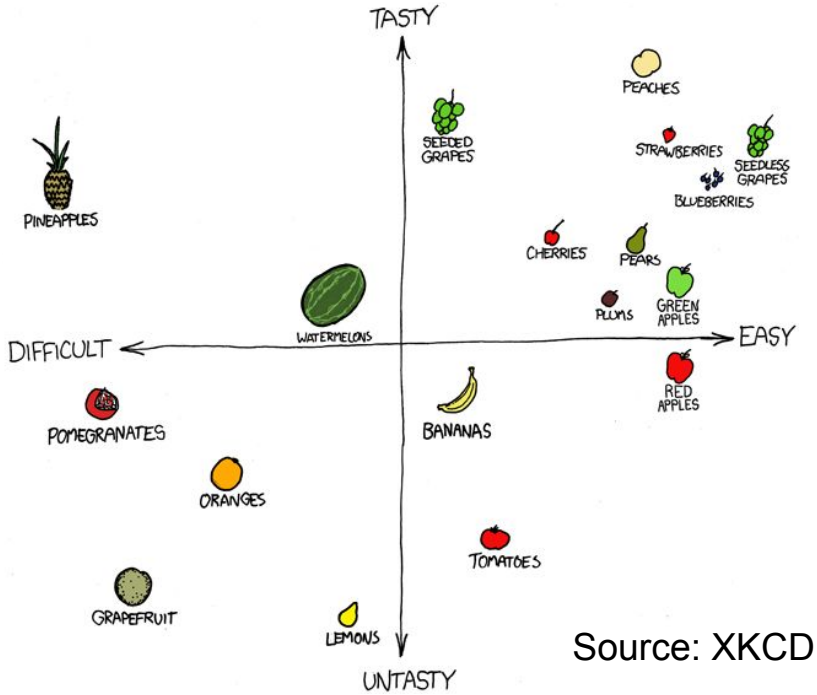
Neural Networks: From Shallow to Deep

Mehrdad Yazdani
May 26, 2017



Source: XKCD

What's all the fuss about? ML in popular media

IBM Deep Blue
1997

IBM Watson 2011

Google DeepMind
AlphaGo 2015

Games (and movies) capture the popular imagination, but the majority of ML research is not addressing games...

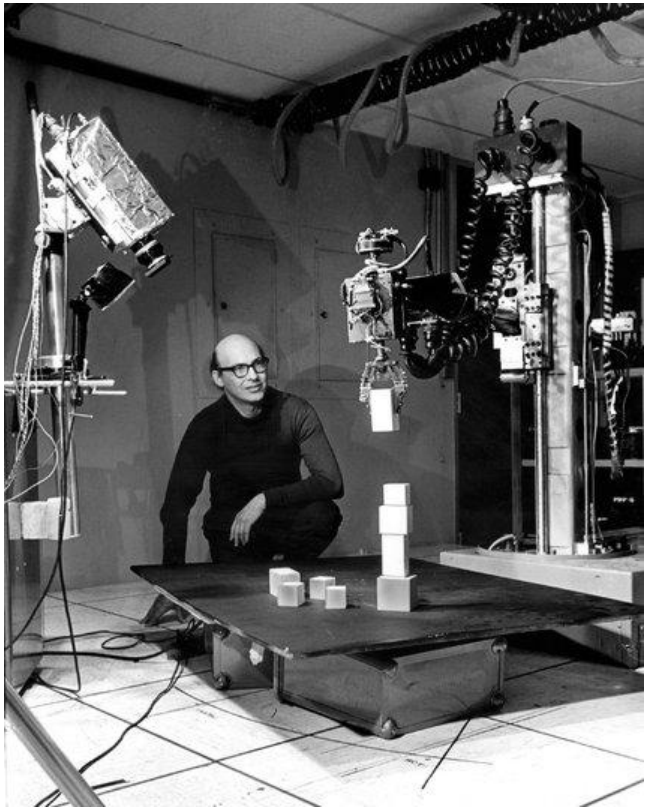
What's all the fuss about?

- Humans solve certain problems easily that machines find difficult

- Can we write algorithms that narrow the human-machine-gap?
  - Machine learning is a collection of methods that attempt to narrow this gap.

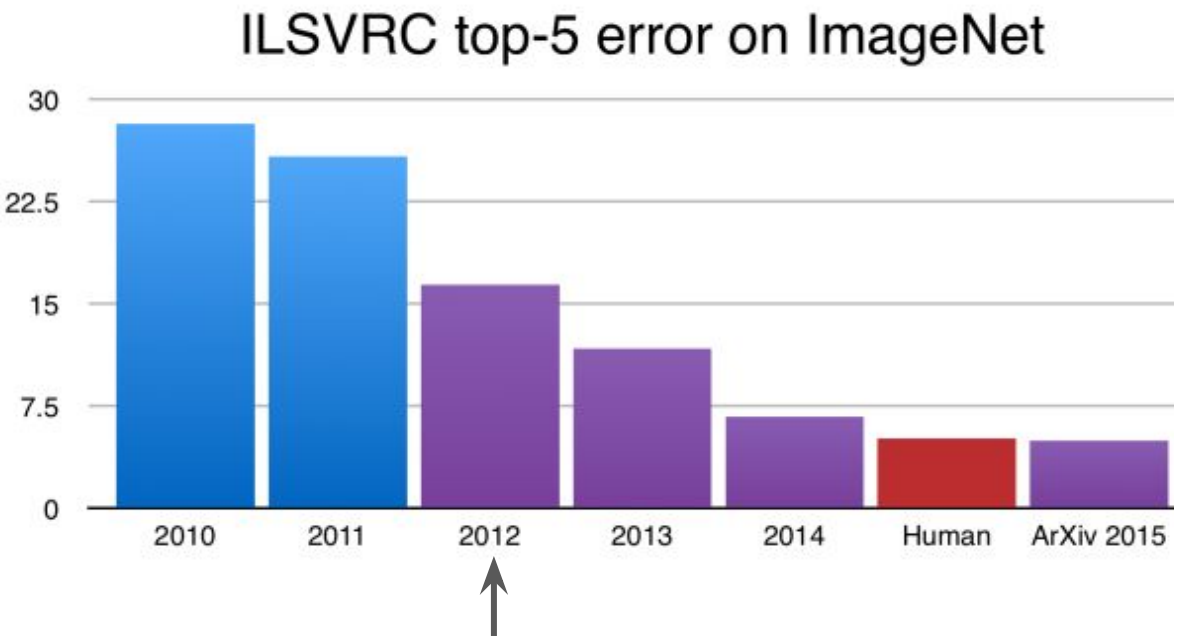
A nontrivial problem

In 1966, Marvin Minsky at MIT asked his undergraduate student Gerald Jay Sussman to "spend the summer linking a camera to a computer and getting the computer to describe what it saw". We now know that the problem is slightly more difficult than that. (Szeliski 2009, Computer Vision)



Impressive Result #1: ImageNet competition

- Object classification challenge
- Over 1 million images with 1,000 categories
  - Many breeds of cats and dogs

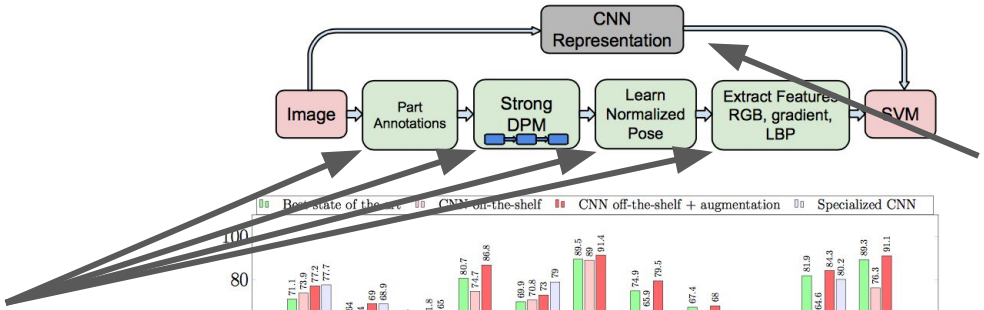


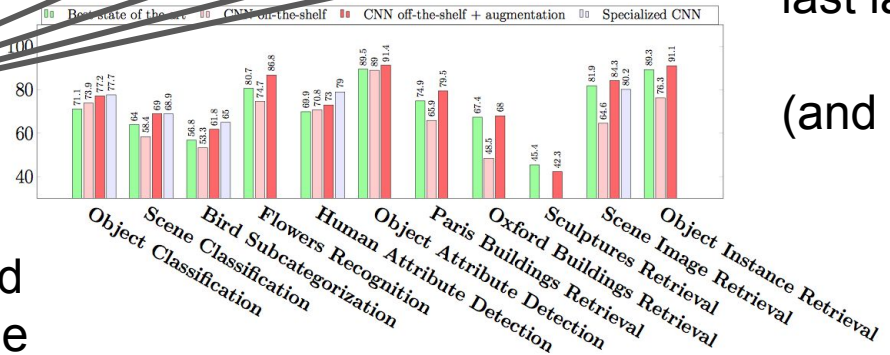
Impressive Result #1: ImageNet competition

- ImageNet object classification challenge “solved”:
  - Suggests that if we have **large** and **labelled** dataset and enough computing power, then we can solve supervised learning problems

- Skeptic response: the “*me, me, me*” problem
  - Getting human level performance for one dataset is interesting, but how does it relate to ***my*** problem? I don’t care about ImageNet! What about ***my*** data?

Impressive Result #2: “transfer learning”





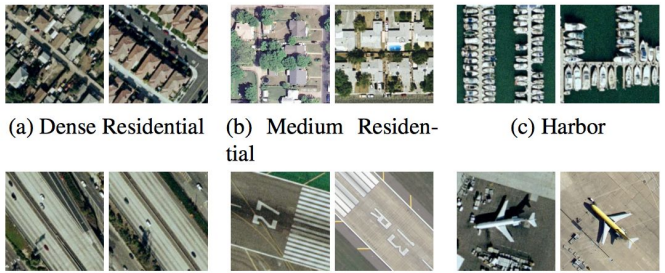
Download a neural network that beats ImageNet from the web and chop off the last layer

(and take this class)

Traditional methods: Requires background in computer vision and lots of time to fine-tune each block

https://arxiv.org/abs/1403.6382
https://arxiv.org/abs/1411.1792

Impressive Result #2: “transfer learning”



(a) Dense Residential (b) Medium Residential (c) Harbor

(d) Freeway (e) Runway (f) Airplane

Figure 1: Examples of the UCMerced Land Use Dataset.

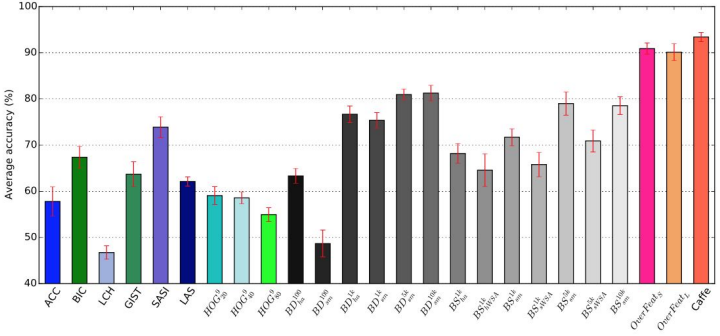


Figure 3: Average accuracy of the descriptors for the UCMerced Land-use Dataset. ConvNets achieve the highest accuracy rates.

http://www.cv-foundation.org/openaccess/content_cvpr_workshops_2015/W13/papers/Penatti_Do_Deep_Features_2015_CVPR_paper.pdf

Feature Sets	Average accuracy
f_{11}	94.8%
f_8, f_{11}	96.6%
f_8, f_{11}, f_2	96.9%
f_8, f_{11}, f_2, f_6	**97.1%**

https://www.academia.edu/30226274/Determining_Feature_Extractors_for_Unsupervised_Learning_on_Satellite_Images

We can use a network that has only seen cats and dogs for completely different image domains!

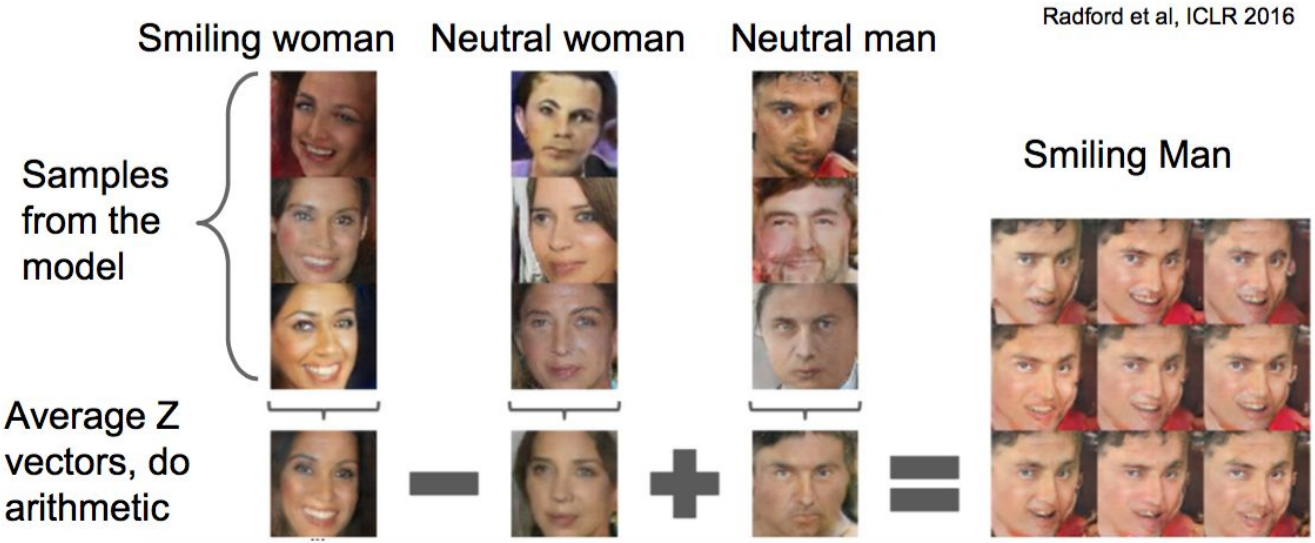
Impressive Result #3: unsupervised learning

- Real world data has an unknown probability density. Can we estimate what this probability density is?

- **Generative Adversarial Networks**: biggest Neural Network innovation in the last 20 years?

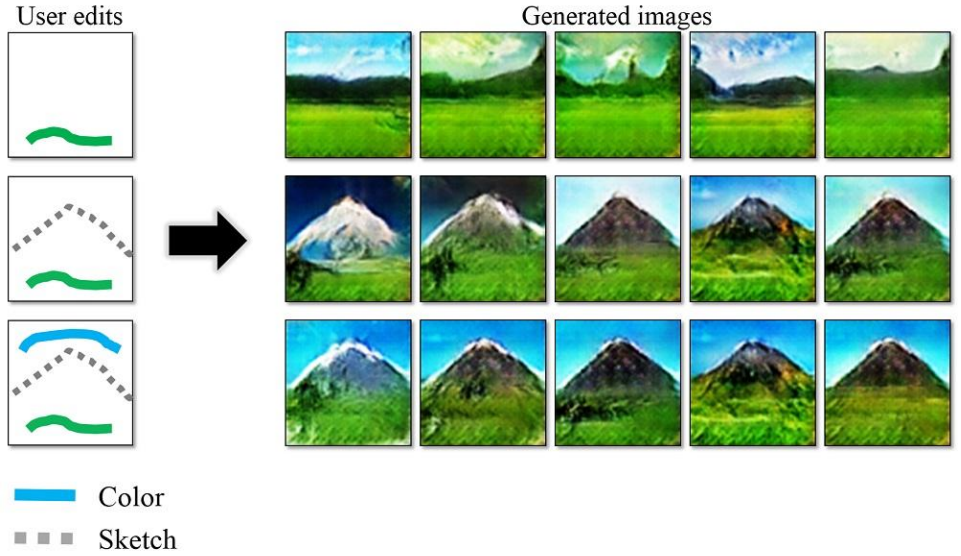
Impressive Result #3: unsupervised learning

Generative Adversarial Networks: biggest Neural Network innovation in the last 20 years?



Impressive Result #3: unsupervised learning

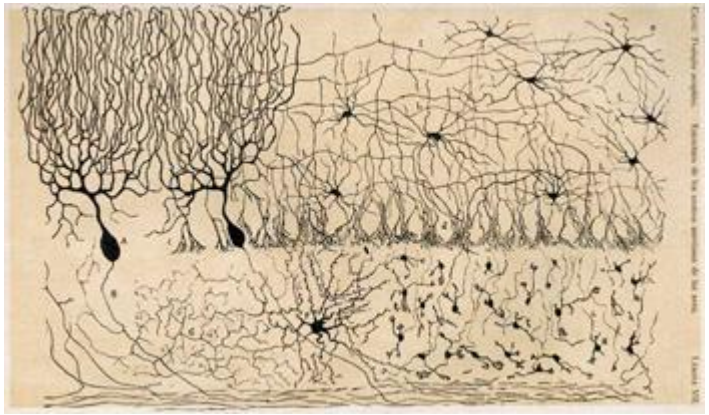
Generative Adversarial Networks: biggest Neural Network innovation in the last 20 years?



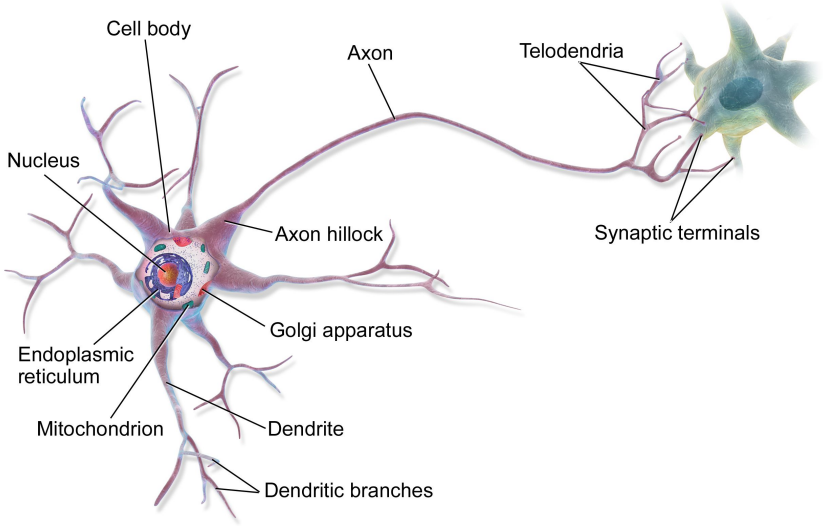
Why now?

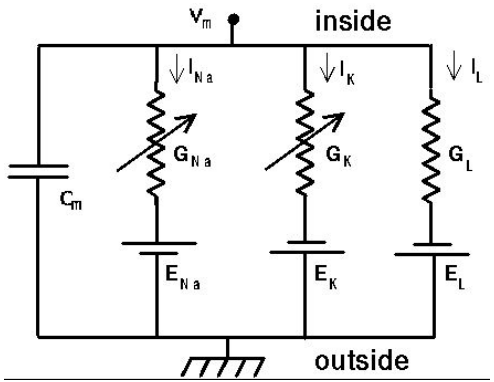
- Massive data amongst vast domains: WWW, IoT, biology, and any other industry that uses computers
- Advances in Hardware: GPUs
- Open source software

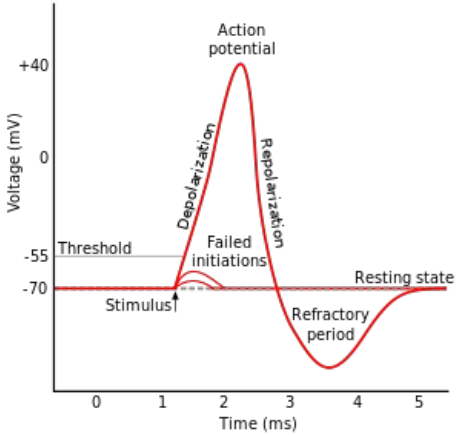
The “Neuron Doctrine”



Ramón y Cajal's drawing of the cells of the chick cerebellum, from *Estructura de los centros nerviosos de las aves*, Madrid, 1905

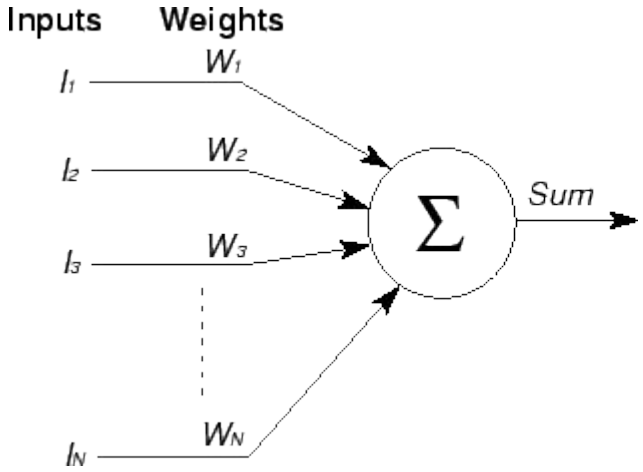






Artificial Neural Networks

- Mcculloch-Pitts Artificial Neuron model: First artificial neuronal units



- Inputs binary, output is binary
- Showed a network of such units can implement any boolean function
- Exercise: how can you create an OR gate? XOR?
- What is the big missing piece?

“Machine Learning” the black box

$$f(\mathbf{x}) \rightarrow \mathbf{y}$$

- “Learn” the functional relationship between the input and output data
- Define a procedure that finds a function that is close to outputs

“Learning”: posed as an optimization problem

Find a function that minimize the expected loss over all data

$$\underset{f}{\operatorname{argmin}} \int p(\mathbf{x},\mathbf{y}) L(f(\mathbf{x}), \mathbf{y}) d\mathbf{x} d\mathbf{y}$$

$$L(f(\mathbf{x}), \mathbf{y}) = \begin{cases} 0 \text{ if } f(\mathbf{x}) = \mathbf{y} \\ c \text{ else, } c > 0 \end{cases}$$

Loss function the engineer selects

Cost for making a mistake

“Learning” in real life

- In practice, we never have $p(\mathbf{x}, \mathbf{y})$

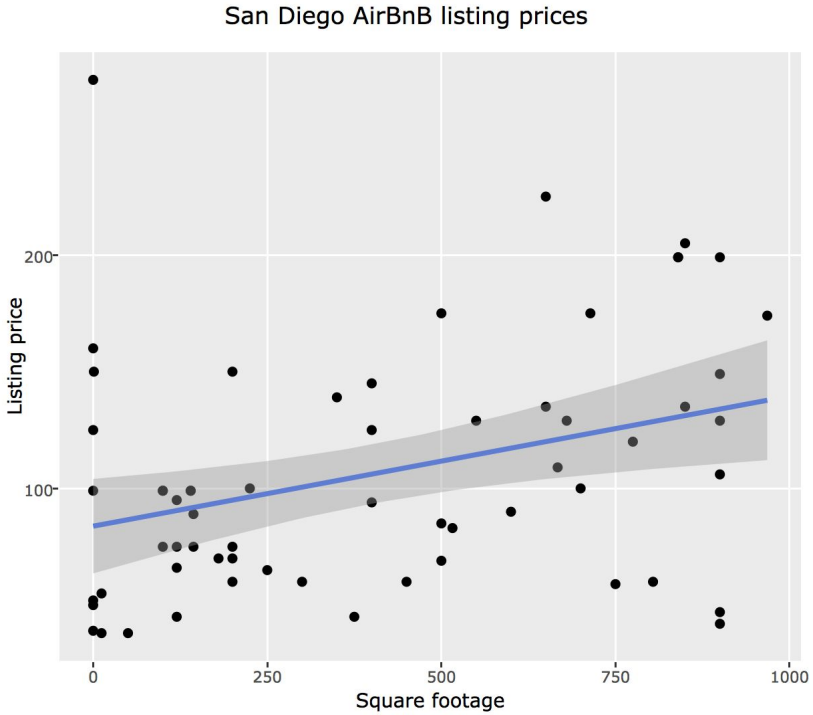
- We only have a sample represented by our dataset: $\{(\mathbf{x}_i, \mathbf{y}_i) | i = 1, \ldots, N\}$

$$\underset{f}{\operatorname{argmin}} \sum_{i=1}^{N} L(f(\mathbf{x}_i), \mathbf{y}_i)$$

Regression

- Predict a numeric value

- Examples:
  - What should be the value of a house?
  - What should be the credit score for a given loan applicant?
  - How long will I live?

