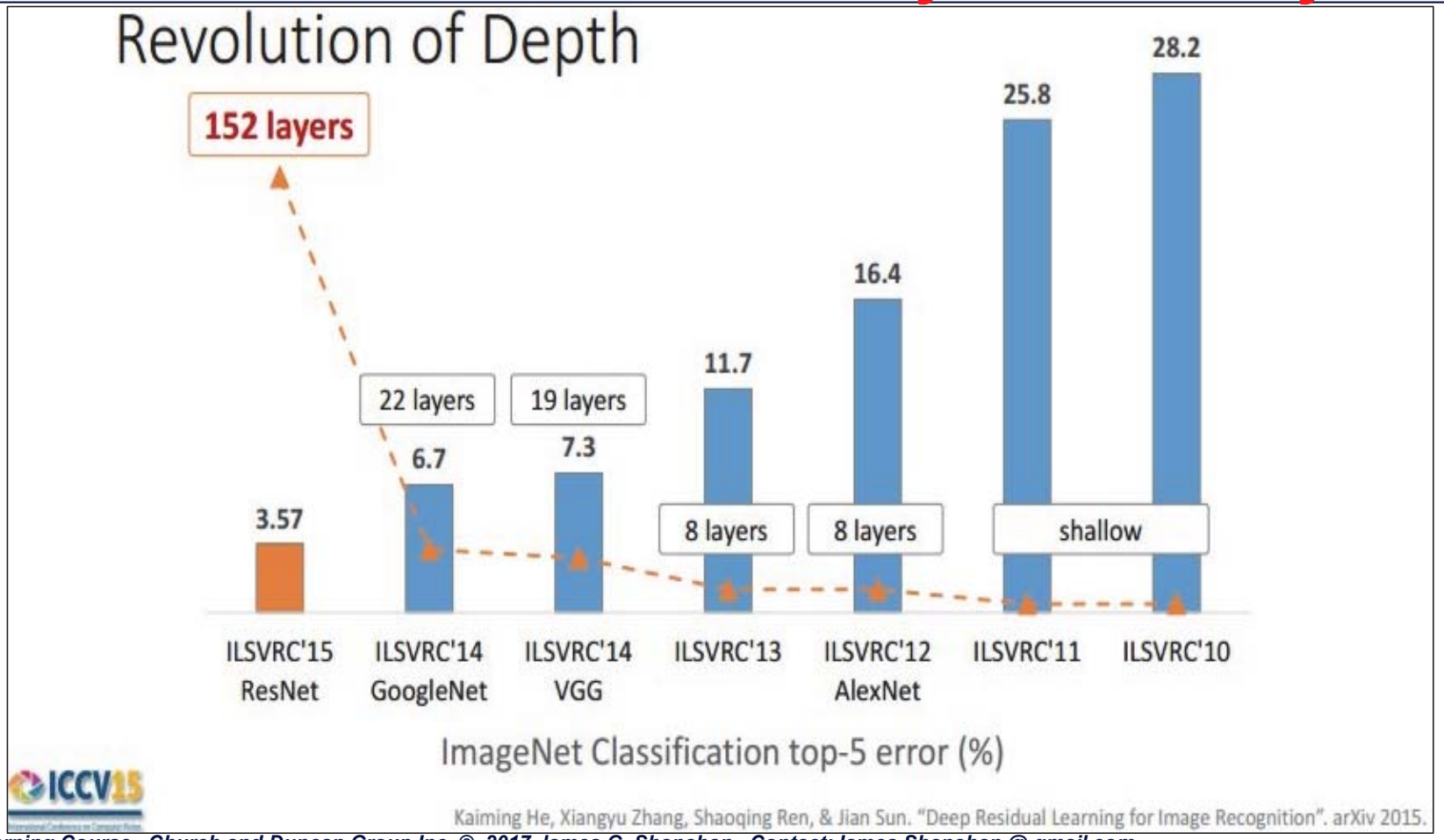


152 layers

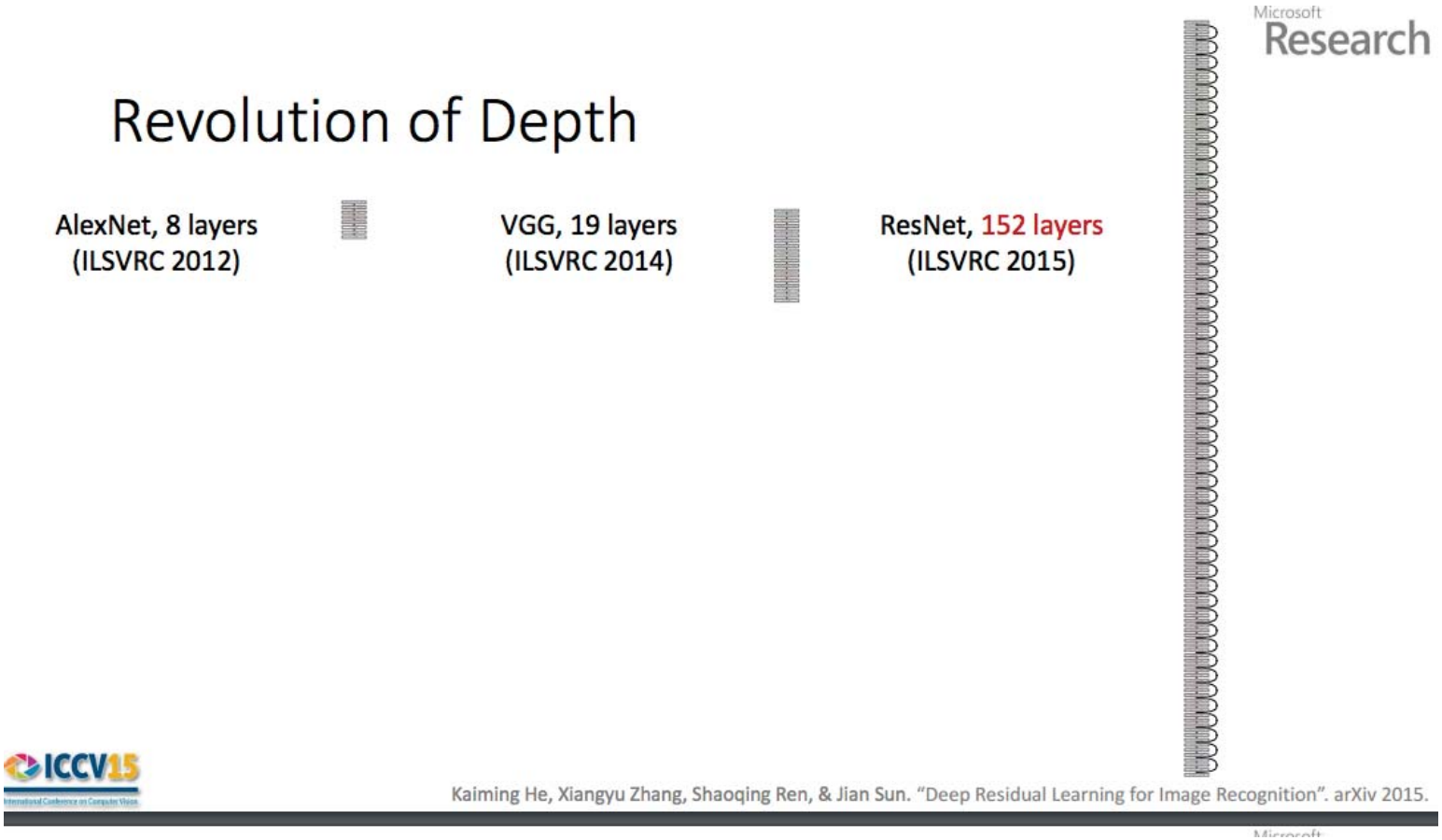
3.57% error

http://www.slideshare.net/iljakuzovkin/paper-overview-deep-residual-learning-for-image-recognition?from_action=save

DNN is Beating human performance (5%)
8Layers → 152 layers



8 → 20 -> 152 layers



no-activation works better (get rid of nonlinearity; gradient flows better)

- The second paper I'd like to highlight is Identity Mappings in Deep Residual Networks, by He et al.
- The original technique only approximated identity connections between layers. I.e., the layers still used nonlinear activations after the identity connections.
- These nonlinearities impede gradient propagation.
- The new paper runs with the identity connection concept.
  - The signal propagates unchanged throughout the entire network. The activation functions between layers are absent. The new technique improves training and test performance.
- For more discussion and more recent work see:
  - https://blog.init.ai/residual-neural-networks-are-an-exciting-area-of-deep-learning-research-acf14f4912e9

Comparisons on CIFAR-10/100

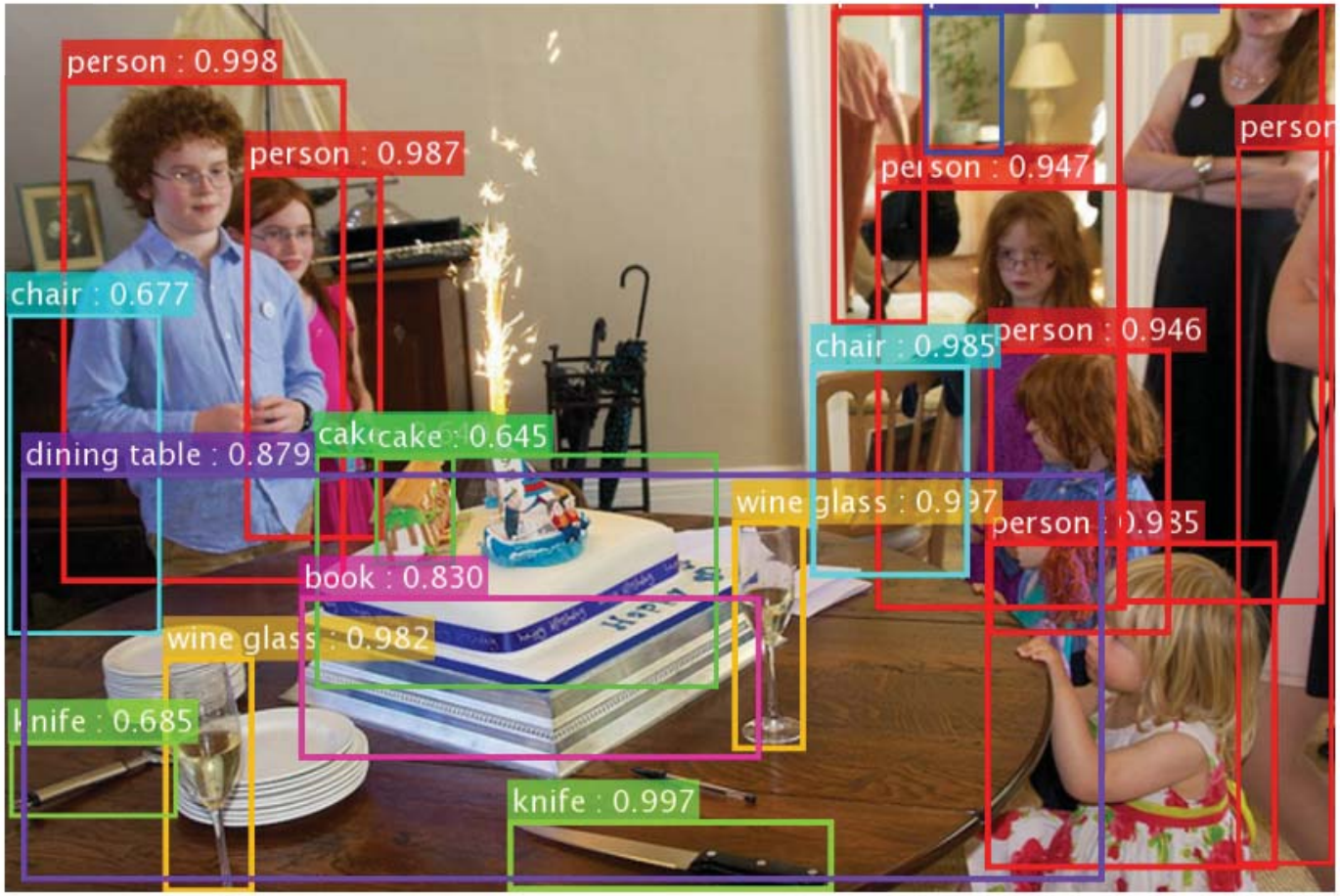
CIFAR-10

method	error (%)
NIN	8.81
DSN	8.22
FitNet	8.39
Highway	7.72
ResNet-110 (1.7M)	6.61
ResNet-1202 (19.4M)	7.93
ResNet-164, pre-activation (1.7M)	5.46
ResNet-1001, pre-activation (10.2M)	**4.92** (4.89±0.14)

CIFAR-100

method	error (%)
NIN	35.68
DSN	34.57
FitNet	35.04
Highway	32.39
ResNet-164 (1.7M)	25.16
ResNet-1001 (10.2M)	27.82
ResNet-164, pre-activation (1.7M)	24.33
ResNet-1001, pre-activation (10.2M)	**22.71** (22.68±0.22)

*all based on moderate augmentation



ResNet's object detection result on COCO

*the original image is from the COCO dataset

ResNets @ ILSVRC & COCO 2015 Competitions

- **1st places in all five main tracks**
  - ImageNet Classification: "*Ultra-deep*" 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

*improvements are relative numbers

Other applications of ReNet

- **Time series CNNs+ ResNet**
  - https://arxiv.org/pdf/1611.06455.pdf
- **Recurrent Residual Learning for Sequence Classification**
  - https://www.aclweb.org/anthology/D16-1093
  - RNNs + residual connections lead to simpler LSTM models
    - In this paper, we explore the possibility of leveraging Residual Networks (ResNet), a powerful structure in constructing extremely deep neural network for image understanding, to improve recurrent neural networks (RNN) for modeling sequential data.
    - We show that for sequence classification tasks, incorporating residual connections into recurrent structures yields similar accuracy to Long Short Term Memory (LSTM) RNN with much fewer model parameters. In addition, we propose two novel models which combine the best of both residual learning and LSTM. Experiments show that the n

ResNet Resources

- **Publications:**
  - [a] *Deep Residual Learning for Image Recognition*
    Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. CVPR 2016.
  - [b] *Identity Mappings in Deep Residual Networks*
    Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Technical report, arXiv 2016.
- **Resources:**
  - tutorial slides
  - slides and video for the talk at ICCV 2015 ImageNet and COCO joint workshop.
  - code/models of 50, 101, and 152-layer ResNets pre-trained on ImageNet.
  - code of 1001-layer ResNet on CIFAR.
  - list of third-party ResNet implementations on ImageNet, CIFAR, MNIST, etc.