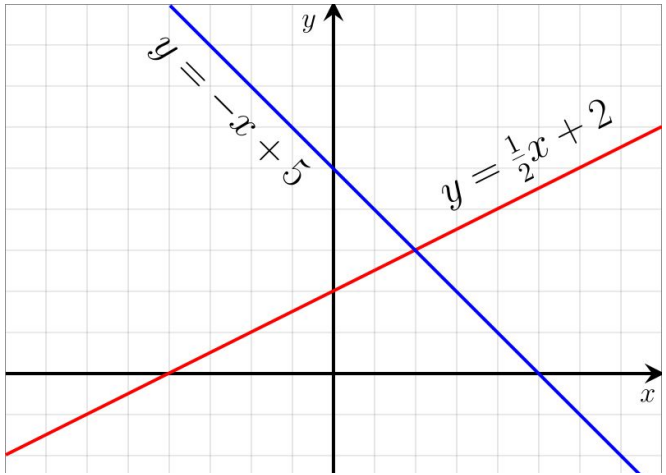


Linear regression

Special case (that we can graph in 2D): $y = mx + b$

slope

y-intercept



General case:

$$y = w_0 + w_1 x_1 + \ldots + w_n x_n$$

Weights: parameters we need to find to model *y*

Output

Inputs: features we are trying to model output *y* with

Linear models need not be linear in features

- A linear model can have nonlinear features, but the model must be a linear function of the weights

$$y = f_{w_0,w_1}(x) = w_0 + w_1 x$$

Both of these functions are *linear* in weights

$$y = f_{w_0,w_1,w_2}(x) = w_0 + w_1 x + w_2 x^2$$

Example of Feature Engineering!

Linear models need not be linear in features

- Sensible to design features that are simple polynomials, exponentials, logarithmic or interactive

$$y = f_{\mathbf{w}}(x_1, x_2) = w_0 + w_1 x_1 + w_2 x_2 + w_3 \log(x_1) + w_4 x_1 x_2$$

- Such nonlinear features seem to appear in nature frequently, hence they are often used in scientific/social studies. For example, Body Mass Index (BMI)

Ordinary Linear Least Squares:

- We collect N (x, y) data pairs.

- We assume that their relationship is linear and that the uncertainty is IID and Normally distributed

$$y_i \approx w_0 + w_1 x_i + \epsilon_i \qquad i = 1, \ldots, N$$
$$\epsilon_i \sim N(0, \sigma)$$

Ordinary Least Squares: Making a lot of assumptions

$$y_i \approx w_0 + w_1 x_i + \epsilon_i \quad \epsilon_i \sim N(0, \sigma) \qquad i = 1, \ldots, N$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \\ 1 & x_N \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_N \end{bmatrix}$$

$$\mathbf{y} \approx \mathbf{X}\mathbf{w} + \epsilon$$

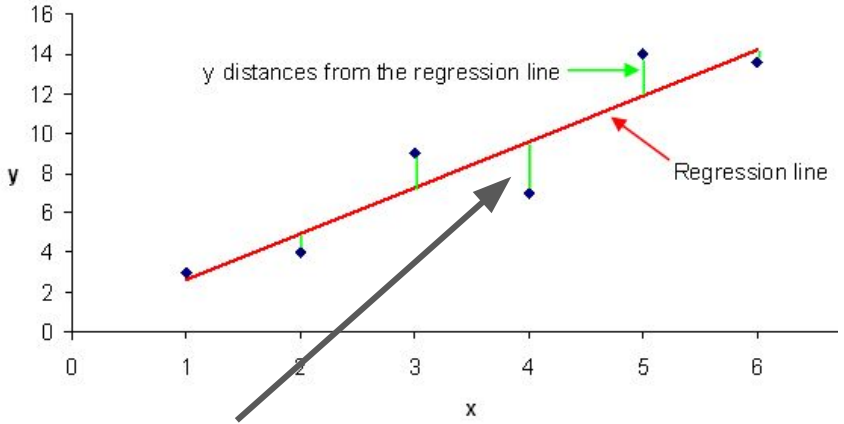
General case: beyond 1 feature

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \approx \begin{bmatrix} 1 & \mathbf{x}_1^T \\ 1 & \mathbf{x}_2^T \\ \vdots & \vdots \\ 1 & \mathbf{x}_N^T \end{bmatrix} \mathbf{w} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_N \end{bmatrix}}$$

$$\mathbf{y} \approx \mathbf{X}\mathbf{w} + \epsilon$$

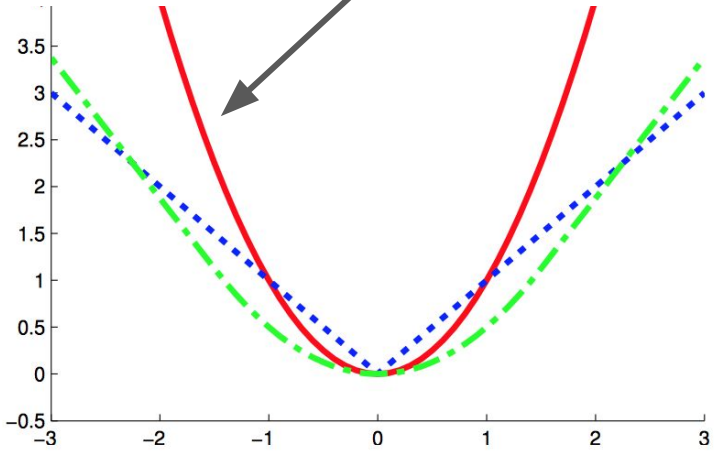
What loss function do we pick?

$$\underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} L(f_{\mathbf{w}}(\mathbf{x}_i), y_i)$$



Residuals: $r_i = |y_i - f_{\mathbf{w}}(\mathbf{x}_i)|$

Selecting a loss function dictates how we treat residuals



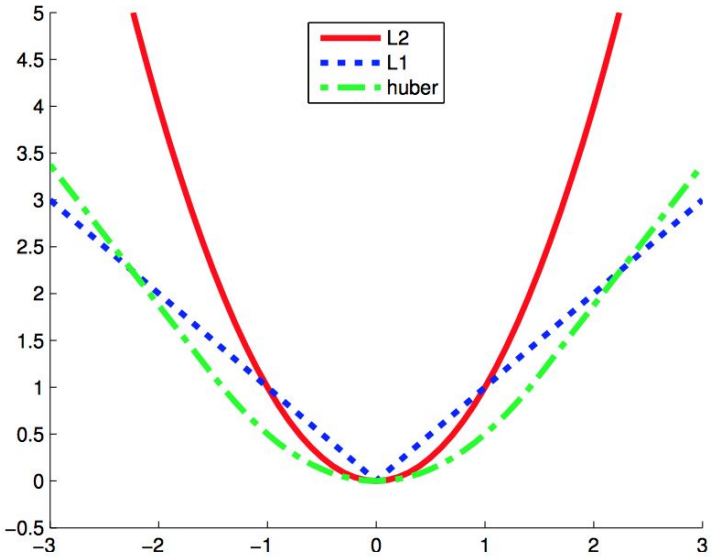
$$L(r_i) = \begin{cases} 0, & \text{if } r_i = 0 \\ c_i, & \text{else}, c_i > 0 \end{cases}$$

Ordinary Least squares

$$\mathbf{y} \approx \mathbf{X}\mathbf{w} + \epsilon$$

$$r_i = |y_i - f_{\mathbf{w}}(\mathbf{x}_i)|$$

$$\underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} r_i^2$$



It turns out that selecting this loss function is the “optimal” one in the case that the uncertainty is modeling IID Gaussian

Terminology of least squares

Residual for example *i* → $r_i = |y_i - f_{\mathbf{w}}(\mathbf{x}_i)|$

Loss for example *i* → $L(r_i) = r_i^2$

Find the set of weights that minimizes the total loss (over training data of course!)

$$\underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} r_i^2 = \underset{\mathbf{w}}{\operatorname{argmin}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$$

Using matrix manipulation can write more compactly

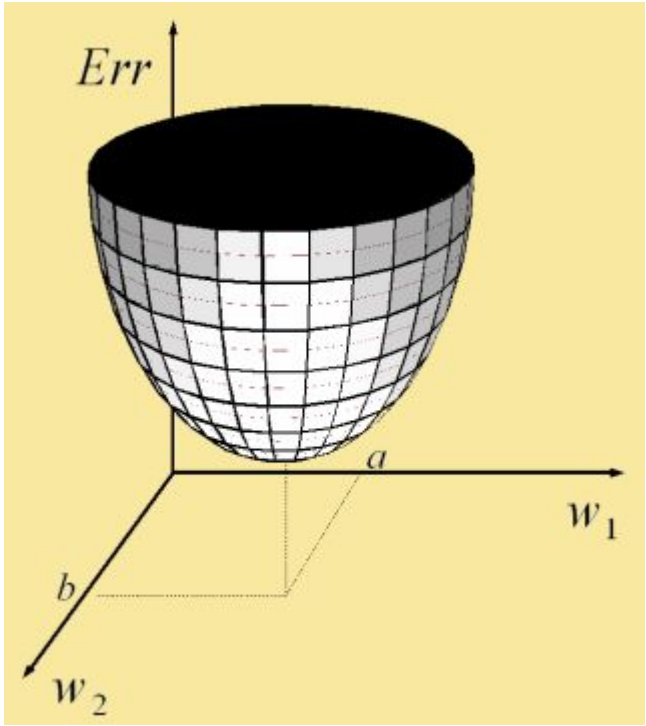
Optimal solution: first derivative

$$\underset{\mathbf{w}}{\operatorname{argmin}}(\mathbf{Xw}-\mathbf{y})^T(\mathbf{Xw}-\mathbf{y})$$

$$\frac{\partial}{\partial \mathbf{w}}(\mathbf{Xw}-\mathbf{y})^T(\mathbf{Xw}-\mathbf{y}) \rightarrow \mathbf{0}$$

Normal Equations: $(\mathbf{Xw}-\mathbf{y})^T\mathbf{X} = \mathbf{0}$

Large literature on fast methods for solving Normal Equations efficiently



The normal equations solution

$$\underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} r_i^2 = \underset{\mathbf{w}}{\operatorname{argmin}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$$

$$(\mathbf{X}\mathbf{w} - \mathbf{y})^T \mathbf{X} = \mathbf{0}$$

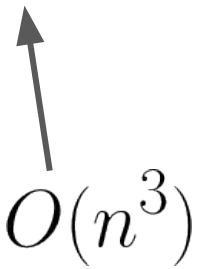
$$\mathbf{w}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Probabilistic perspective: really only optimal for the case when the uncertainty is assumed to be Gaussian

If inverse does not exist, use pseudo-inverse

Challenges with the Normal Equations

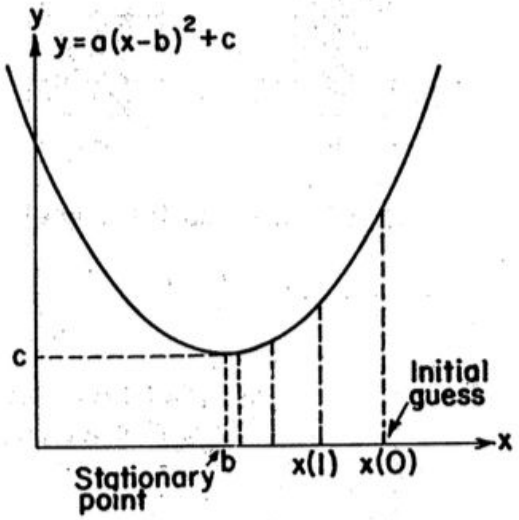
$$\mathbf{w}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$



- In “classical” statistics there are not that many features to consider, so this is not a major issue
- In modern machine learning having, the number of features can range from thousands to millions

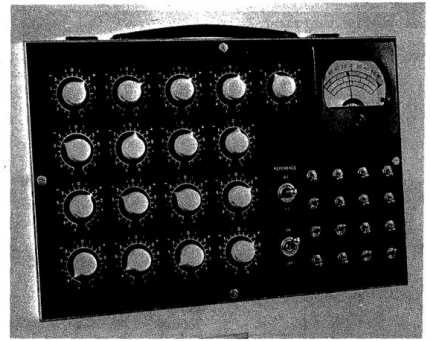
Widrow's ADALINE (LMS filter)

1. Start with an initial guess of weights
2. Compute the gradient at that location.
3. "Step" in the opposite direction



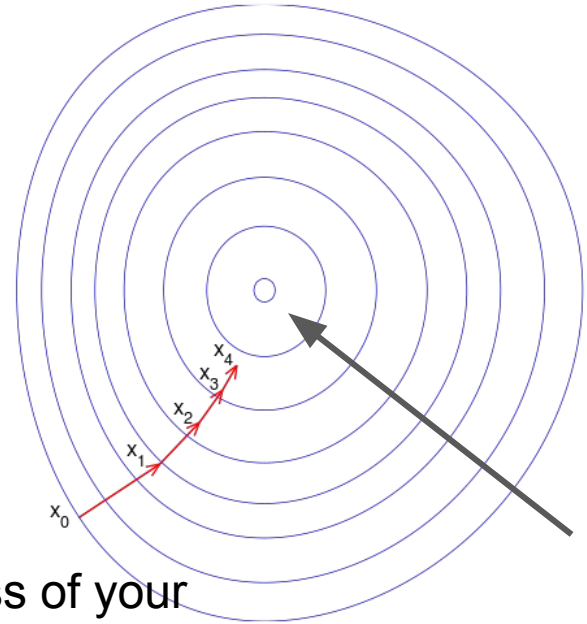
ADAPTIVE SWITCHING CIRCUITS

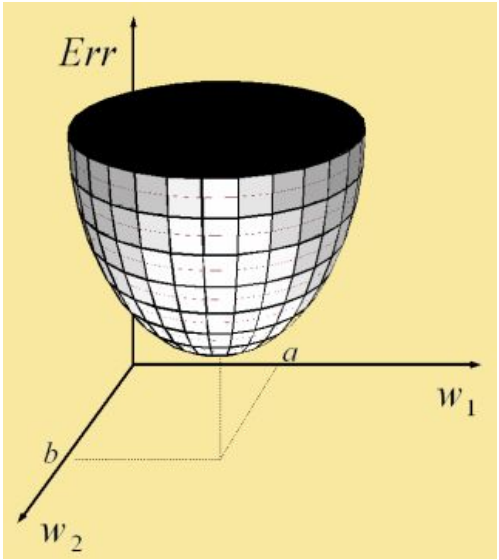
Bernard Widrow
and
Marcian E. Hoff



Adjust weights with potentiometers

Gradient Descent on level surfaces

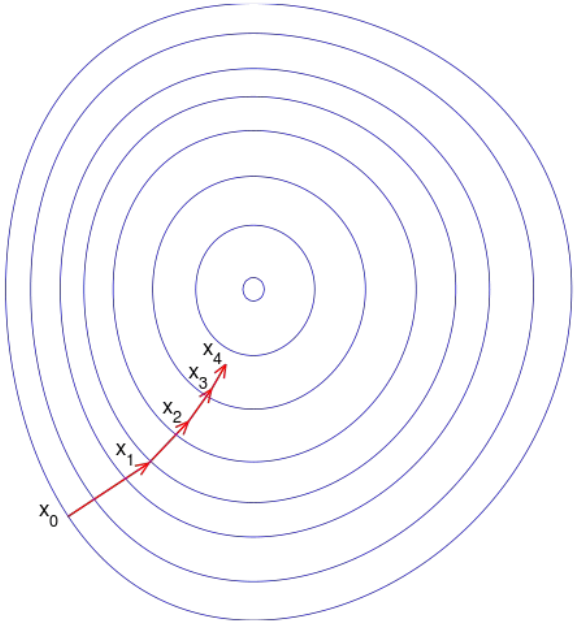




1. Start with an initial guess of your solution.
2. Compute the gradient at that location.
3. “Step” in the opposite direction

Iterate this process until you converge to minimum!

Gradient Descent on level surfaces



$\alpha > 0$

Current estimate of weights

Gradient at current weight estimate

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} L(\mathbf{w}_t)$$

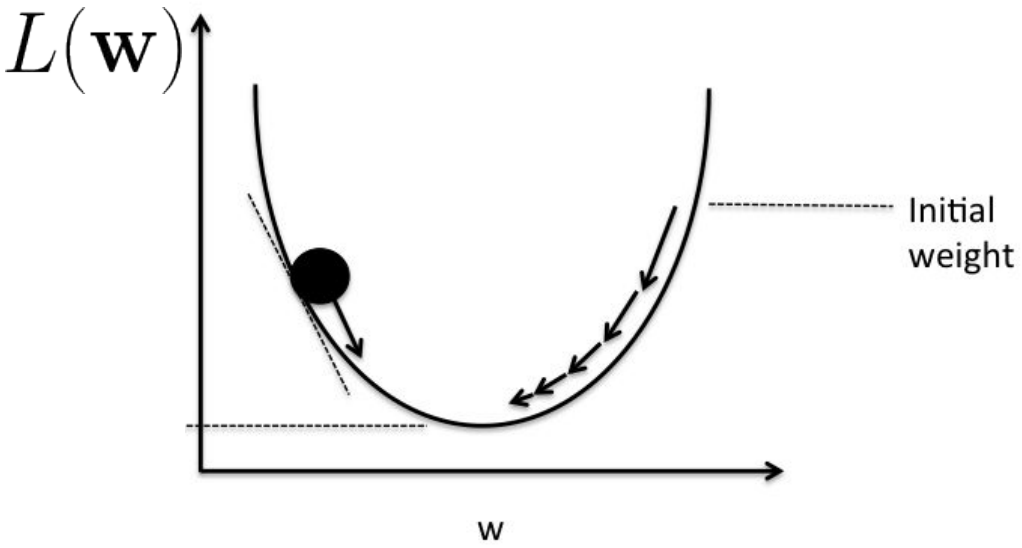
For linear neural network

$$L(\mathbf{w}) = (\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y})$$

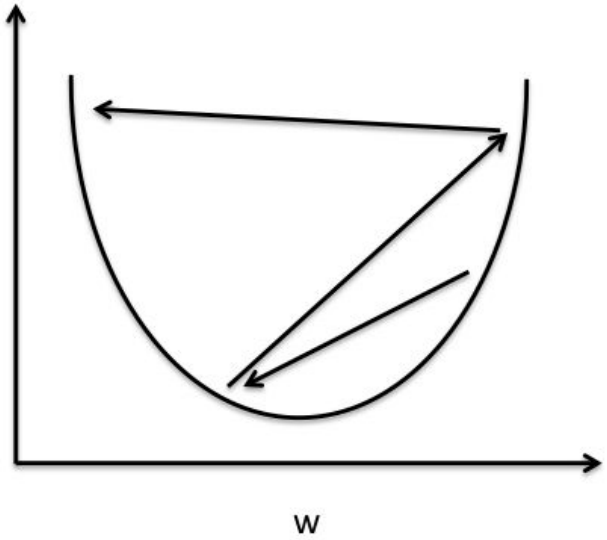
$$\nabla_{\mathbf{w}} L(\mathbf{w}) = (\mathbf{y} - \mathbf{X}\mathbf{w})^T \mathbf{X}$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha (\mathbf{y} - \mathbf{X}\mathbf{w}_t)^T \mathbf{X}$$

Good step size



Bad step size



$0 < \alpha < 1.0$

There can be more optimal ways of selecting the best step size for each iteration, but not used in practical Gradient Descent

Stochastic Gradient Descent

Batch gradient descent: Computationally expensive for lots of data!

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha(\mathbf{y} - \mathbf{X}\mathbf{w}_t)^T\mathbf{X}$$

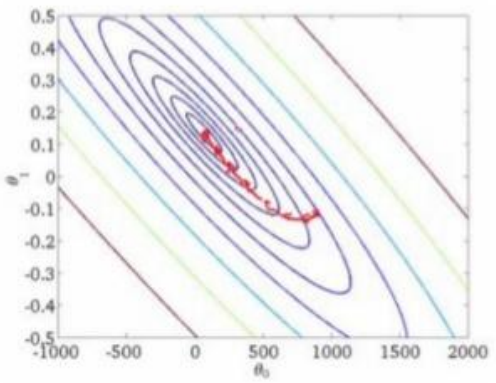
Online learning: compute (stochastic) gradient *one* example at a time

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha(y_i - \mathbf{x}_i^T\mathbf{w}_t)\mathbf{x}_i$$

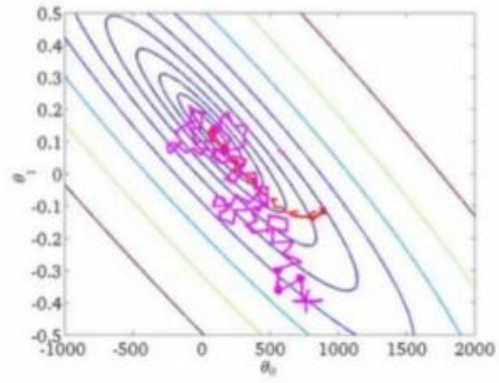
$$i = 1, \ldots, N$$

Stochastic Gradient Descent vs Batch

Batch gradient descent:



Stochastic gradient descent:

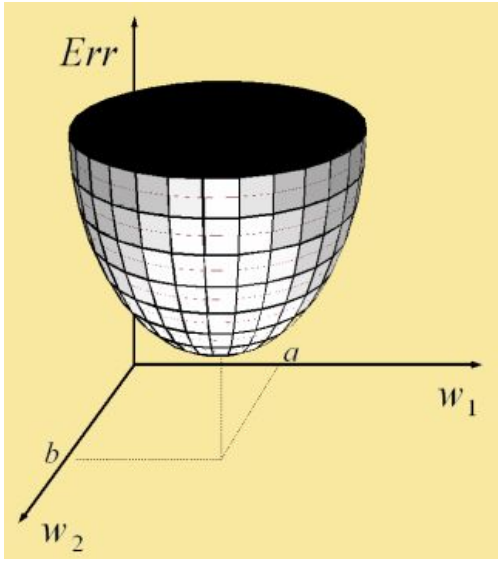


Compromise between extremes: mini-batch gradient descent

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha(\mathbf{y}_M - \mathbf{X}_M \mathbf{w}_t)^T \mathbf{X}_M$$

Batch size: $M \ll N$

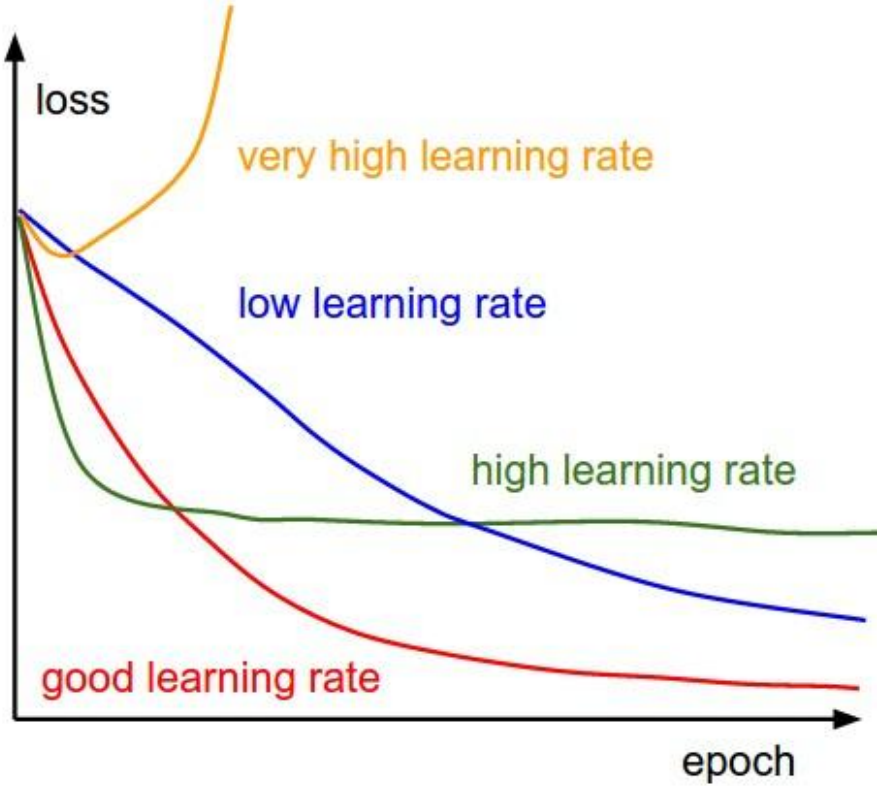
Discussion: is the optimal solution always good?