~1700s	• The prologue
1940-1970	• Act 1 Tinker: Hack it up
1970-1995	• Act 2 BackProp: theory to the rescue
1995-2007	• Act 3 Layer by layer learning, a medieval pastime
2007-2015	• Act 4 Introspection: better init. and activation functions
2015 -	• Act 5 Express-laning the gradient: Skip Connections, the SoTA frontier (LSTMs, ResNet, Highway Nets, DenseNets)
	• The epilogue

CNN architectures for images: comparison

Architecture	#Parameters	Training time estimates	#layers	Rank	Unique contribution
GoogLeNet	4M	~3-5 days	22	#2	Average Pooling instead of Fully Connected layers at the top of the ConvNet; inception modules with shortcut connections
AlexNet	60M	~1-2 day	8	#4	Network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).
VGGNet	140M	Each batch takes Similar to ResNet; Took 2-3 weeks on four NVIDIA Titan Black GPUs (half VRAM of Titan X)	19	#3	the depth of the network is a critical component for good performance; features an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end
ResNet	~70M	2-3 weeks on 8 GPU	152	#1	It features special *skip connections* and a heavy use of batch normalization. Kaiming's presentation (video, slides),

Dense Networks

ResNet

- **PRO**
  - An advantage of ResNets is that the gradient flows directly through the identity function from later layers to the earlier layers.
- **Con**
  - However, the identity function and the output of H_L are combined by summation, which may impede the information flow in the network.

Densely Connected Convolutional Networks

Gao Huang*
Cornell University
gh349@cornell.edu

Zhuang Liu*
Tsinghua University
liuzhuangthu@gmail.com

Kilian Q. Weinberger
Cornell University
kqw4@cornell.edu

Laurens van der Maaten
Facebook AI Research
lvdmaaten@fb.com

Abstract

Recent work has shown that convolutional networks can be substantially deeper, more accurate, and efficient to train if they contain shorter connections between layers close to the input and those close to the output. In this paper, we embrace this observation and introduce the Dense Convolutional Network (DenseNet), which connects each layer to every other layer in a feed-forward fashion. Whereas traditional convolutional networks with L layers have L connections—one between each layer and its subsequent layer—our network has $\frac{L(L+1)}{2}$ direct connections. For each layer, the feature-maps of all preceding layers are used as inputs, and its own feature-maps are used as inputs into all subsequent layers. DenseNets have several compelling advantages: they alleviate the vanishing-gradient problem, strengthen feature propagation, encourage feature reuse, and substantially reduce the number of parameters. We evaluate our proposed architecture on four highly competitive object recognition benchmark tasks (CIFAR-10, CIFAR-100, SVHN, and ImageNet). DenseNets obtain significant improvements over the state-of-the-art on most of them, whilst requiring less memory and computation to achieve high performance. Code and models are available at https://github.com/liuzhuang13/DenseNet.

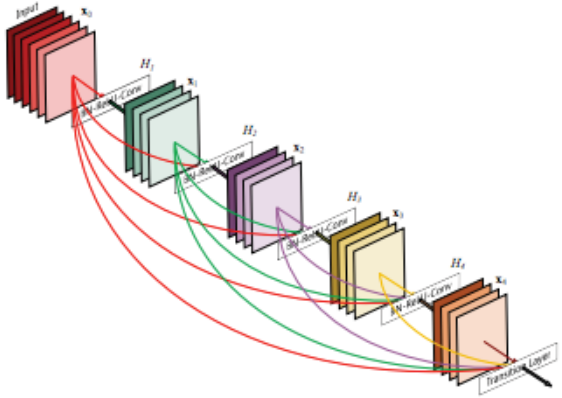


Figure 1. A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

Networks [33] and Residual Networks (ResNets) [11] have surpassed the 100-layer barrier.

As CNNs become increasingly deep, a new research problem emerges: as information about the input or gradient passes through many layers, it can vanish and "wash out" by the time it reaches the end (or beginning) of the network. Many recent publications address this or related problems. ResNets [11] and Highway Networks [33] bypass signal from one layer to the next via identity connec-

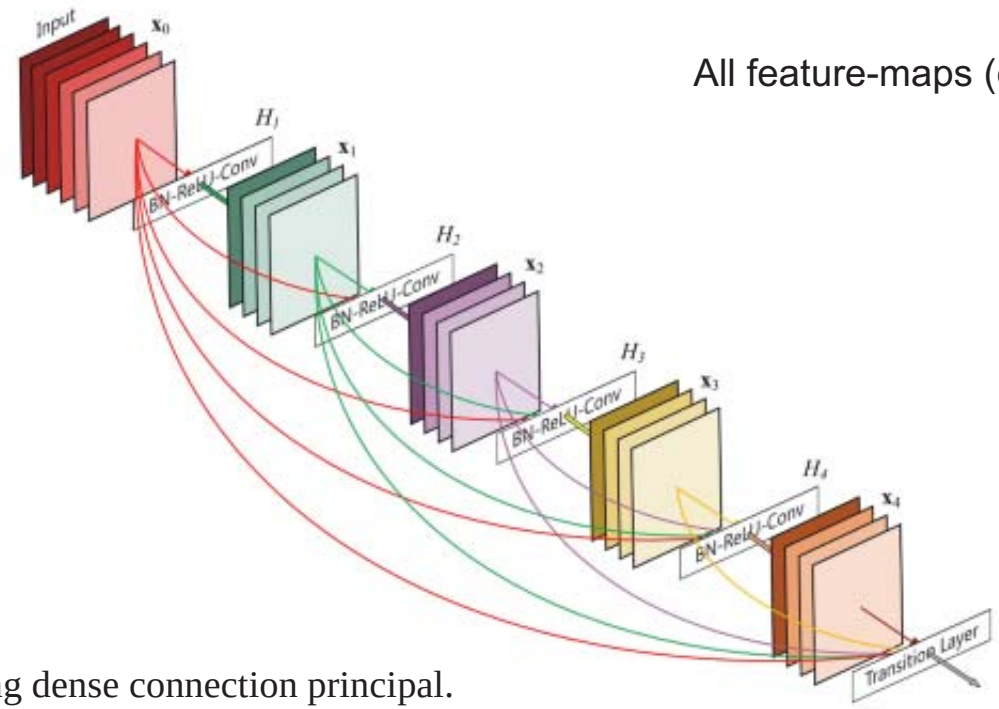
"Densely Connected Convolutional Networks" (CVPR 2017, Oral) by Gao Huang*, Zhuang Liu*, Laurens van der Maaten and Kilian Weinberger

DenseNet similar accuracy as ResNet but ½ the FLOPS

- DenseNet is a network architecture where each layer is directly connected to every other layer in a feed-forward fashion (within each dense block).
- For each layer, the feature maps of all preceding layers are treated as separate inputs whereas its own feature maps are passed on as inputs to all subsequent layers.
- This connectivity pattern yields state-of-the-art accuracies on CIFAR10/100 (with or without data augmentation) and SVHN.
- On the large scale ILSVRC 2012 (ImageNet) dataset, DenseNet achieves a similar accuracy as ResNet, but using less than half the amount of parameters and roughly half the number of FLOPs.

Dense Networks

- **A 5-layer dense block with a growth rate of k = 4 (feature maps). Each layer takes all preceding feature-maps as input.**



All feature-maps (e.g., 4 here serve as input to

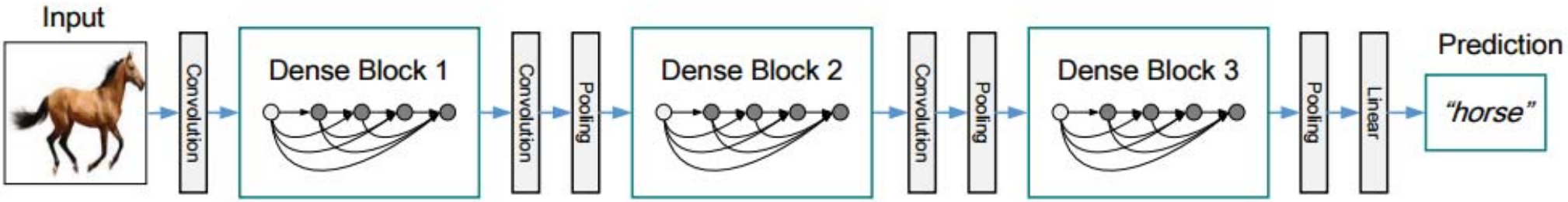
Image demonstrating dense connection principal.
From: https://arxiv.org/abs/1608.06993

Dense connectivity. To further improve the information flow between layers we propose a different connectivity pattern: we introduce direct connections from any layer to all subsequent layers. Figure 1 illustrates the layout of the resulting DenseNet schematically. Consequently, the ℓ^{th} layer receives the feature-maps of all preceding layers, $\mathbf{x}_0, \ldots, \mathbf{x}_{\ell-1}$, as input:

$$\mathbf{x}_\ell = H_\ell([\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{\ell-1}]), \tag{2}$$

where $[\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{\ell-1}]$ refers to the concatenation of the feature-maps produced in layers $0, \ldots, \ell-1$. Because of its dense connectivity we refer to this network architecture as *Dense Convolutional Network (DenseNet)*. For ease of implementation, we concatenate the multiple inputs of $H_\ell(\cdot)$ in eq. (2) into a single tensor.