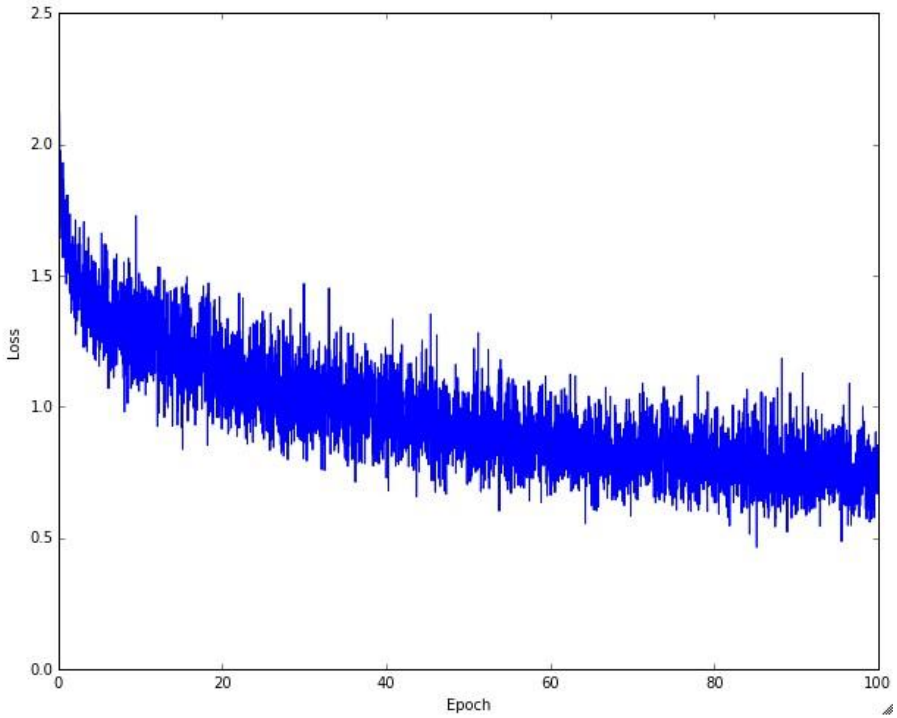
- Solving least squares loss functions means finding the global minimum
- The shape of the loss function is determined by the training data:
- Finding global minimum is only “global” for training data!

Does stochastic gradient descent provide a hedge?

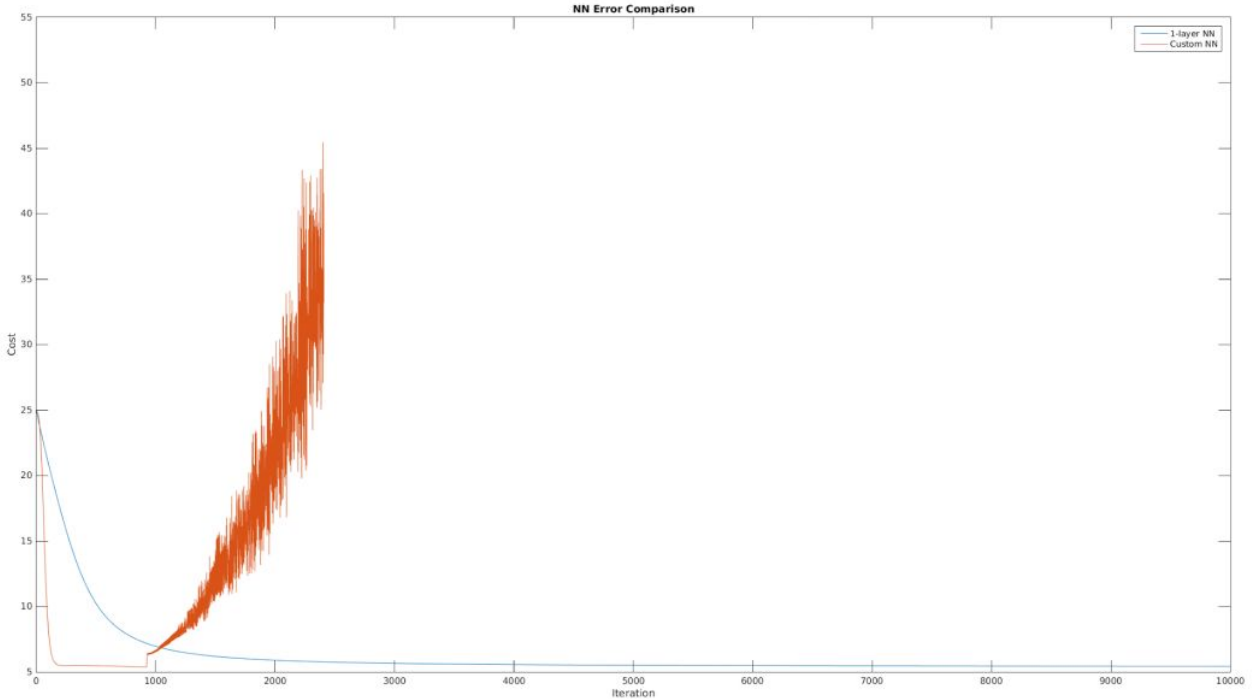
Learning curves



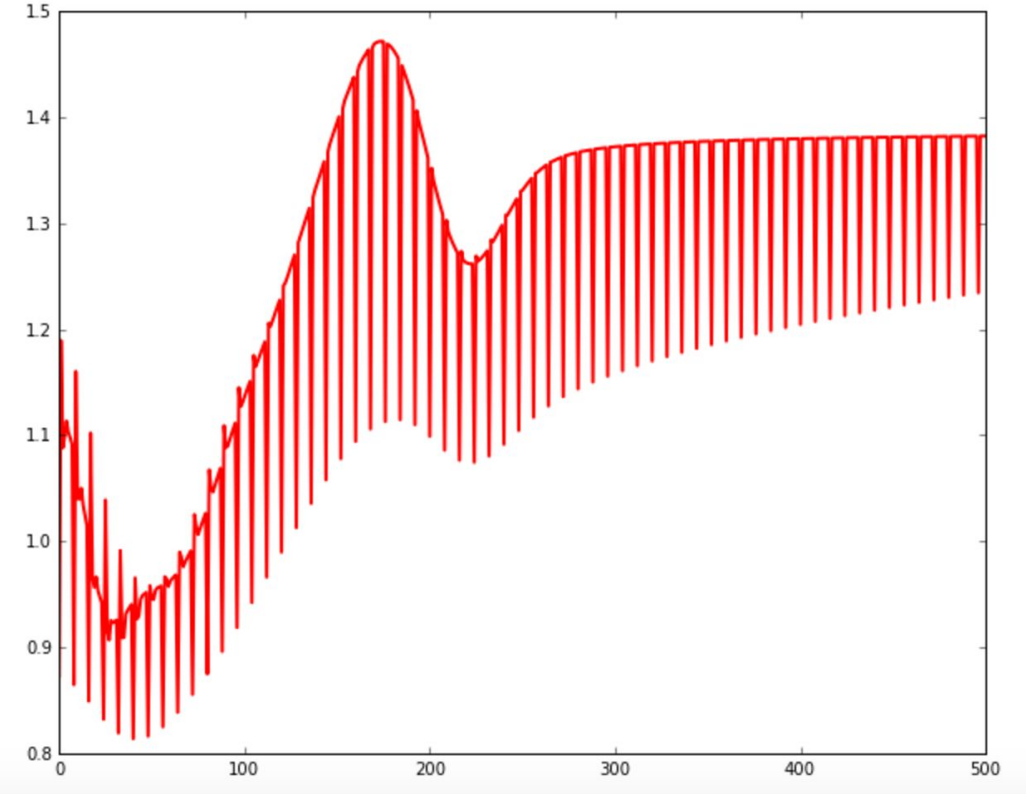
Learning curves in practice



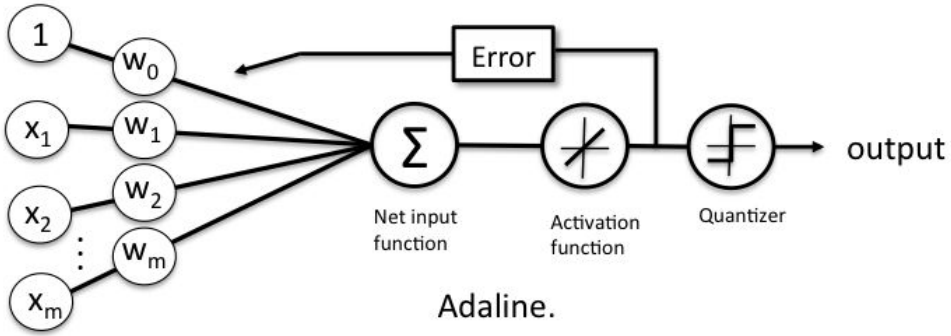
Learning curves in practice



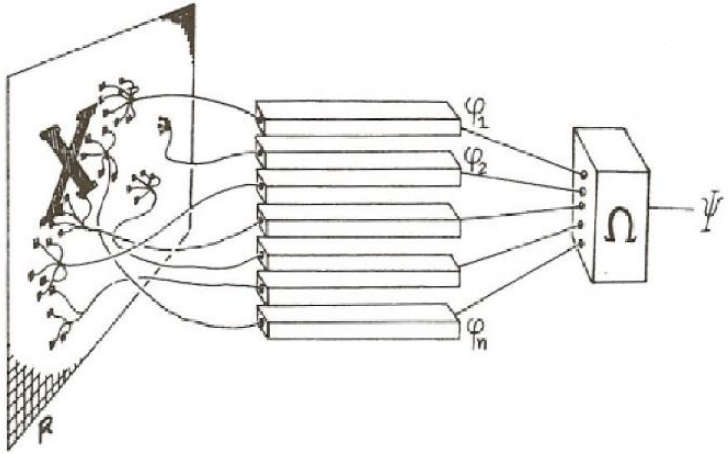
Learning curves in practice



Going back to original ANN

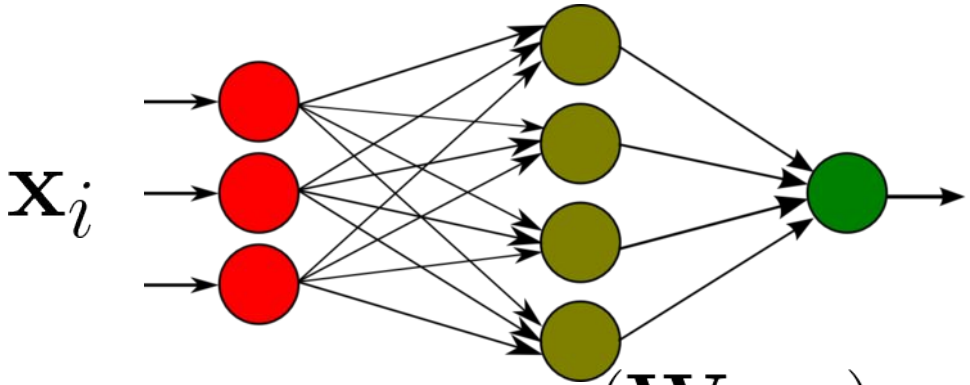


How we think of the Perceptron and ADALINE today



Original architecture from Rosenblatt's work

Add a hidden layer: either random or designed



Non-linear activation function

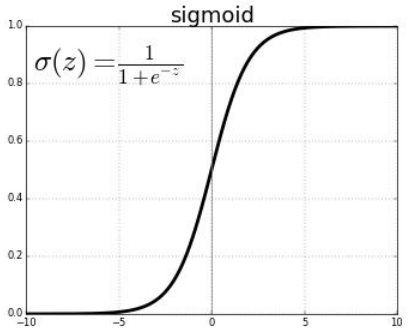
Input layer weights: $\mathbf{W}_0 \in \mathbf{R}^{m \times n}$

Input layer weights: $\mathbf{W}_1 \in \mathbf{R}^{1 \times (m+1)}$

Add a hidden layer: either random or designed

Input: $\mathbf{x}_i$

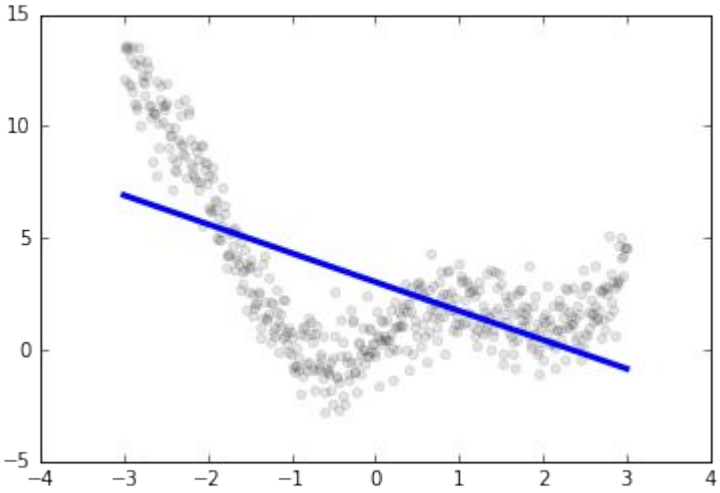
Linear transformation of input followed by non-linear activation function $\mathbf{z}_i = \sigma(\mathbf{W}_0 \mathbf{x}_i)$



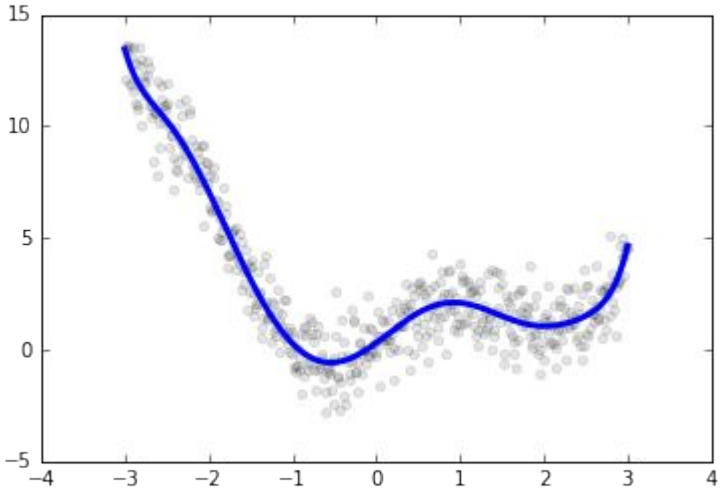
What do we use for the input layer weights? Select at random!

Learn output layer weights on transformed input $y_i = \mathbf{W}_1 \mathbf{z}_i$

Learn using the methods (such as online learning) we have discussed

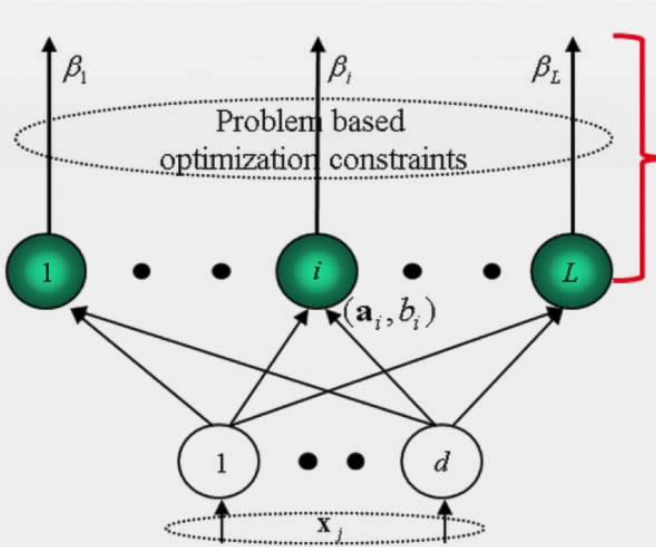


No hidden layers



Random hidden layer

“Extreme Learning Machines” : Rosenblatt’s idea?



Feature learning
Clustering
Regression
Classification

L Random Hidden Neurons (which need not be algebraic sum based) or other ELM feature mappings. Different type of output functions could be used in different neurons:

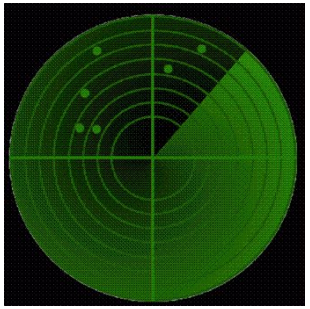
$$h_i(\mathbf{x}) = G_i(\mathbf{a}_i, b_i, \mathbf{x})$$

d Input Nodes

Extreme learning machines

Binary classification: make 1 of 2 choices

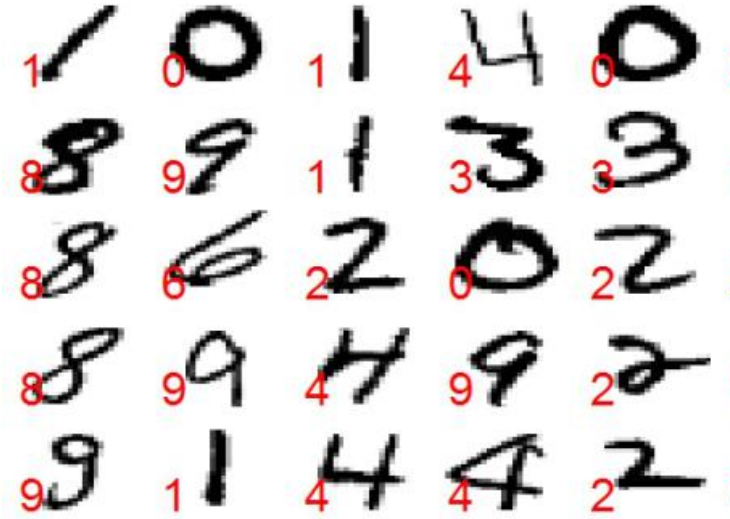
- Detection theory: is a signal present or not?
  - Used extensively in radar in World War 2!
  - Metrics and terminology from the (ROC curve, AUC, F-score, etc.)



- Other examples:
  - Should I give a loan to this customer or not?
  - Is this transaction fraudulent or not?
  - Is this food healthy or not?
  - Is this email spam or not?

Multiclass/multinomial classification:

- Determine 1 of *many* possible choices

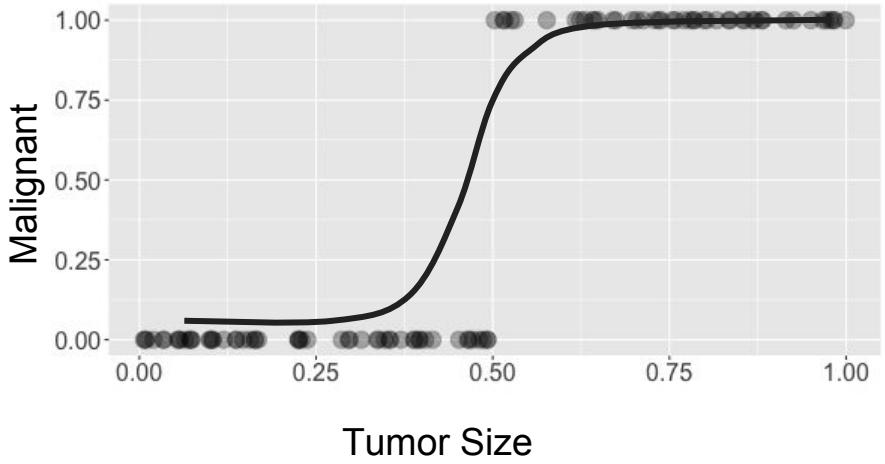


- Examples:
  - Character classification
  - What product will a customer buy?
  - What type of customer buys this product?
  - What type of email is this?

http://www.r-bloggers.com/a-little-h2o-deeplearning-experiment-on-the-mnist-data-set/

Logistic Regression

- A model for giving us the probability of an event
  - Logistic function takes any input features and maps to a number between 0 and 1 (interpreted as a probability)



Logistic function is an “s-curve” and the linear model controls where to place the curve and how fast it rises

Logistic Regression

- A linear model for computing the log-odds ratio of an event

- For example, consider *p* as probability of:
  - examples include: detecting fraud, passing an exam, etc.