A Densenet with 3 DenseBlocks



- **A deep DenseNet with three dense blocks.**
- **The layers between two adjacent blocks are referred to as transition layers and change feature map sizes via convolution and pooling.**

Dense Network

- Dense Network, or DenseNet is an architecture takes the insights of the skip connection to the extreme.
- The idea here is that if connecting a skip connection from the previous layer improves performance, why not connect every layer to every other layer? That way there is always a direct route for the information backwards through the network.

Method	Depth	Params	C10	C10+	C100	C100+	SVHN
Network in Network [22]	-	-	10.41	8.81	35.68	-	2.35
All-CNN [31]	-	-	9.08	7.25	-	33.71	-
Deeply Supervised Net [20]	-	-	9.69	7.97	-	34.57	1.92
Highway Network [33]	-	-	-	7.72	-	32.39	-
FractalNet [17]	21	38.6M	10.18	5.22	35.34	23.30	2.01
with Dropout/Drop-path	21	38.6M	7.33	4.60	28.20	23.73	1.87
ResNet [11]	110	1.7M	-	6.61	-	-	-
ResNet (reported by [13])	110	1.7M	13.63	6.41	44.74	27.22	2.01
ResNet with Stochastic Depth [13]	110	1.7M	11.66	5.23	37.80	24.58	1.75
	1202	10.2M	-	4.91	-	-	-
Wide ResNet [41]	16	11.0M	-	4.81	-	22.07	-
	28	36.5M	-	4.17	-	20.50	-
with Dropout	16	2.7M	-	-	-	-	1.64
ResNet (pre-activation) [12]	164	1.7M	11.26*	5.46	35.58*	24.33	-
	1001	10.2M	10.56*	4.62	33.47*	22.71	-
DenseNet ($k = 12$)	40	1.0M	**7.00**	5.24	**27.55**	24.42	1.79
DenseNet ($k = 12$)	100	7.0M	**5.77**	**4.10**	**23.79**	**20.20**	1.67
DenseNet ($k = 24$)	100	27.2M	**5.83**	**3.74**	**23.42**	**19.25**	**1.59**
DenseNet-BC ($k = 12$)	100	0.8M	**5.92**	4.51	**24.15**	22.27	1.76
DenseNet-BC ($k = 24$)	250	15.3M	**5.19**	**3.62**	**19.64**	**17.60**	1.74
DenseNet-BC ($k = 40$)	190	25.6M	-	**3.46**	-	**17.18**	-

Table 2. Error rates (%) on CIFAR and SVHN datasets. L denotes the network depth and k its growth rate. Results that surpass all competing methods are **bold** and the overall best results are **blue**. "+" indicates standard data augmentation (translation and/or mirroring). $*$ indicates results run by ourselves. All the results of DenseNets without data augmentation (C10, C100, SVHN) are obtained using Dropout. DenseNets achieve lower error rates while using fewer parameters than ResNet. Without data augmentation, DenseNet performs better by a large margin.

The Street View House Numbers (SVHN)

Deeper DenseNets work well with k=124-40 feature-maps

Training algo hyperparameters

- All the networks are trained using SGD.
- On CIFAR and SVHN we train using mini-batch size 64 for 300 and 40 epochs, respectively.
- The initial learning rate is set to 0.1, and is divided by 10 at 50% and 75% of the total number of training epochs.
- On ImageNet, we train models for 90 epochs with a mini-batch size of 256.
- The learning rate is set 0.1 initially, and is lowered by a factor of 10 after epoch 30 and epoch 60.
- Because of GPU memory constraints, our largest model (DenseNet-161) is trained with a mini-batch size 128.
- To compensate for the smaller batch size, we train this model for 100 epochs, and divide the learning rate by 10 after epoch 90.

Homework

Skip connections Notebook

- **https://github.com/awjuliani/TF-Tutorials/blob/master/Deep%20Network%20Comparison.ipynb**

Deep Neural Network Comparison

```
In [ ]: #Load necessary libraries
import tensorflow as tf
import numpy as np
import tensorflow.contrib.slim as slim
import input_data
import matplotlib.pyplot as plt
%matplotlib inline
```

Load CIFAR Dataset

To obtain the CIFAR10 dataset, go here: https://www.cs.toronto.edu/~kriz/cifar.html

The training data is stored in 5 separate files, and we will alternate between them during training.

```
In [ ]: def unpickle(file):
    import cPickle
    fo = open(file, 'rb')
    dict = cPickle.load(fo)
    fo.close()
    return dict
```

```
In [ ]: currentCifar = 1
cifar = unpickle('./cifar10/data_batch_1')
cifarT = unpickle('./cifar10/test_batch')
```

```
In [ ]: total_layers = 25 #Specify how deep we want our network
units_between_stride = total_layers / 5
```