- Odds ratio: event happening versus not happening $\frac{p}{1-p}$

Logistic Regression

- The logs odds is then defined as:

$$\log(\frac{p}{1-p})$$

- In logistic regression, we use a linear equation to model the log-odds

$$\log(\frac{p}{1-p}) = w_0 + w_1 x_1 + \ldots + w_n x_n$$

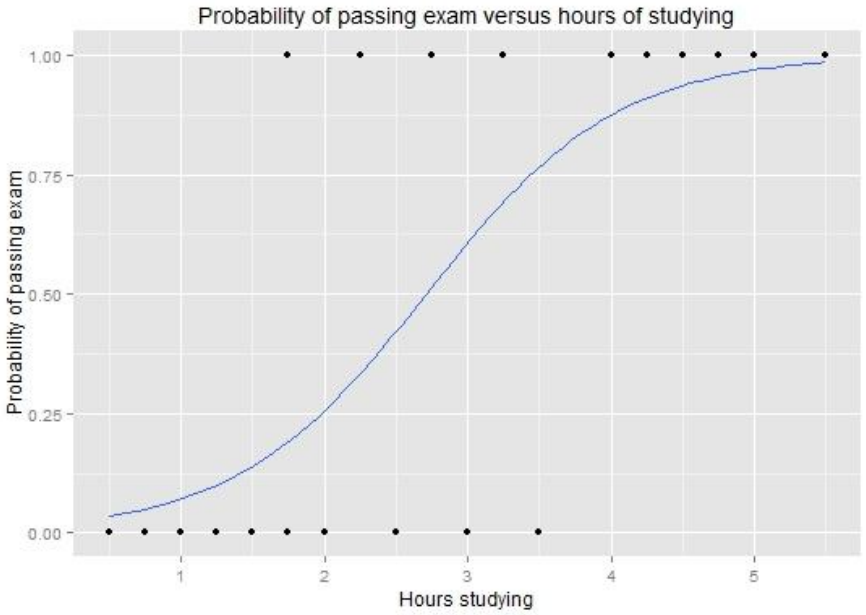
Logistic Regression

- Can be also written as

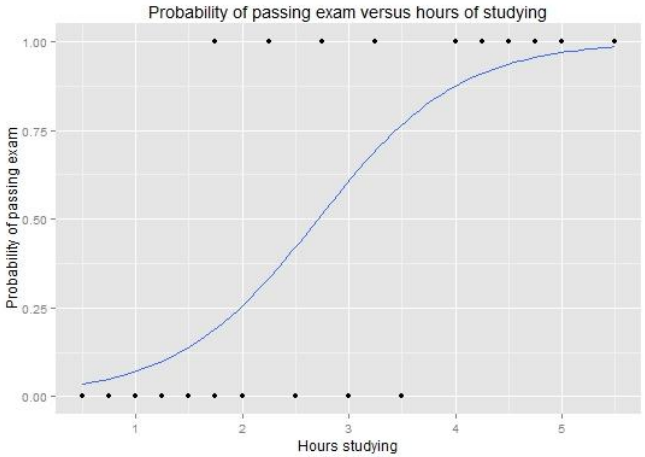
$$p = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + ... + w_n x_n)}}$$

- Example: predict probability of passing an exam from number of hours studied

$$p = \frac{1}{1 + e^{-(w_0 + w_1 x_1)}}$$



Logit function



$$p = \frac{1}{1 + e^{-(w_0 + w_1 x_1)}}$$

Adjusting w_0 shifts the s-curve left and right

Adjusting w_1 controls the rate of s-curve "ramping up"

Need to define a loss function so we can find appropriate set of weights

Loss function for logistic regression

- Kullback-Leibler divergence: measuring the “distance” between two distributions
  - The symmetric variant is a legitimate distance metric that can be used for K-Nearest Neighbors or other distance

$$D_{KL} = \sum_{j} P(j) \log \frac{P(j)}{Q(j)}$$

Loss function for logistic regression

Definition of KL divergence for a set with two items (binary classification)

$$D_{KL} = -\sum_{j \in \{0,1\}} P(j) \log Q(j) + \sum_{j \in \{0,1\}} P(j) \log P(j)$$

"Cross entropy" loss for example *i*:

$$L_i(\mathbf{w}) = -y_i \log(\sigma(\mathbf{w}^T \mathbf{x}_i)) - (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i))$$

Loss function for logistic regression

$$L_i(\mathbf{w}) = -y_i \log(\sigma(\mathbf{w}^T \mathbf{x}_i)) - (1 - y_i)\log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i))$$

Apply stochastic gradient descent (SGD) on cross entropy loss function:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla L_i(\mathbf{w}_t) \quad i = 1, \ldots, N$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha (y_i - \sigma(\mathbf{w}_t^T \mathbf{x}_i))\mathbf{x}_i$$

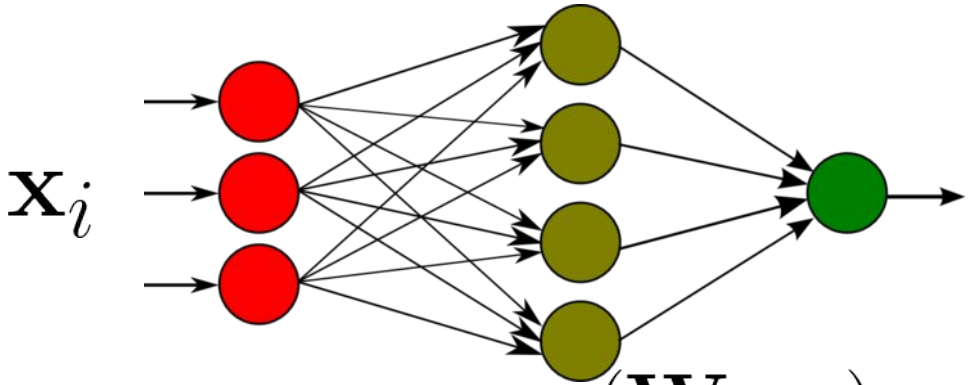
Can easily extend to mini-batch SGD: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \sum_{k=1}^{M} \nabla L_k(\mathbf{w}_t)$

Other optimization strategies

- Hessian optimization, Gauss-Newton, Conjugate Gradient Descent, L-BGFS, …
  - Used extensively in solving optimization problems, but not so much in Neural Nets

- More common to use previous gradient step information:
  - Momentum, ADAM, Ada-Grad

Multilayer Neural Networks



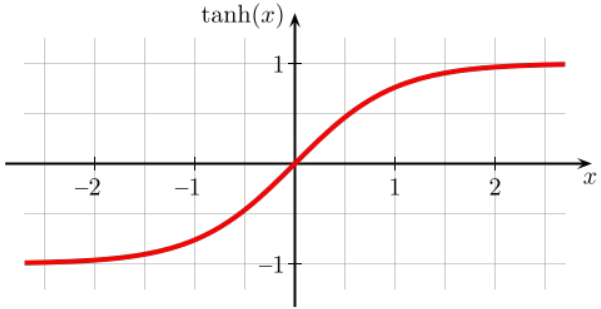
Input layer weights: $\mathbf{W}_0 \in \mathbf{R}^{m \times n}$

Output layer weights: $\mathbf{W}_1 \in \mathbf{R}^{1 \times (m+1)}$

Single-hidden layer neural networks

Input: $\mathbf{x}_i$

Linear transformation of input followed by non-linear activation function $\mathbf{z}_i = \sigma(\mathbf{W}_0\mathbf{x}_i)$



Output layer weights on transformed input $y_i = \mathbf{W}_1\mathbf{z}_i$

Can we learn both sets of weights?

$\mathbf{W}_0 \in \mathbf{R}^{m \times n}$

$\mathbf{W}_1 \in \mathbf{R}^{1 \times (m+1)}$

Learn both sets of weights

$$L(\mathbf{W}) = \sum_{i=1}^{N} (y_i - \mathbf{W}_1 \sigma(\mathbf{W}_0 \mathbf{x}_i))^2$$

$$\mathbf{W} = (\mathbf{W}_0, \mathbf{W}_1) \qquad \underset{\mathbf{W}}{\operatorname{argmin}} L(\mathbf{W})$$

What is deep learning?

Shallow

Traditional linear inference

$$p(y|\mathbf{x}) \propto W_1\mathbf{x}$$

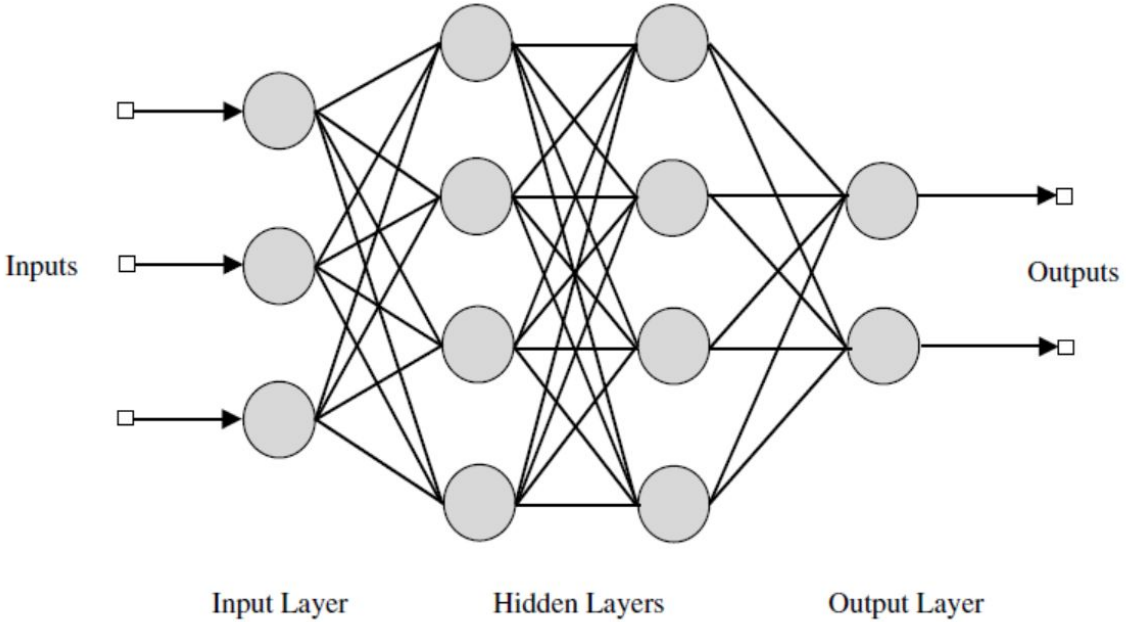
$$p(y|\mathbf{x}) \propto W_2 f_1(W_1\mathbf{x})$$

$$p(y|\mathbf{x}) \propto W_3 f_2\Big((W_2 f_1(W_1\mathbf{x})\Big)$$

Deep

$$p(y|\mathbf{x}) \propto W_{n+1} f_n\Big(\ldots W_3 f_2\Big((W_2 f_1(W_1\mathbf{x}))\Big)\Big)$$

Neural Networks



$$\mathbf{w} = (\mathbf{W}_0, \mathbf{W}_1, \mathbf{W}_2)$$

Note: We have a lot more parameters than before!

$$\mathbf{x}$$

$$\downarrow$$

$$\mathbf{z}_1 = \sigma(\mathbf{W}_0\mathbf{x})$$

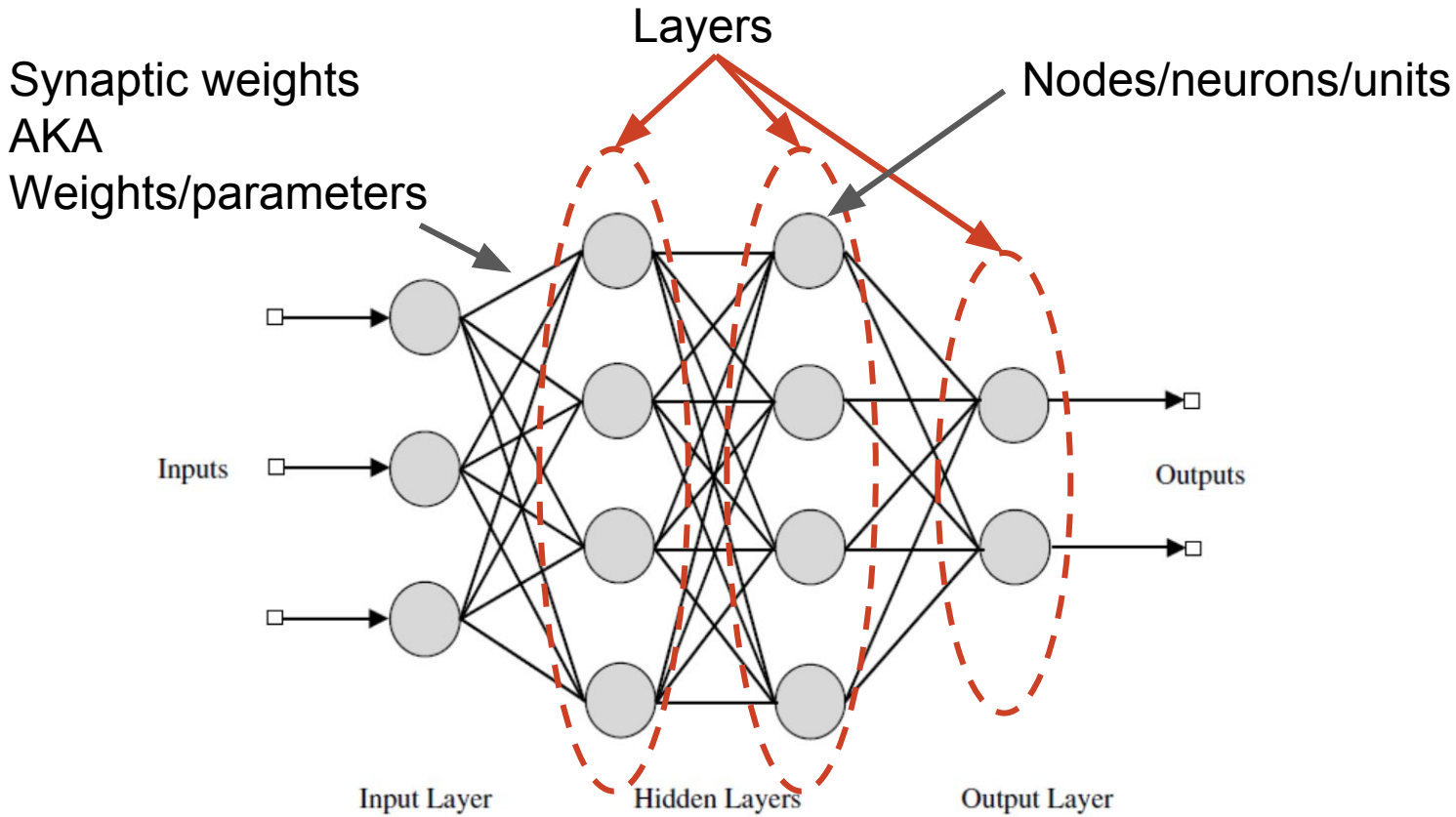
$$\downarrow$$

$$\mathbf{z}_2 = \sigma(\mathbf{W}_1\mathbf{z}_1)$$

$$\downarrow$$

$$\mathbf{y} \approx \mathbf{W}_2\mathbf{z}_2$$

Neural Networks jargon:



Explosion in number of parameters

$$\mathbf{x} \qquad \mathbf{x} \in \mathbf{R}^n$$

$$\downarrow$$

$$\mathbf{y} \approx \mathbf{W}\mathbf{x} \quad \mathbf{y} \in \mathbf{R}^m$$

Number of parameters that we need to find for linear model: $m \times (n+1) = O(mn)$

$$\mathbf{x}$$

$$\downarrow$$

$$\mathbf{z}_1 = \sigma(\mathbf{W}_0\mathbf{x})$$

$$\downarrow$$

$$\mathbf{z}_2 = \sigma(\mathbf{W}_1\mathbf{z}_1)$$

$$\downarrow$$

$$\mathbf{y} \approx \mathbf{W}_2\mathbf{z}_2$$

Explosion in number of parameters

Number of parameters that we need to find for linear model:

$$m \times (n+1) = O(mn)$$

Number of parameters that we need to find for Neural Network:

$$O(m_1 n + m_2 m_1 + m m_2)$$

$$\mathbf{x}$$

$$\downarrow$$

$$\mathbf{z}_1 = \sigma(\mathbf{W}_0 \mathbf{x})$$

$$\downarrow$$

$$\mathbf{z}_2 = \sigma(\mathbf{W}_1 \mathbf{z}_1)$$

$$\downarrow$$

$$\mathbf{y} \approx \mathbf{W}_2 \mathbf{z}_2$$

What's happening here?

- Problem: linear model often fails to capture the relationship between input and output.

- Perform sequence of transformations that "untangles" the nonlinearities so that we can use a linear model in the transformed space

$$\mathbf{x}$$

$$\downarrow$$

$$\mathbf{z}_1 = \sigma(\mathbf{W}_0\mathbf{x})$$

$$\downarrow$$

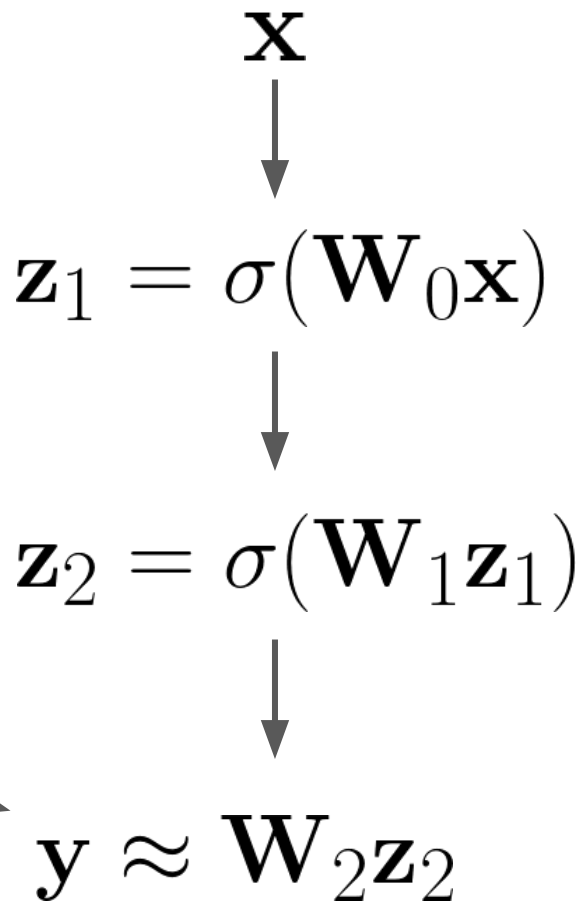
$$\mathbf{z}_2 = \sigma(\mathbf{W}_1\mathbf{z}_1)$$

$$\downarrow$$

$$\mathbf{y} \approx \mathbf{W}_2\mathbf{z}_2$$

What's happening here?

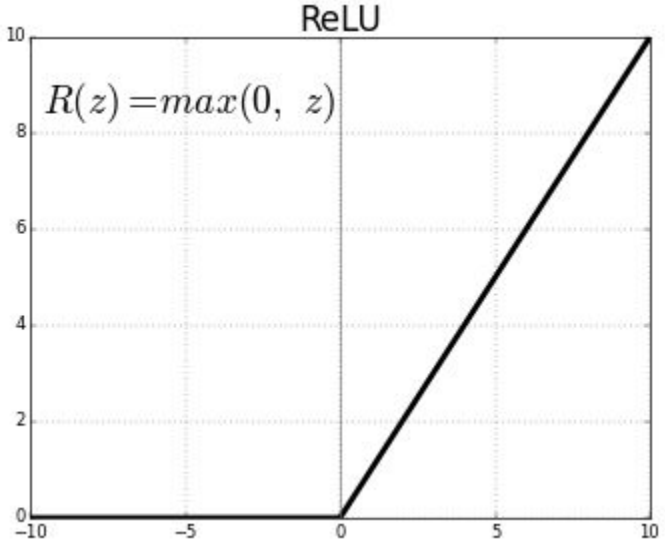
- Each layer is *learning* a new feature



Hopefully the features learned at the output layer are sufficient for a linear model to solve effectively!

Activation functions

In principle, “any” nonlinear function can be used. Rectified Linear Unit (AKA ReLU) used the most commonly



$$\mathbf{x}$$

$$\downarrow$$

$$\mathbf{z}_1 = \sigma(\mathbf{W}_0\mathbf{x})$$

$$\downarrow$$

$$\mathbf{z}_2 = \sigma(\mathbf{W}_1\mathbf{z}_1)$$

$$\downarrow$$

$$\mathbf{y} \approx \mathbf{W}_2\mathbf{z}_2$$

Finding set of weights

- Once architecture defined, specify loss function.
  - Regression: mean squared error
  - Classification: cross entropy or hinge loss

- Apply stochastic gradient descent (SGD): $\mathbf{w} = (\mathbf{W}_0, \mathbf{W}_1, \mathbf{W}_2)$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} L(\mathbf{w}_t)$$

$$\mathbf{x}$$

$$\downarrow$$

$$\mathbf{z}_1 = \sigma(\mathbf{W}_0 \mathbf{x})$$

$$\downarrow$$

$$\mathbf{z}_2 = \sigma(\mathbf{W}_1 \mathbf{z}_1)$$

$$\downarrow$$

$$\mathbf{y} \approx \mathbf{W}_2 \mathbf{z}_2$$

Apply stochastic gradient descent

- Computing the gradient on the loss function involves heavy application of the chain rule
  - Referred to as the “backpropagation algorithm”

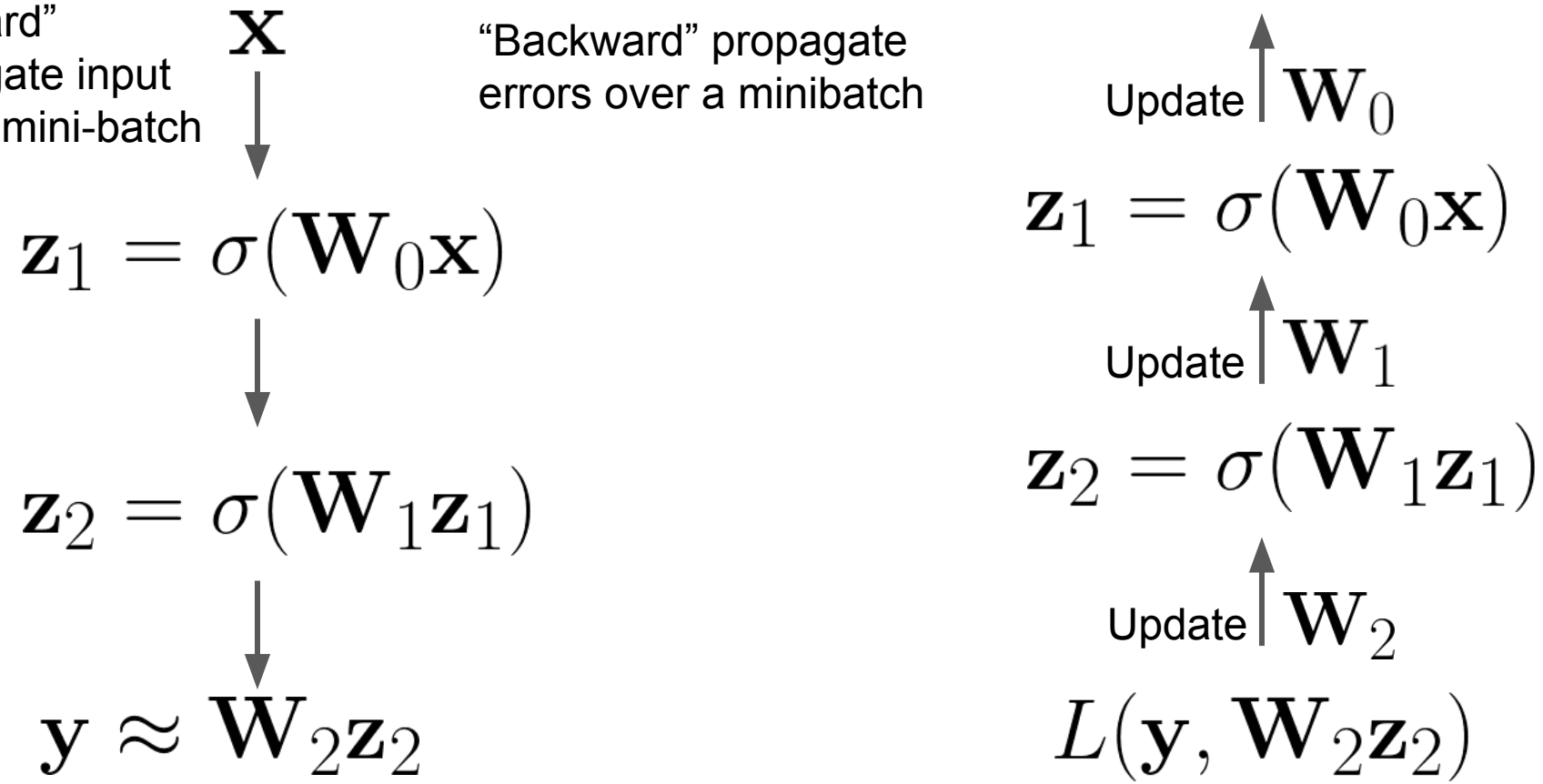
$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} L(\mathbf{w}_t)$$

- Automatic differentiation software makes applying gradient descent easy!

SGD for Neural Networks: backpropagation

"Forward" propagate input over a mini-batch

"Backward" propagate errors over a minibatch



Neural Networks jargon

- Deep learning and neural networks: stressing the idea of more than one hidden layer
- “Width” of the network: number of hidden neurons in a single layer
- Fully connected multilayer perceptron:

- Convolutional layer a type of ANN with *constrained* weights and layers so that the number of parameters is drastically lower (highly relevant for structured input such as images)
- Recurrent Neural Networks

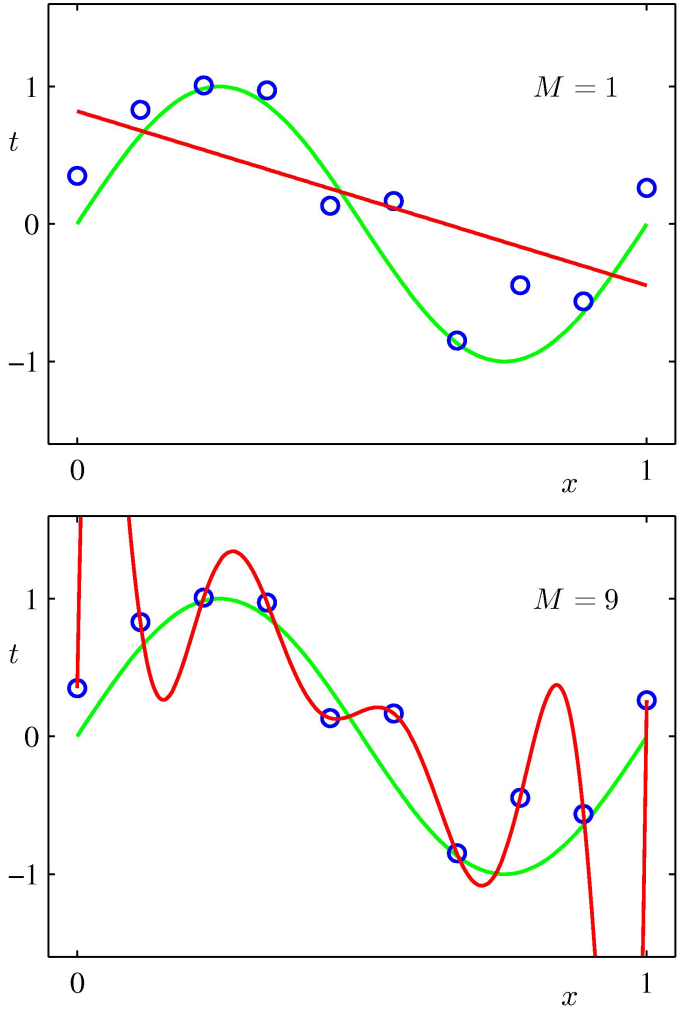
Explosion in number of parameters: easy to overfit

Number of parameters that we need to find for linear model:

$$m \times (n+1) = O(mn)$$

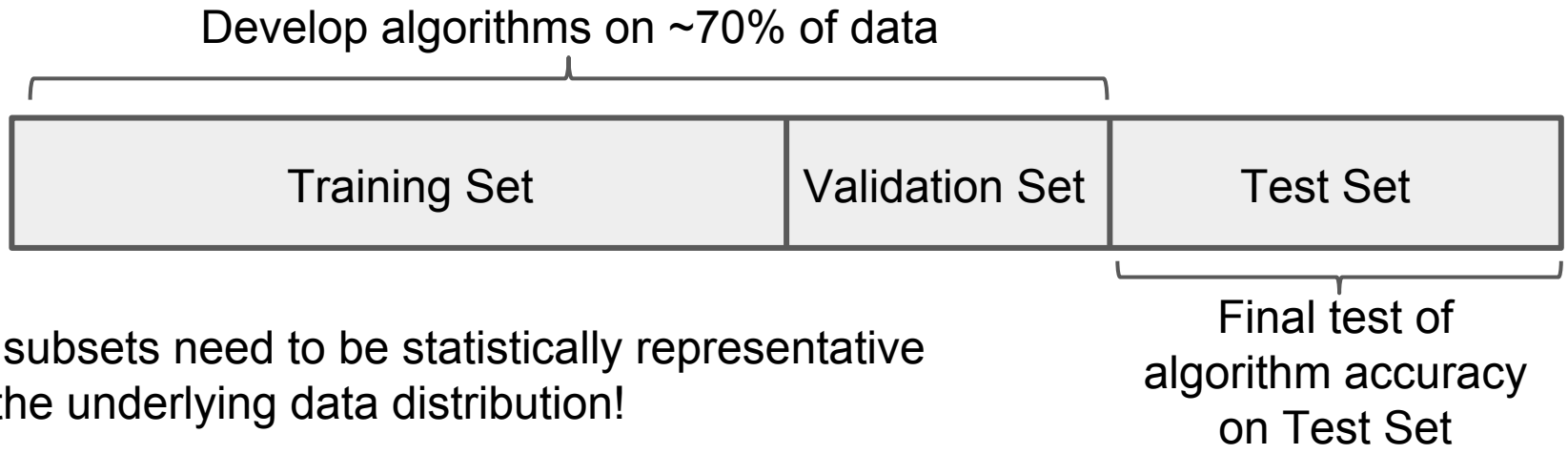
Number of parameters that we need to find for Neural Network:

$$O(m_1 n + m_2 m_1 + m m_2)$$



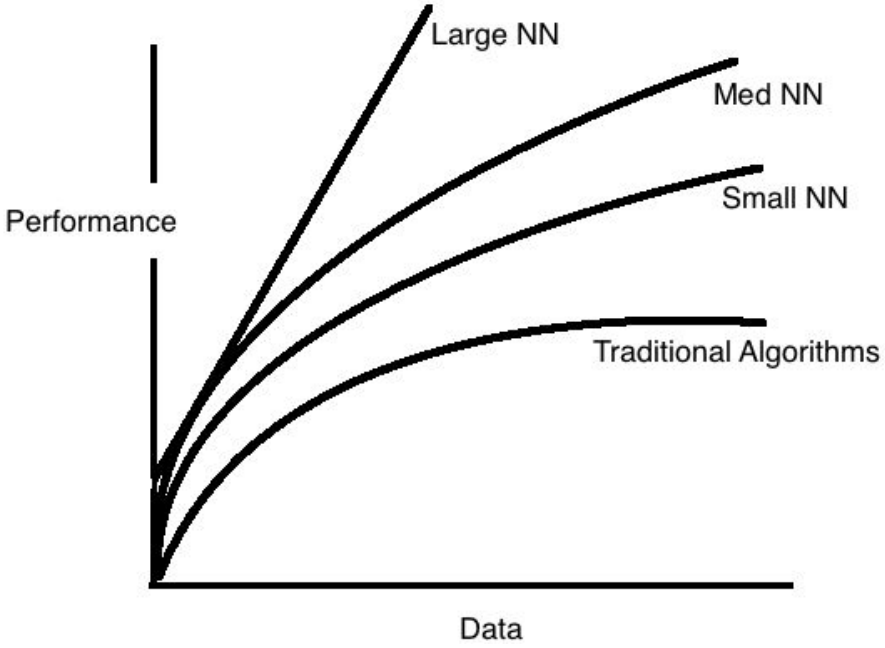
Train, validation, and test sets

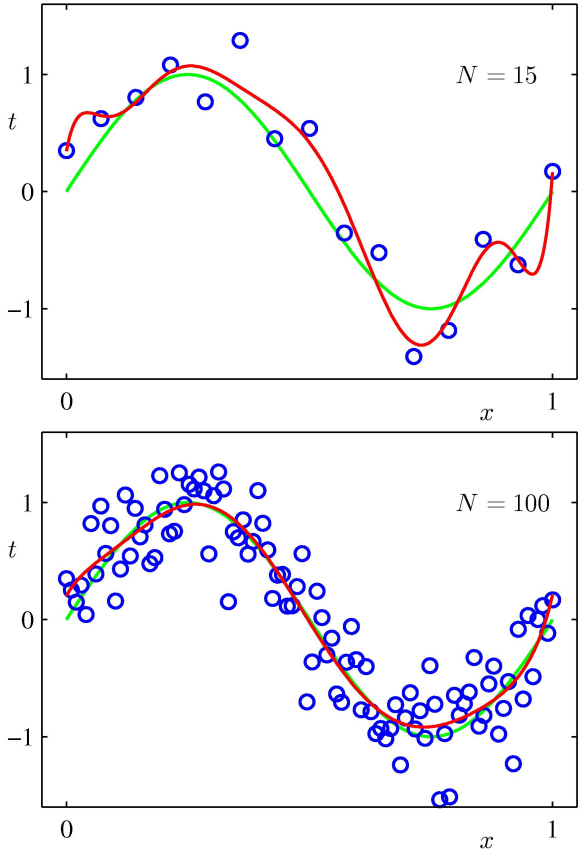
- Constantly evaluating algorithm on the test can also lead to overfitting
  - Better approach: split data into 3 sets



All subsets need to be statistically representative of the underlying data distribution!

“Nuts and bolts of deep learning” by Andrew Ng

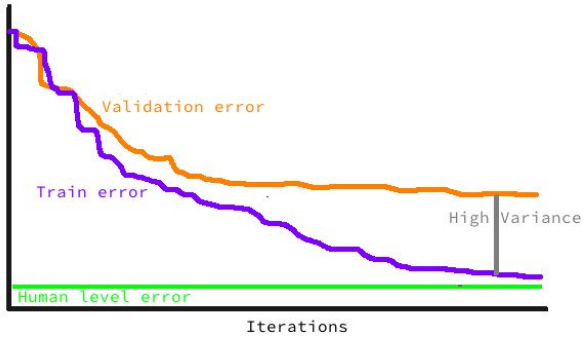




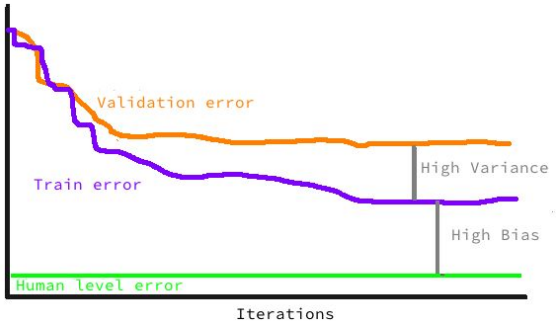
https://medium.com/@aniketvartak/nuts-and-bolts-of-applying-deep-learning-summary-84b8a8e873d5#.m5axyaokt

Learning curves scenarios:

High bias: validation error and train error both bad, but are about the same

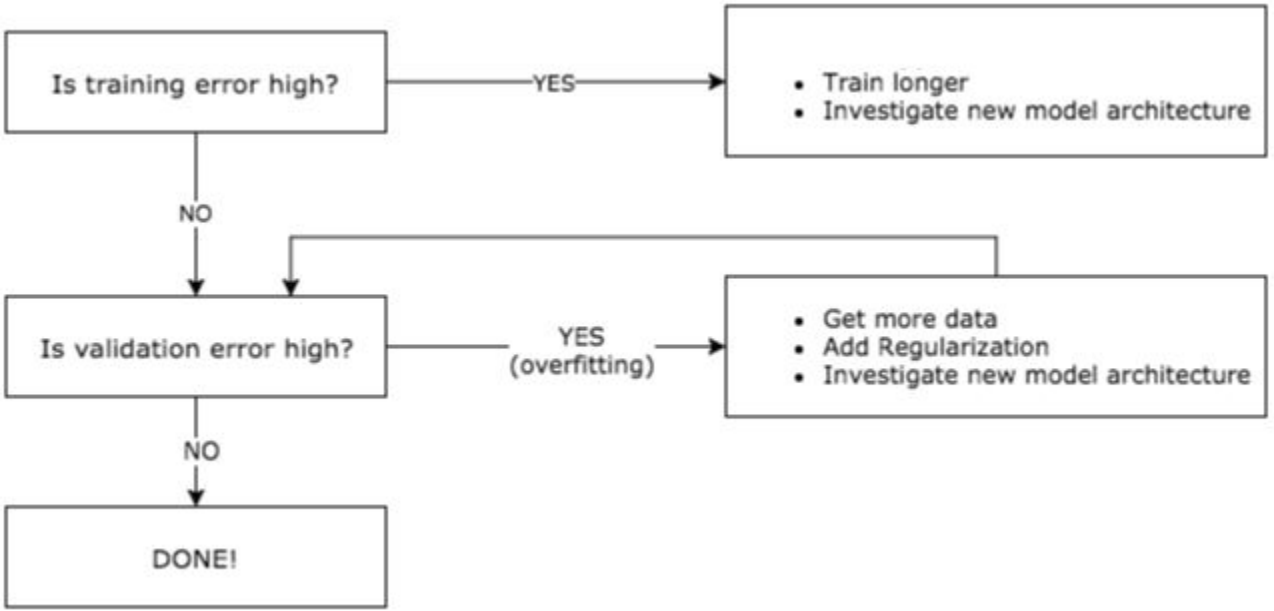


High variance: validation error high but train error low

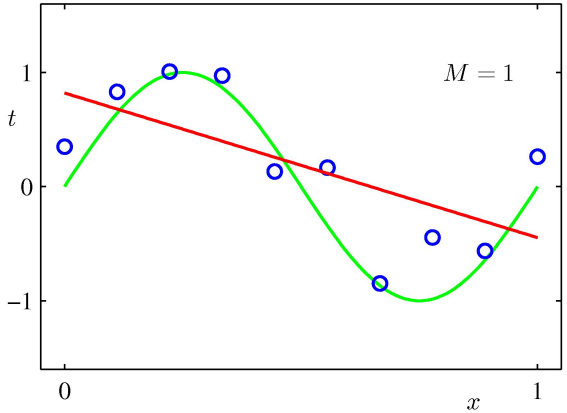


High bias and high variance

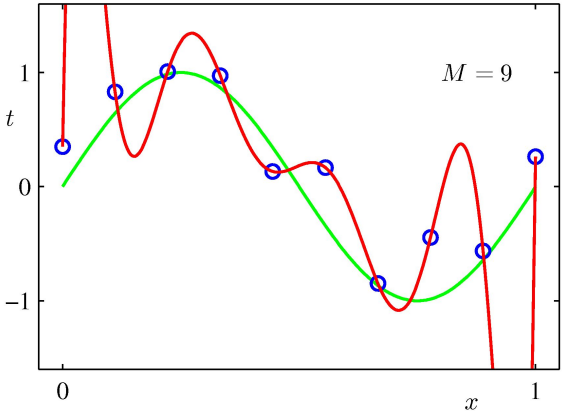
Deep Learning (ML) Flowchart



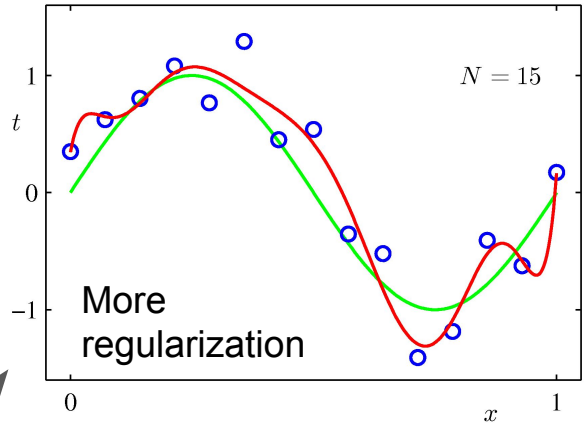
Machine Learning Flowchart

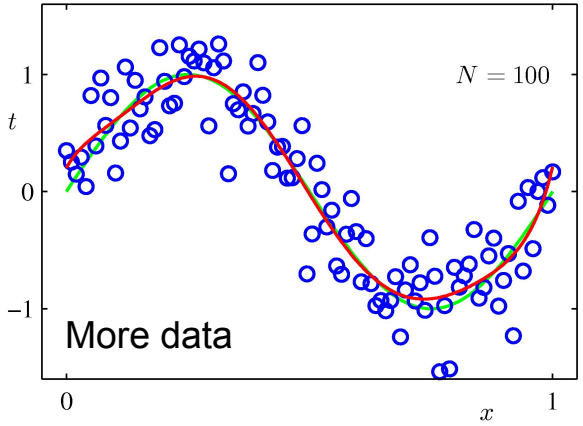


Train error high: investigate new model/architecture

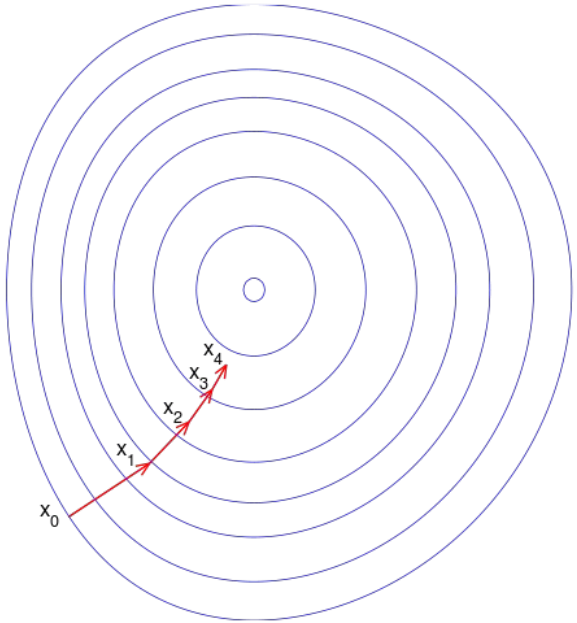


Validation error high: add regularization or get more data





Regularization SGD: choosing initial weights



$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} L(\mathbf{w}_t)$$

- Choose “small” random weights (better than all zeros to break symmetry)
- “Small” to avoid vanishing/exploding gradients
- Xavier initialization often a good strategy

Normalizing inputs

- Normalizing inputs so that they have mean zero and standard deviation helps convergence and better solutions

- Combined with the view that weights should be initialized to “small” random values, it appears that Neural Networks perform better when inputs to activation functions are in a “small” input region

Batch Normalization

- Conclusion from previous ideas is that the inputs to *each* layer should be in a “small” region

- Batch Normalization: normalize inputs to *each* layer

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

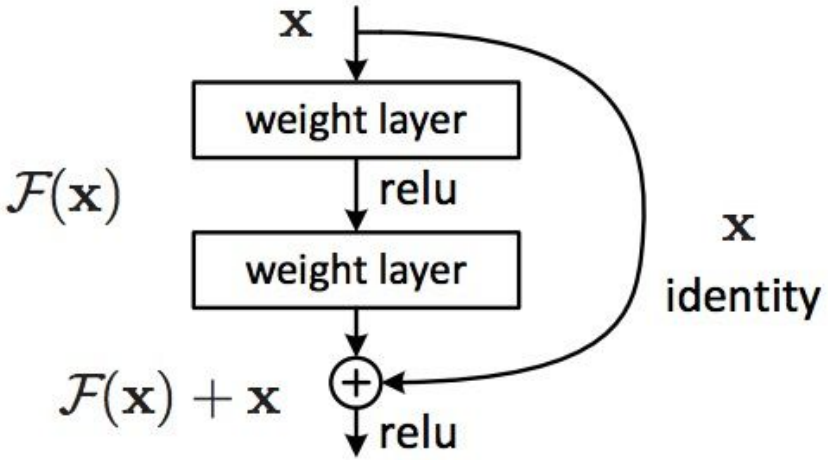
Regularization to prevent overfitting

- “Weight decay” : adding an L2 penalty to the loss function

- “Drop out”: randomly remove neurons from training procedure over each epoch
  - Slows learning but empirically shown to get better performance

Regularization to prevent overfitting

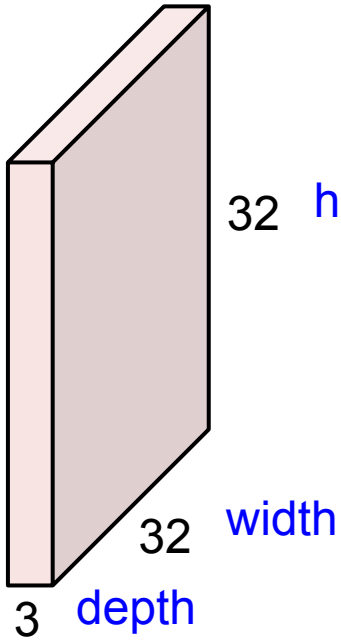
- Add skip layers
  - A layer that is an identity map
  - ResNet architecture: https://arxiv.org/abs/1512.03385



- Randomly remove layers (but keep skip layer) during training
  - Deep Networks with Stochastic Depth: https://arxiv.org/abs/1603.09382

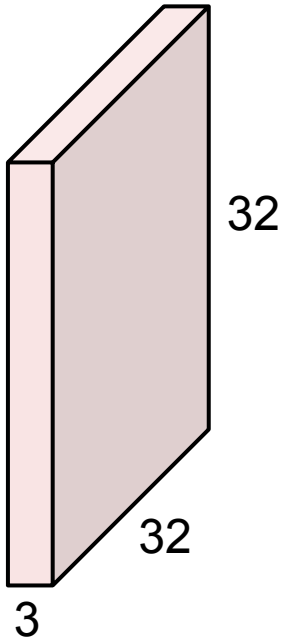
Convolution Layer

32x32x3 image

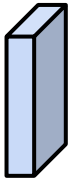


Convolution Layer

32x32x3 image



5x5x3 filter

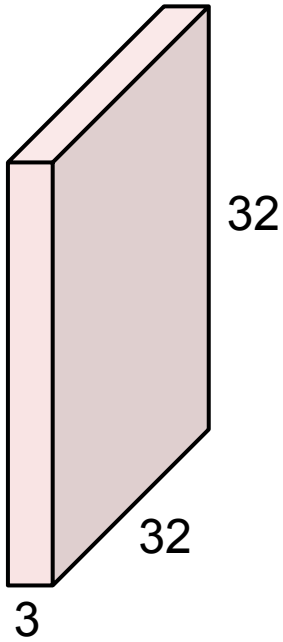


Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

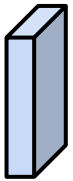
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