RegularNet

A Deep Neural Network composed exclusively of regular and strided convolutional layers. While this architecture works well for relatively shallow networks, it becomes increasingly more difficult to train as the network depth increases.

```
In [ ]: tf.reset_default_graph()

input_layer = tf.placeholder(shape=[None,32,32,3],dtype=tf.float32,name='input')
label_layer = tf.placeholder(shape=[None],dtype=tf.int32)
label_oh = slim.layers.one_hot_encoding(label_layer,10)

layer1 = slim.conv2d(input_layer,64,[3,3],normalizer_fn=slim.batch_norm,scope='conv_'+str(0))
for i in range(5):
    for j in range(units_between_stride):
        layer1 = slim.conv2d(layer1,64,[3,3],normalizer_fn=slim.batch_norm,scope='conv_'+str((j+1)
 + (i*units_between_stride)))
    layer1 = slim.conv2d(layer1,64,[3,3],stride=[2,2],normalizer_fn=slim.batch_norm,scope='conv_s
_'+str(i))

top = slim.conv2d(layer1,10,[3,3],normalizer_fn=slim.batch_norm,activation_fn=None,scope='conv_to
p')

output = slim.layers.softmax(slim.layers.flatten(top))

loss = tf.reduce_mean(-tf.reduce_sum(label_oh * tf.log(output) + 1e-10, axis=[1]))
trainer = tf.train.AdamOptimizer(learning_rate=0.001)
update = trainer.minimize(loss)
```

ResNet

An implementation of a Residual Network as described in Identity Mappings in Deep Residual Networks.

Homework using prebuilt algos in TF

- **Run experiments using implementations of Skip connection networks on CIFAR10 and report results on different network depths**
- **With little parameter tuning you should be able to get them to perform above 90% accuracy on a test set after a little work.**
  - The full code for training each of these models, and comparing them to a traditional networks is available here. I hope this walkthrough has been a helpful introduction to the world of really deep neural networks!
  - https://chatbotslife.com/resnets-highwaynets-and-densenets-oh-my-9bb15918ee32

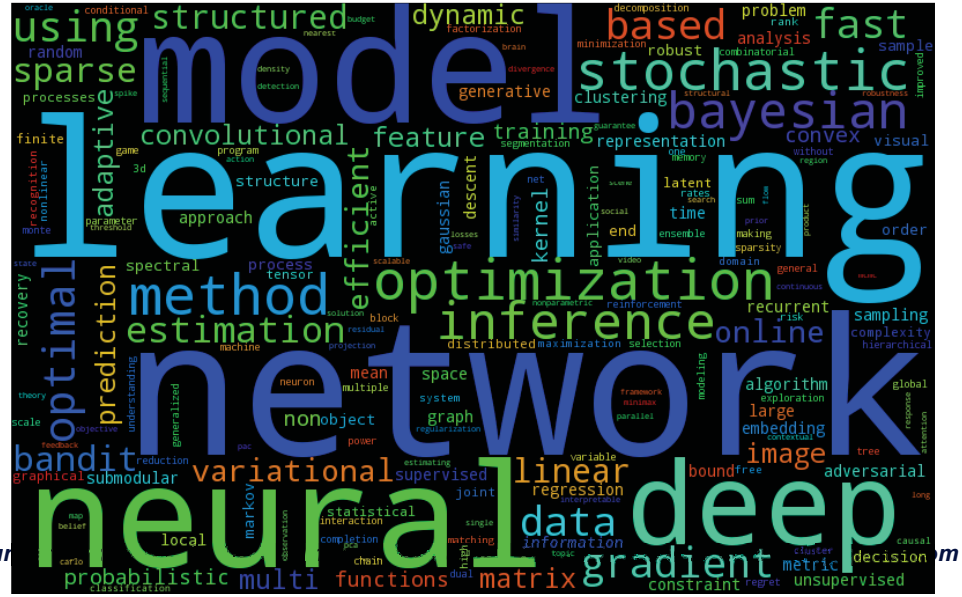
▽ the (misunderstood) gradient: Outline

- ~1700s • The prologue
- 1940-1970 • Act 1 Tinker: Hack it up
- 1970-1995 • Act 2 BackProp: theory to the rescue
- 1995-2006 • Act 3 Layer by layer learning, a medieval pastime
- 2006-2015 • Act 4 Introspection: better init. and activation functions
- 2015 - • Act 5 Express-laning the gradient: Skip Connections, the SoTA frontier (LSTMs, ResNet, Highway Nets, DenseNets)
- • The epilogue

Credits

Cast

- The network as itself
- ▽ (Nabla), the gradient as itself
- The neuron (and its nonlinear incarnations) as itself
- Autodiff (the relief cavalry)
- Supporting cast of concepts (e.g., loss function, SGD, etc.)



Executive producers

Donald Hebb
(1904–1985)
Developed a theory explaining how networks of neurons in the brain enable learning.

Marvin Minsky
(born 1927)
Developed one of the first artificial neural networks, but became a proponent of the symbolic approach to AI.

John McCarthy
(1927–2011)
Helped found the discipline of artificial intelligence; championed the use of mathematical logic in AI.

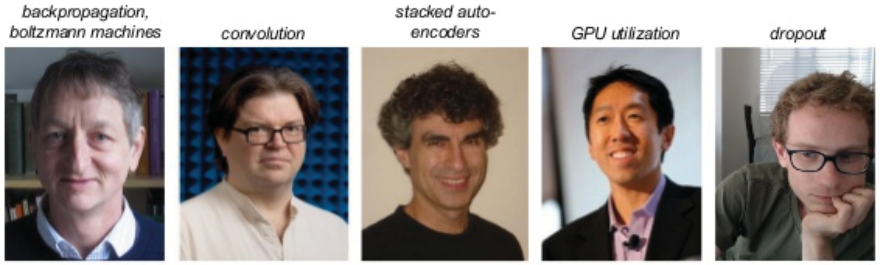
David Rumelhart
(1942–2011)
Developed simulations showing how neural processing could enable perception, and devised much more sophisticated neural networks.

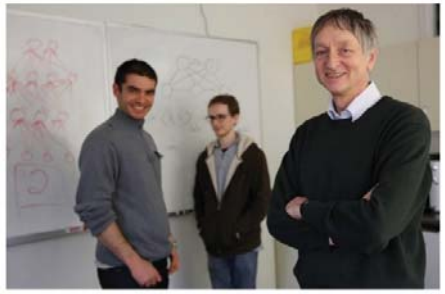
Allen Newell
(1927–1992)
and **Herbert Simon**
(1916–2001)
Created the Logic Theorist, a program designed to mimic the logic skills of a human being.

James McClelland
(born 1948)
Founded the connectionist movement with Rumelhart, and cowrote the seminal work *Parallel Distributed Processing.*

Frank Rosenblatt
(1928–1971)
Created an electronic device called the Perceptron that simulated simple neural learning.

Geoffrey Hinton
(born 1947)
Developed techniques that allowed many more layers to be used in neural networks, creating the foundation for deep learning.





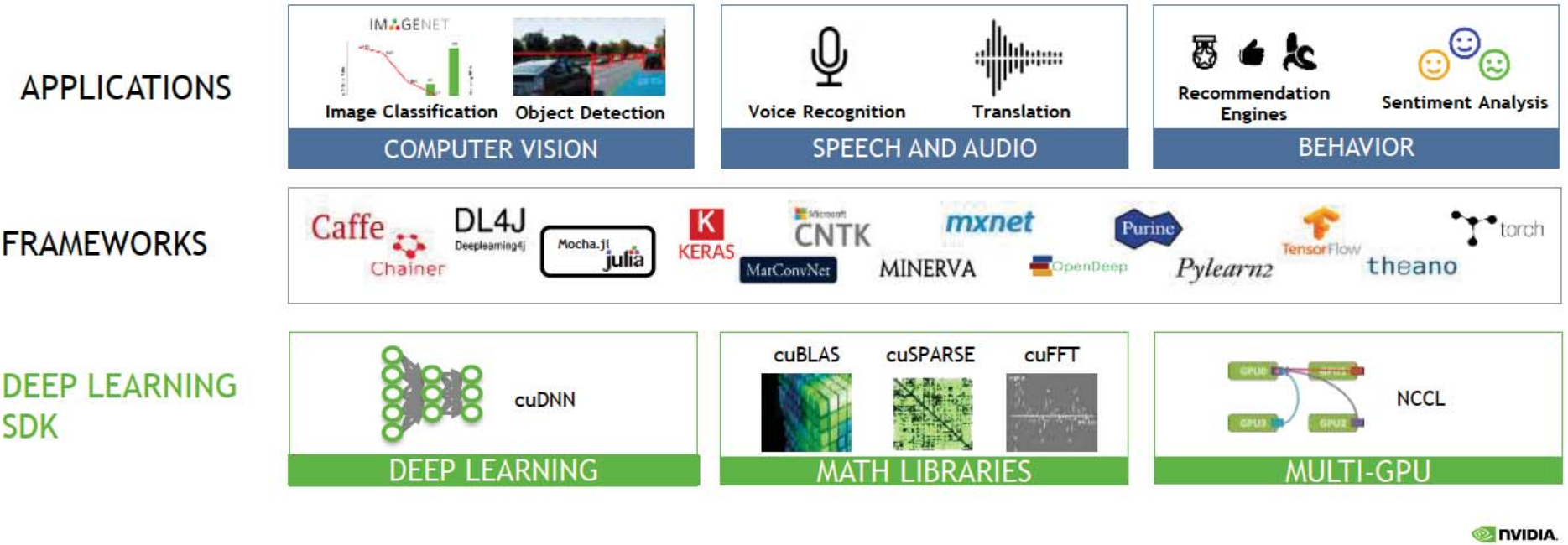
Ilya Sutskever, Alex Krizhevsky, Geoffrey Hinton

https://www.wired.com/2013/03/google_hinton/

https://www.pinterest.com/pin/526921225134807091

Deep Learning supporting EcoSystem

- **High Performance GPU-Acceleration for Deep Learning**



End of lecture