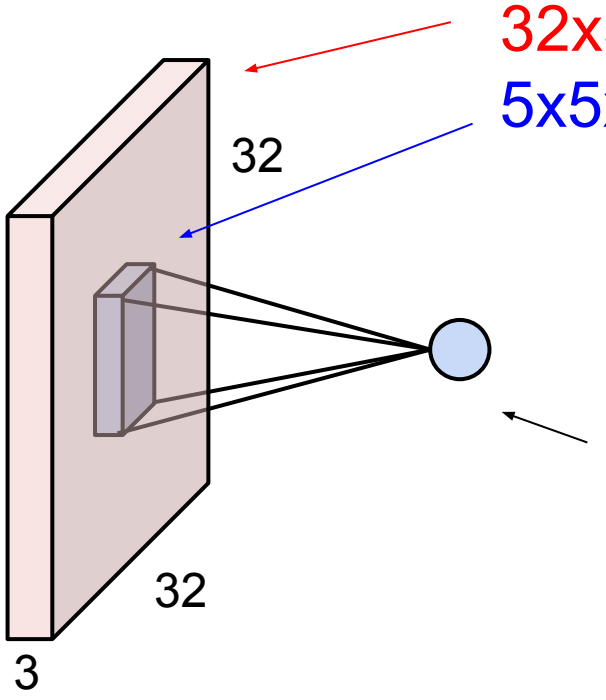


5x5x3 filter



Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

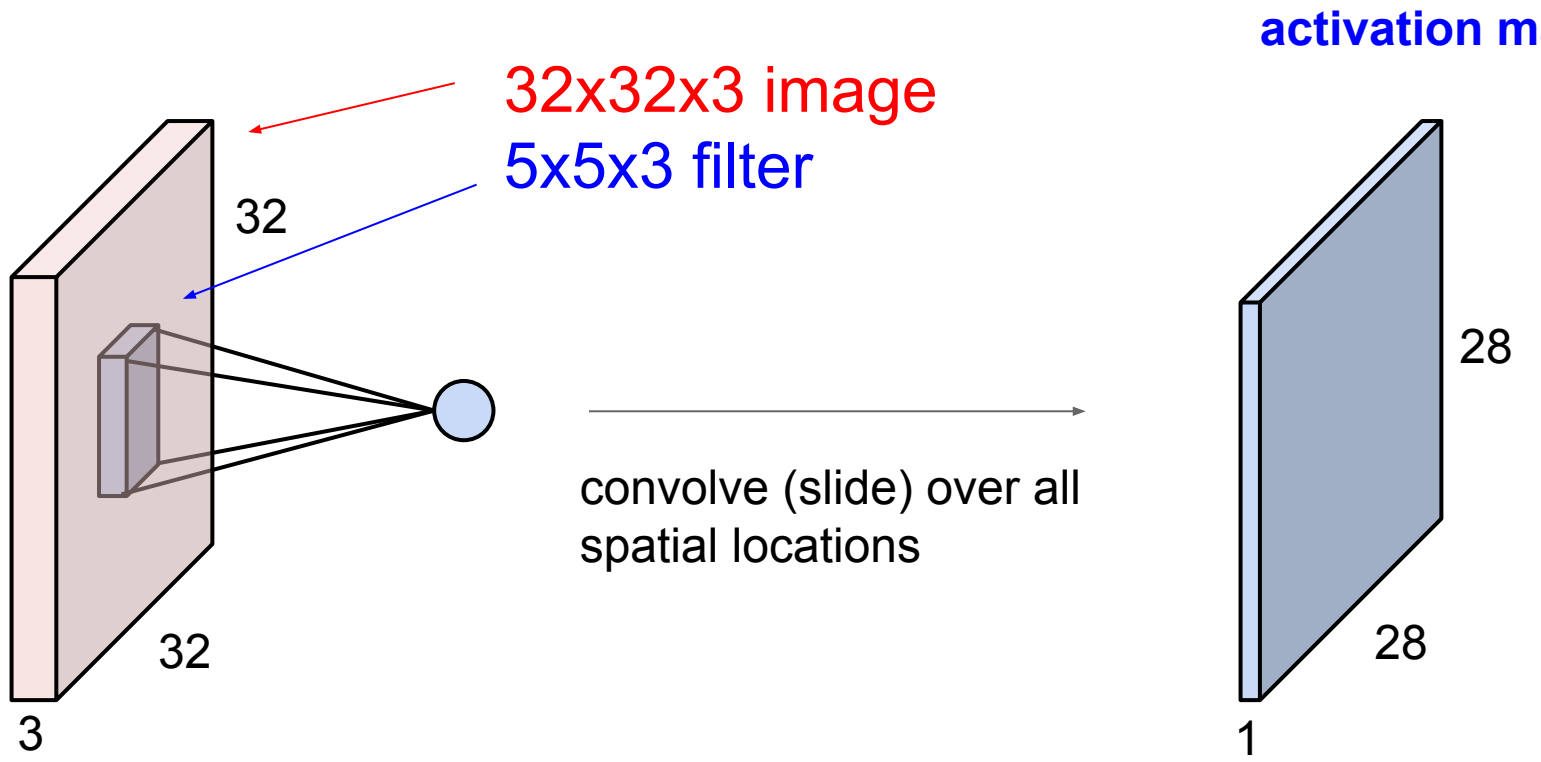
Convolution Layer



1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 = 75-dimensional dot product + bias)

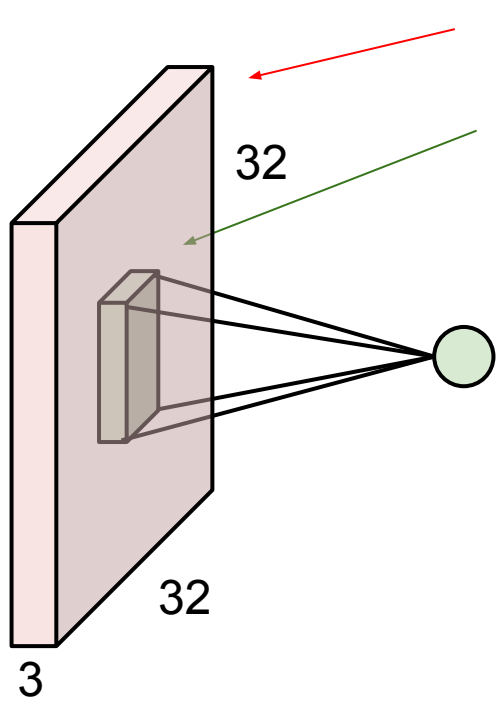
$$w^T x + b$$

Convolution Layer

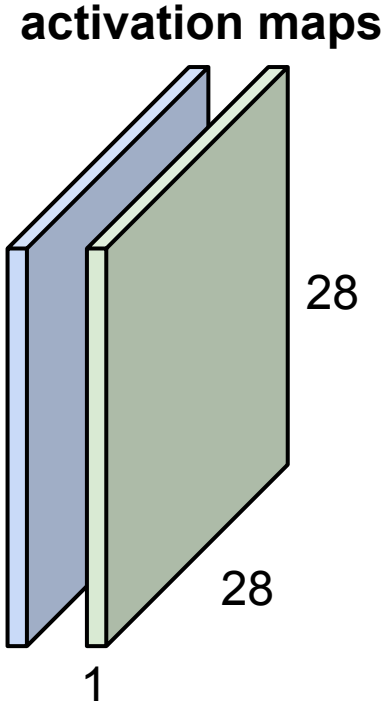


Convolution Layer

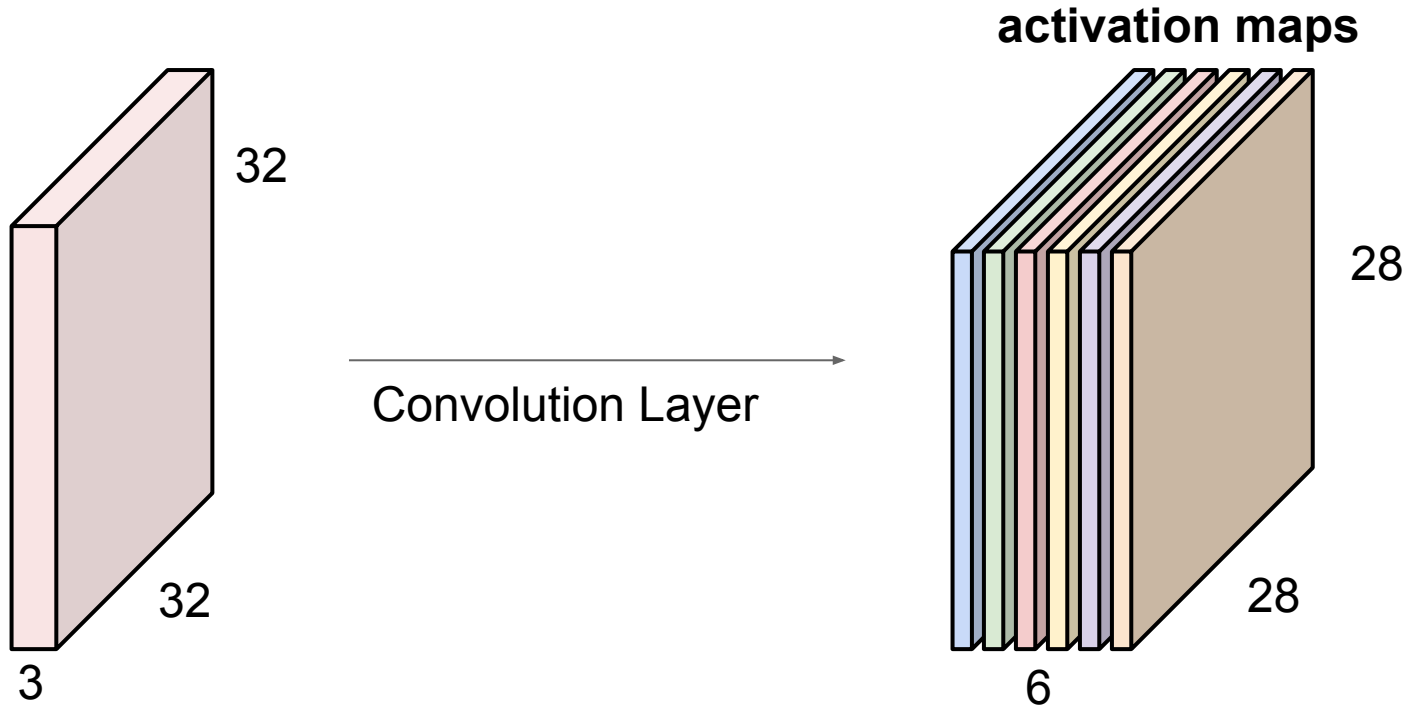
consider a second, green filter



convolve (slide) over all spatial locations

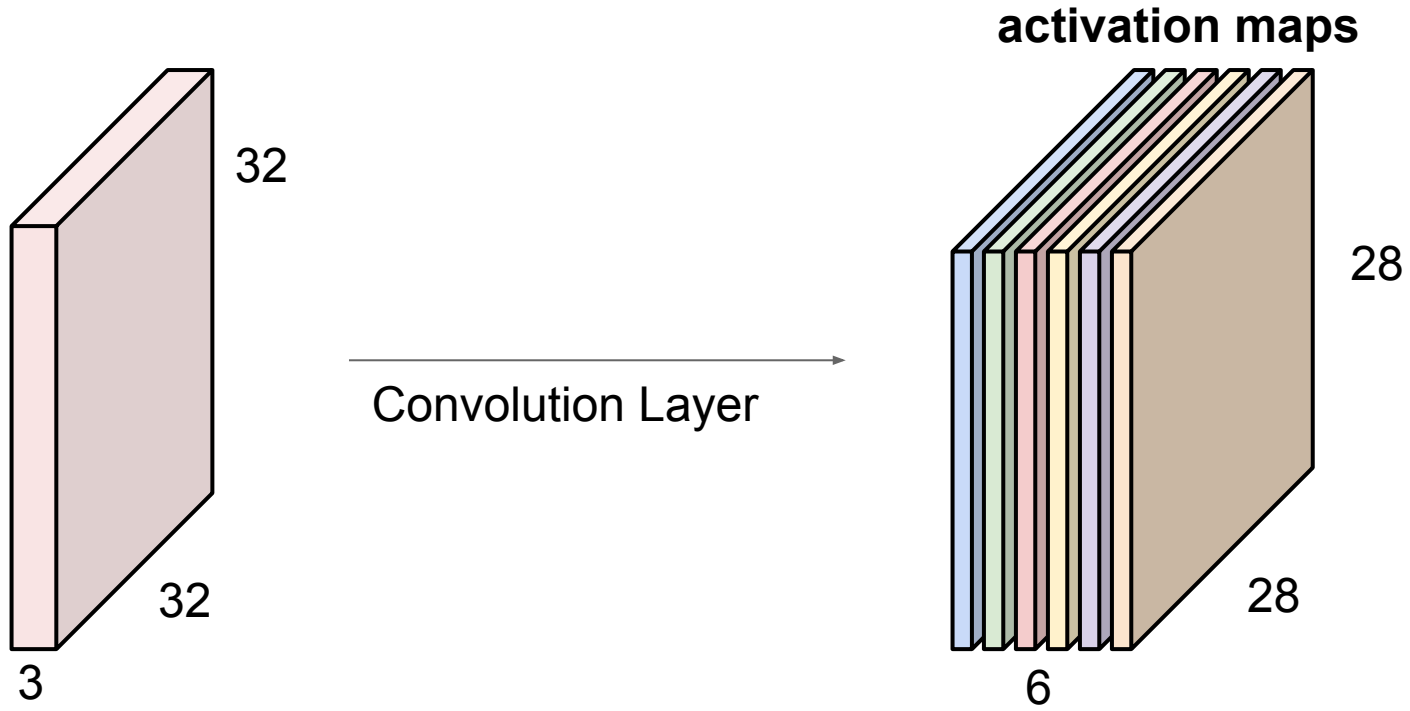


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

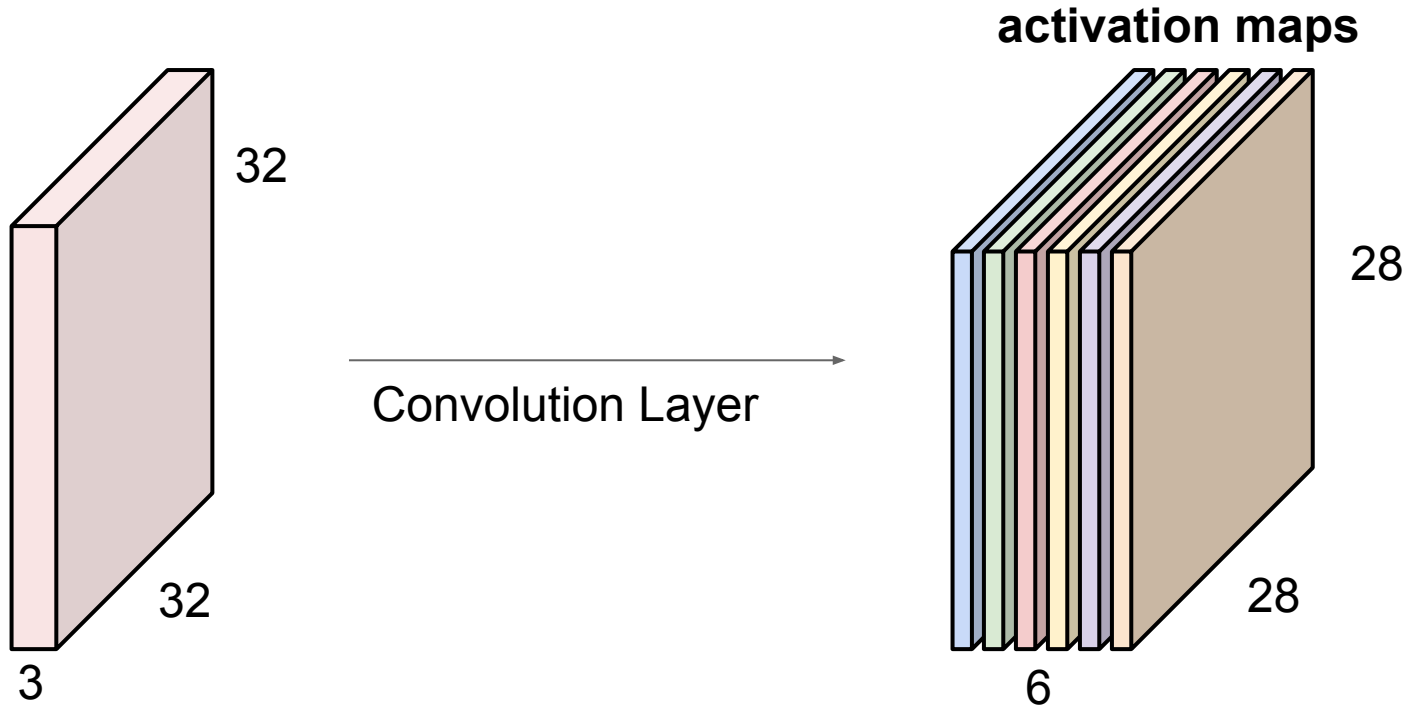
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We processed [32x32x3] volume into [28x28x6] volume.

Q: how many parameters would this be if we used a fully connected layer instead?

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

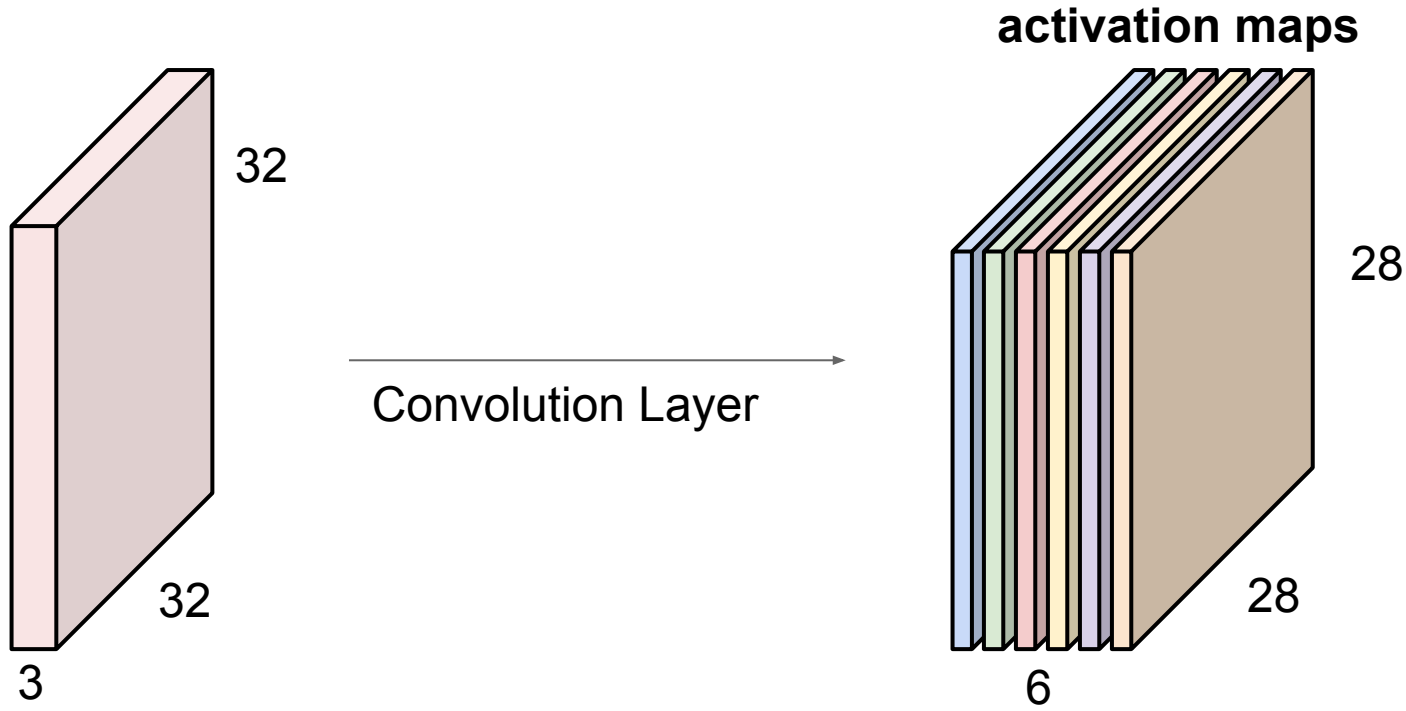


We processed [32x32x3] volume into [28x28x6] volume.

Q: how many parameters would this be if we used a fully connected layer instead?

A: (32*32*3)*(28*28*6) = **14.5M parameters**, **~14.5M multiplies**

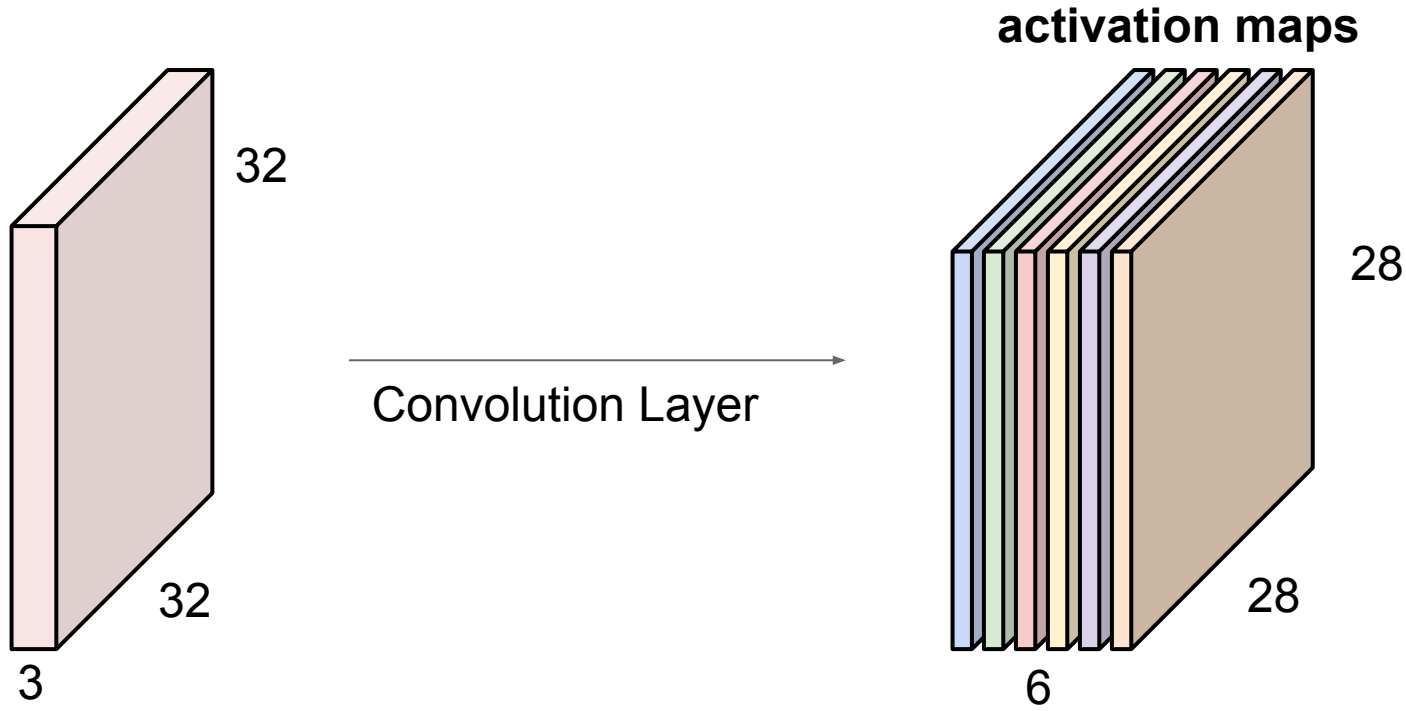
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We processed [32x32x3] volume into [28x28x6] volume.

Q: how many parameters are used instead?

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

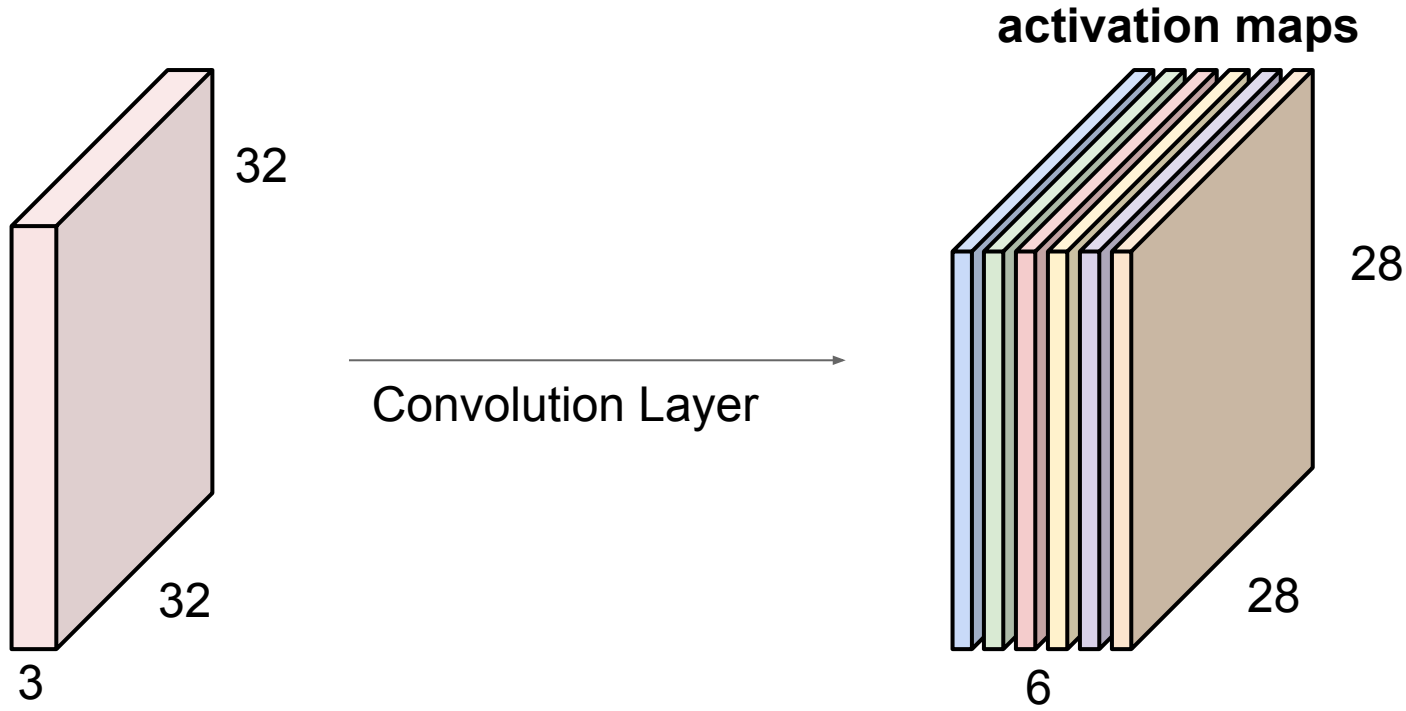


We processed [32x32x3] volume into [28x28x6] volume.

Q: how many parameters are used instead? --- And how many multiplies?

A: (5*5*3)*6 = **450 parameters**

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We processed [32x32x3] volume into [28x28x6] volume.

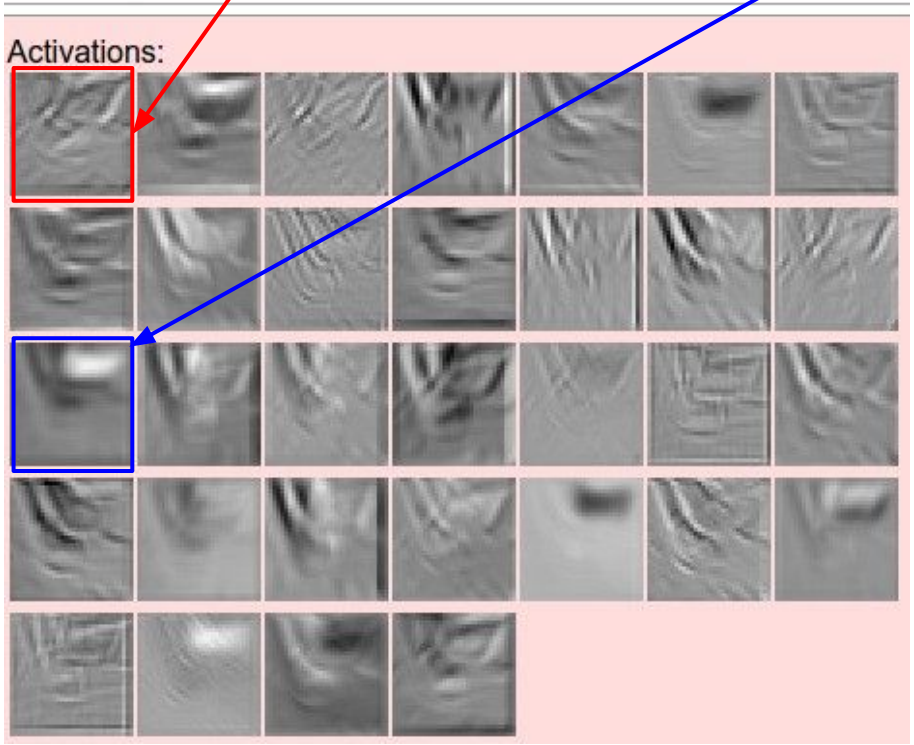
Q: how many parameters are used instead?

A: (5*5*3)*6 = **450 parameters**, (5*5*3)*(28*28*6) = **~350K multiplies**

Convolution summary

one filter =>
one activation map

example 5x5 filters
(32 total)

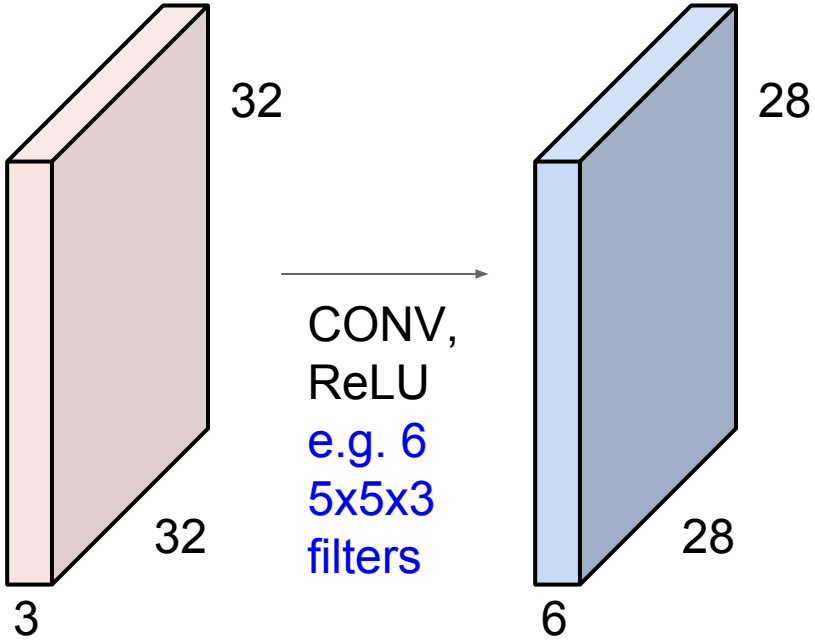


We call the layer convolutional because it is related to convolution of two signals:

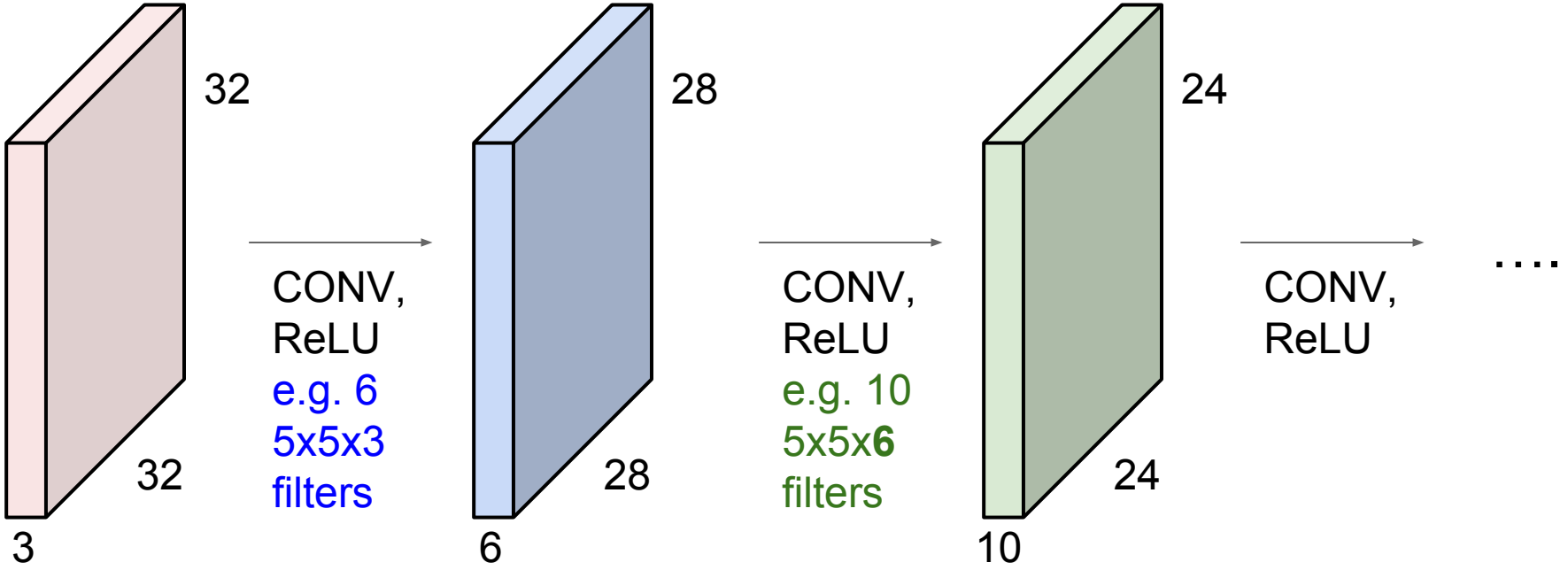
$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

elementwise multiplication and sum of a filter and the signal (image)

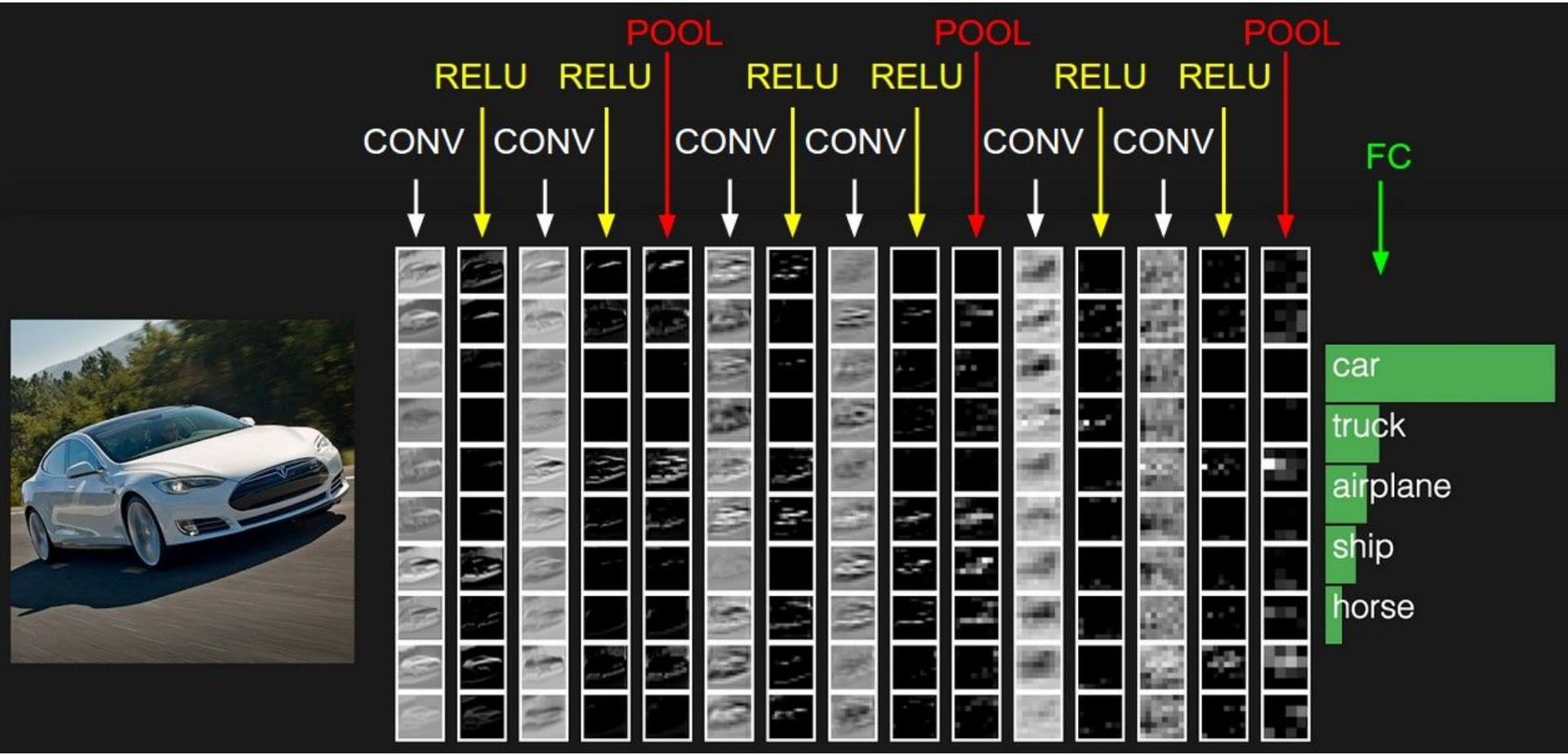
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions

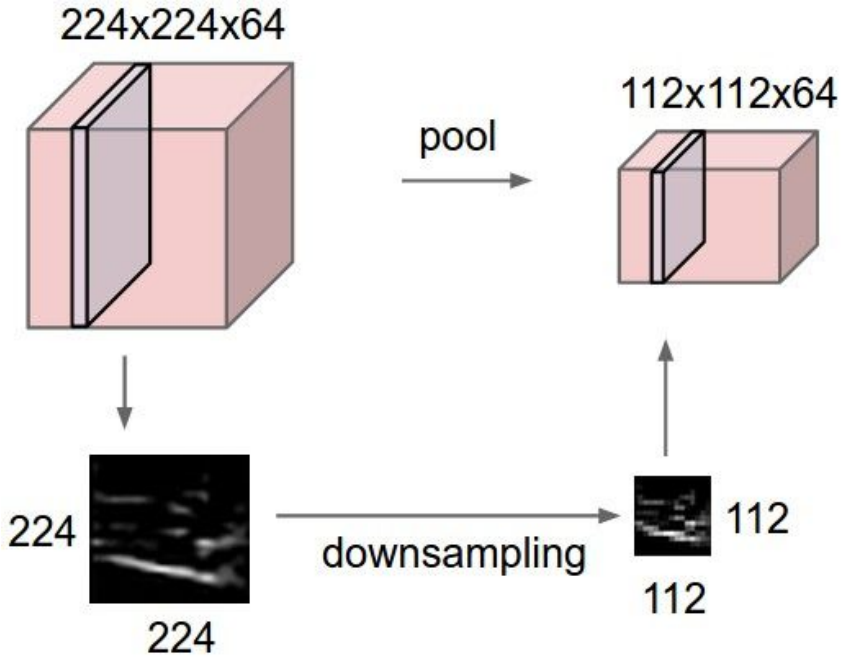


two more layers to go: POOL/FC

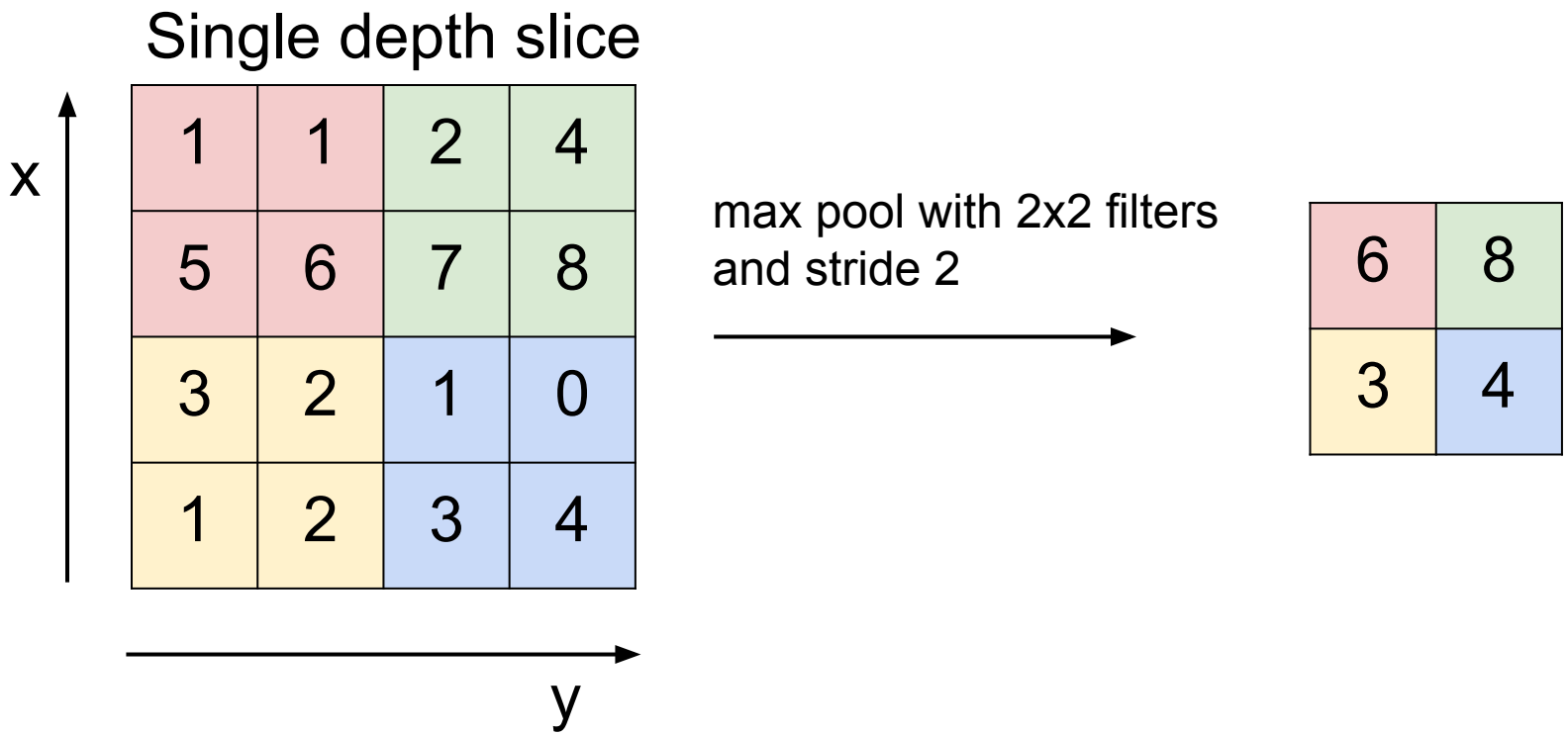


Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:

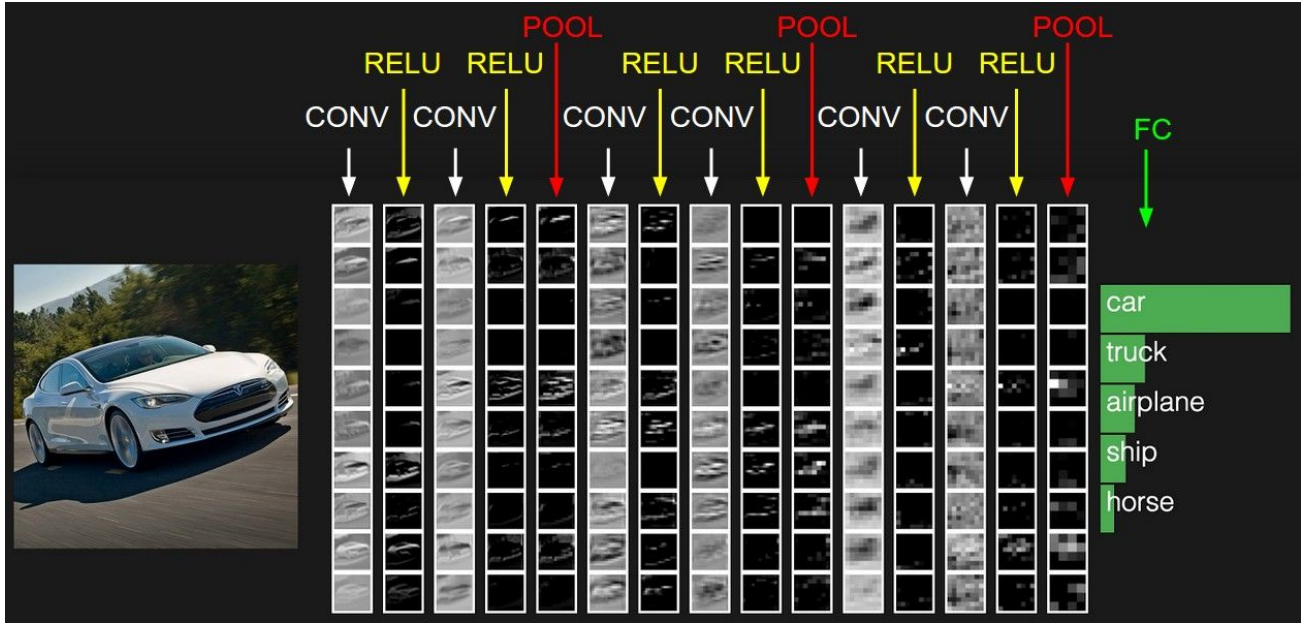


MAX POOLING


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

Fully Connected Layer (FC layer)

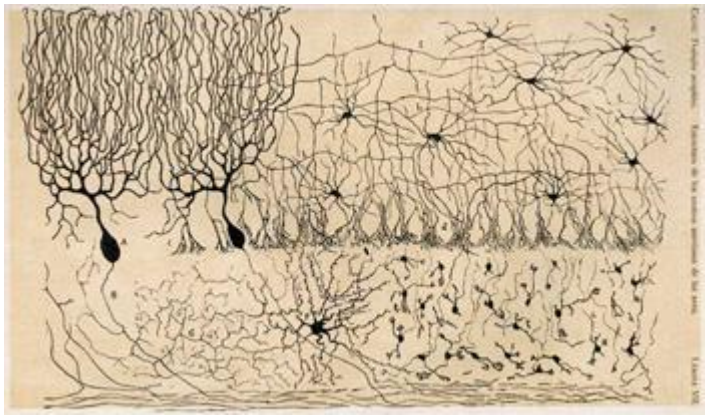
- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



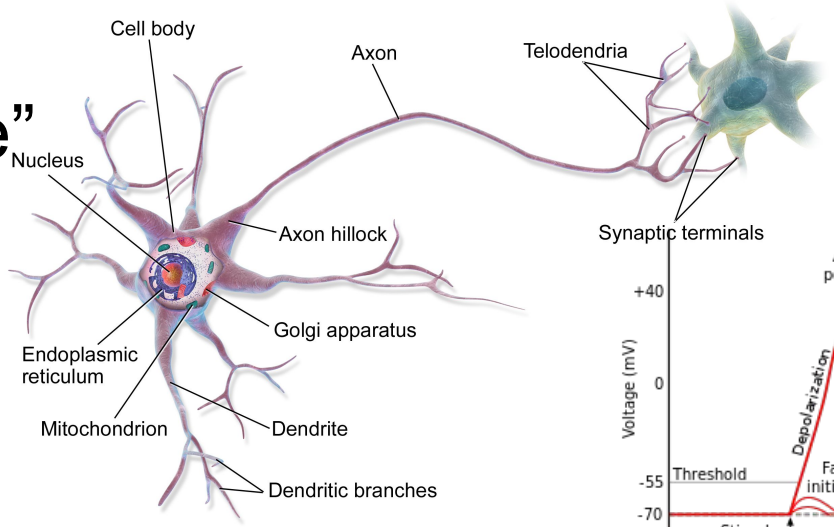
[ConvNetJS demo: training on CIFAR-10]

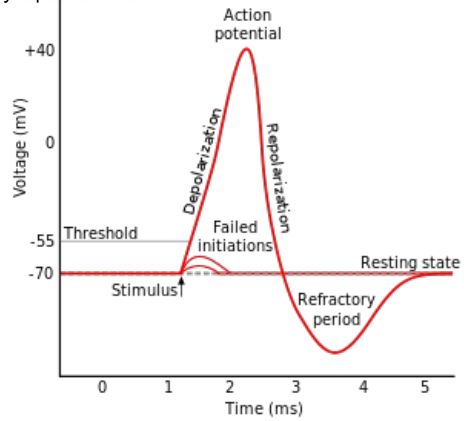
http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

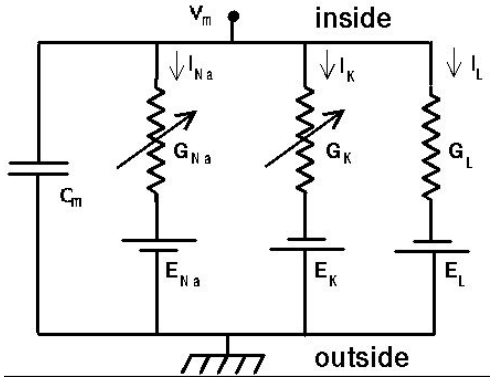
The “Neuron Doctrine”

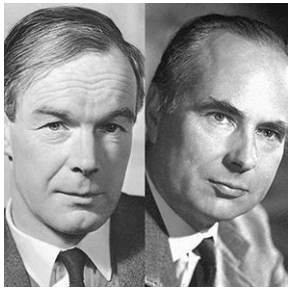


Ramón y Cajal's drawing of the cells of the chick cerebellum, from *Estructura de los centros nerviosos de las aves*, Madrid, 1905



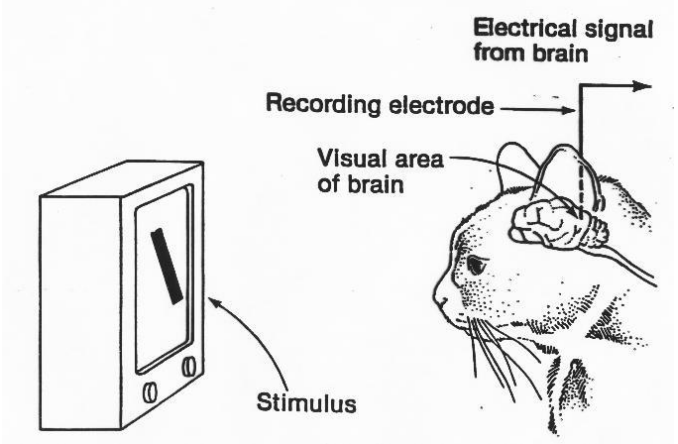




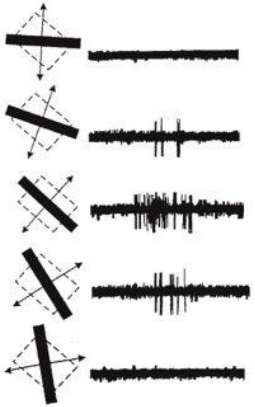


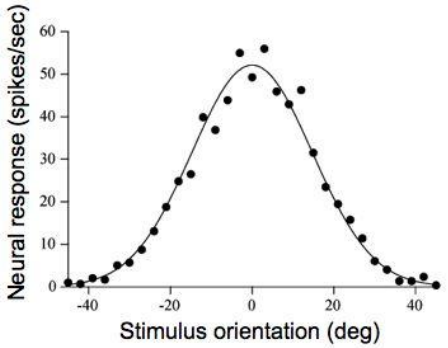
Hodgkin and Huxley

Hubel and Weisel experiment

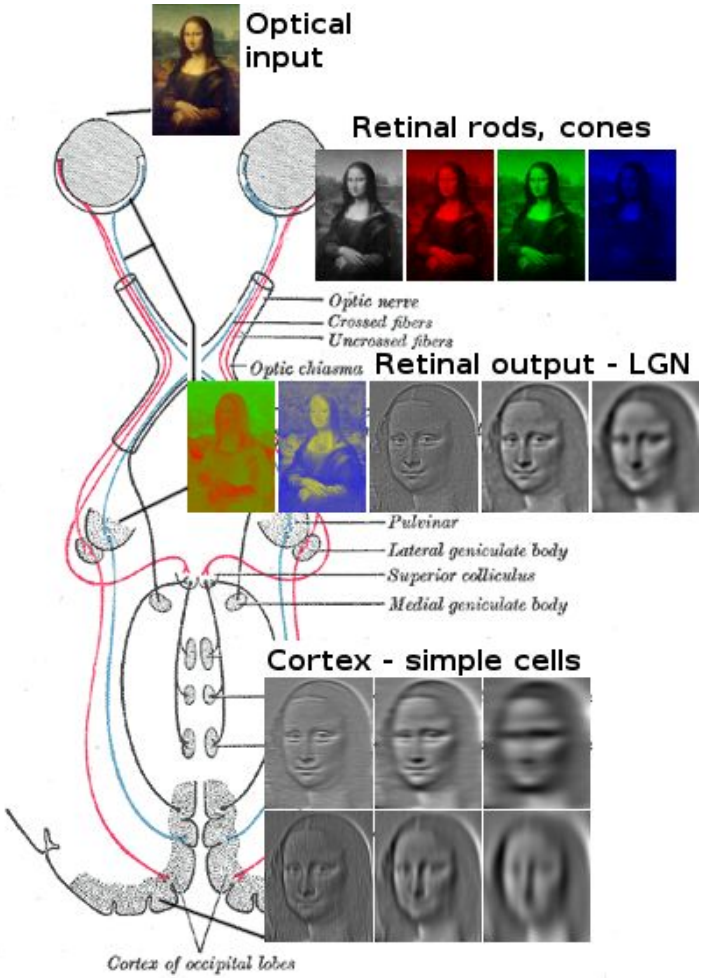


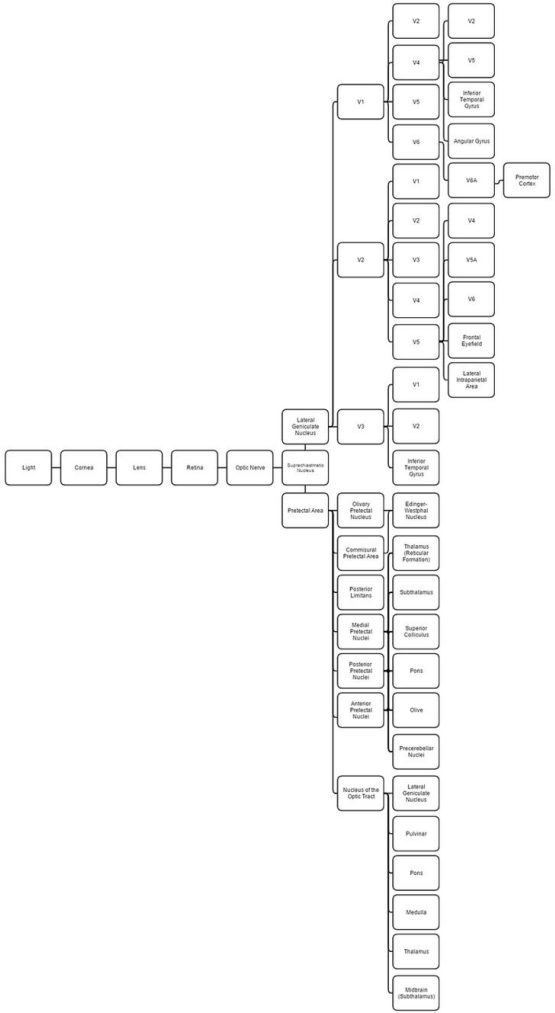
V1 physiology: orientation selectivity





Hubel & Wiesel, 1968

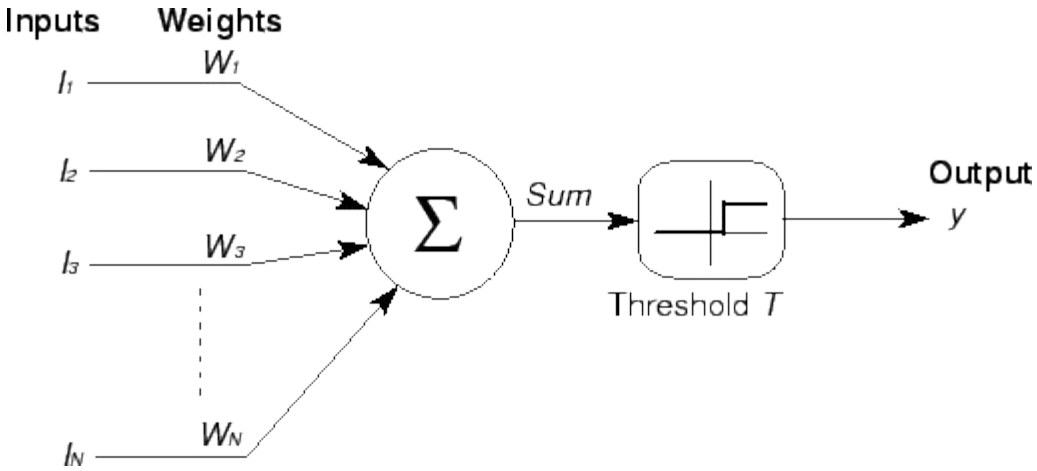

Optical input
Retinal rods, cones
Optic nerve
Crossed fibers
Uncrossed fibers
Optic chiasma
Retinal output - LGN
Pulvinar
Lateral geniculate body
Superior colliculus
Medial geniculate body
Cortex - simple cells
Cortex of occipital lobes

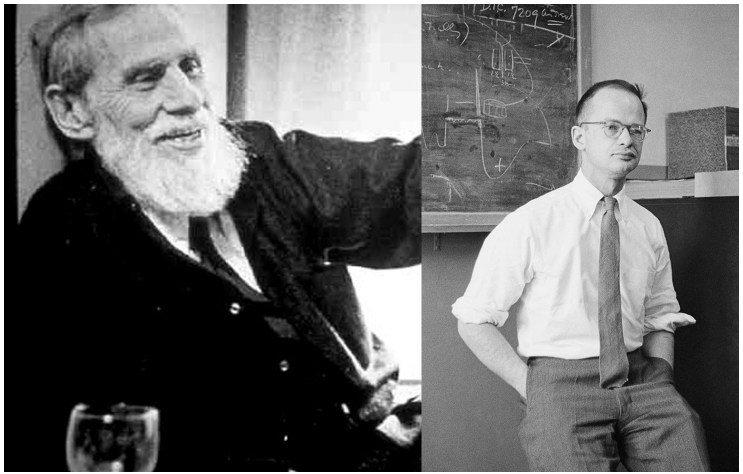

Light
Cornea
Lens
Retina
Optic Nerve
Suprachiasmatic Nucleus
Lateral Geniculate Nucleus
Pretectal Area
V1
V2
V3
V2
V4
V5
V6
V1
V2
V3
V4
V5
V1
V2
Inferior Temporal Gyrus
Olivary Pretectal Nucleus
Commisural Pretectal Area
Posterior Limitans
Medial Pretectal Nuclei
Posterior Pretectal Nuclei
Anterior Pretectal Nuclei
Nucleus of the Optic Tract
V2
V5
Inferior Temporal Gyrus
Angular Gyrus
V6A
Premotor Cortex
V4
V5A
V6
Frontal Eyefield
Lateral intraparietal Area
Edinger-Westphal Nucleus
Thalamus (Reticular Formation)
Subthalamus
Superior Colliculus
Pons
Olive
Precerebellar Nuclei
Lateral Geniculate Nucleus
Pulvinar
Pons
Medulla
Thalamus
Midbrain (Subthalamus)

Key ideas omitted

- Hebbian Learning
- Minsky’s Snark
- Willshaw Attractor Networks

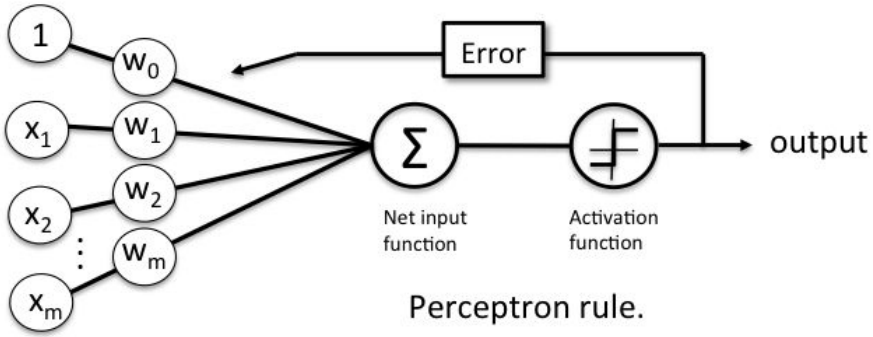
Mcculloch Pitts Artificial Neuron



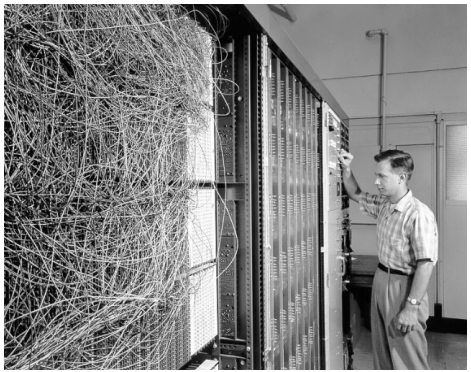


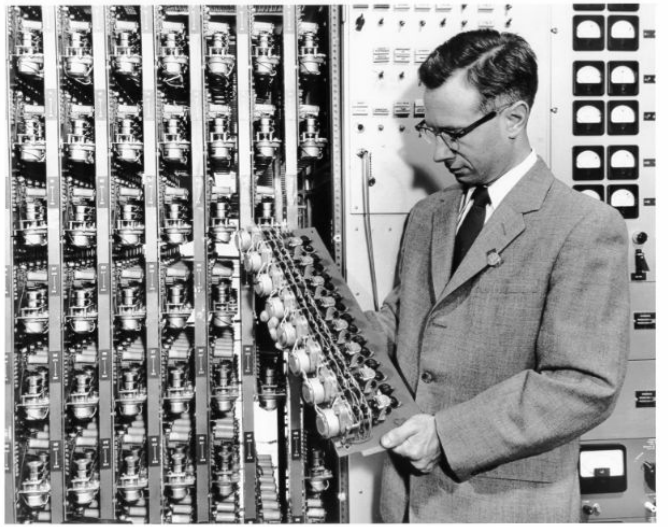
- Inputs binary, output is binary
- In principle, a network of such units can implement any boolean function
- Exercise: how can you create an OR gate? XOR?

Rosenblatt: learning law for artificial neural networks



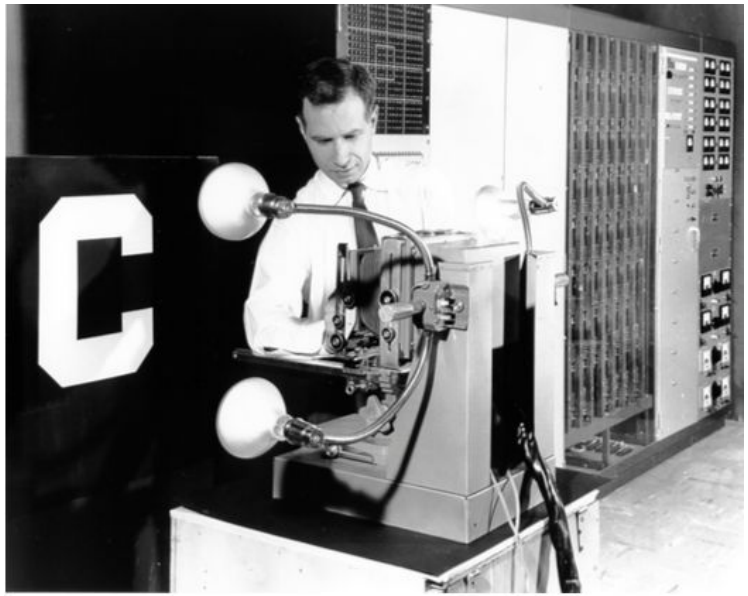
Perceptron rule.



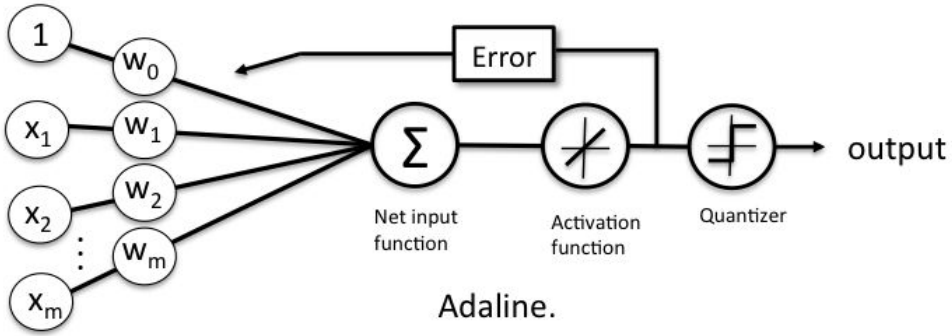


8 x 8 x 8 weights (where each weight is a Minsky “Snark” Neurocomputer

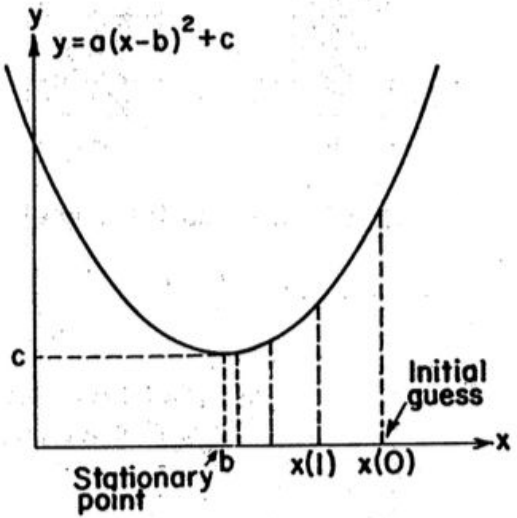
Rosenblatt: learning law for artificial neural networks



Widrow's ADALINE (LMS filter)

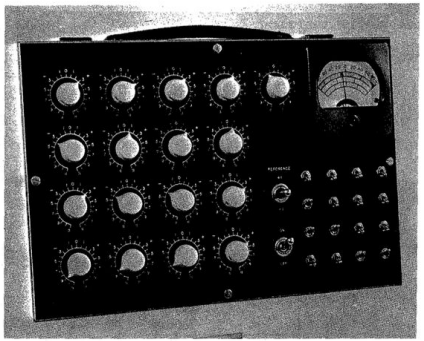


Adaline.



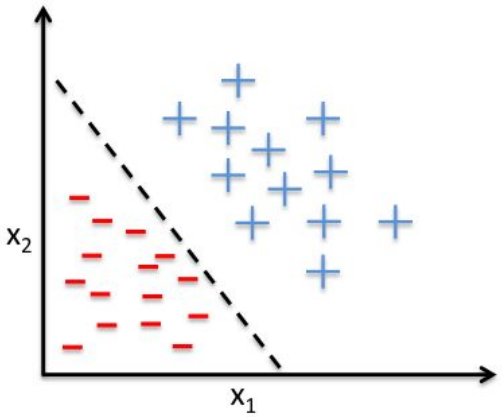
ADAPTIVE SWITCHING CIRCUITS

Bernard Widrow
and
Marcian E. Hoff

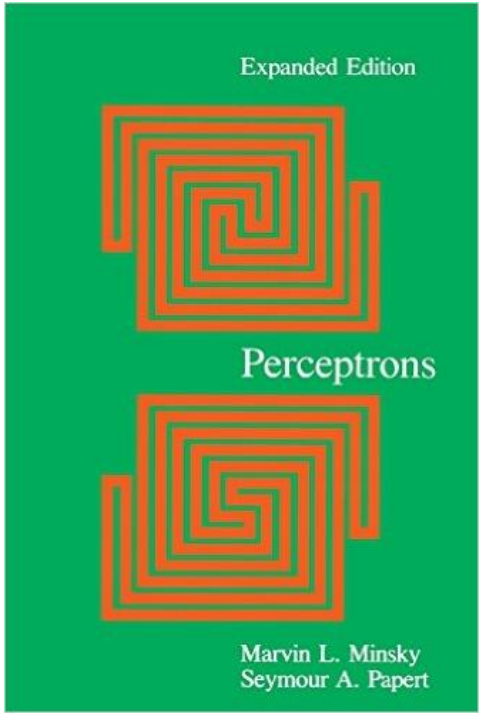


Adjust weights with potentiometers

Minsky and Papert's *Perceptrons*

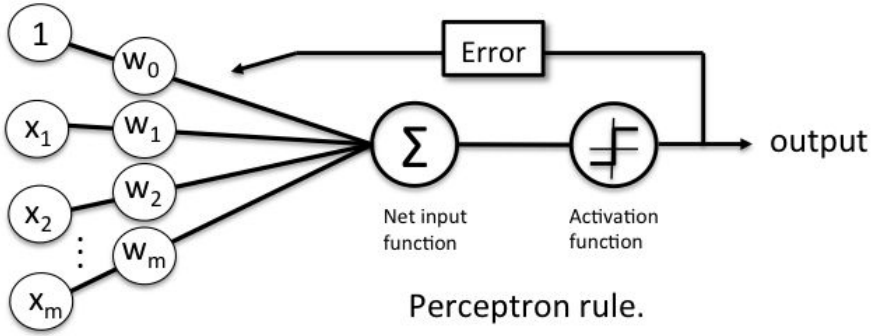


Example of a linear decision boundary for binary classification.



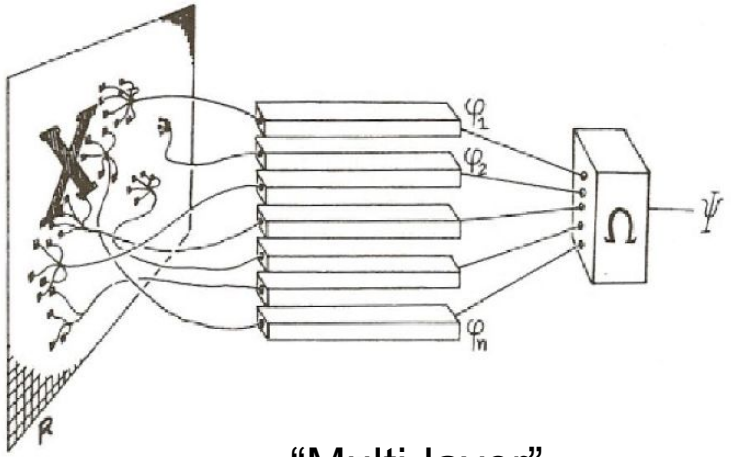
Famously showed that the XOR problem could not be solved and therefore kill this research for almost a decade

Rosenblatt misunderstanding?



“Single layer”

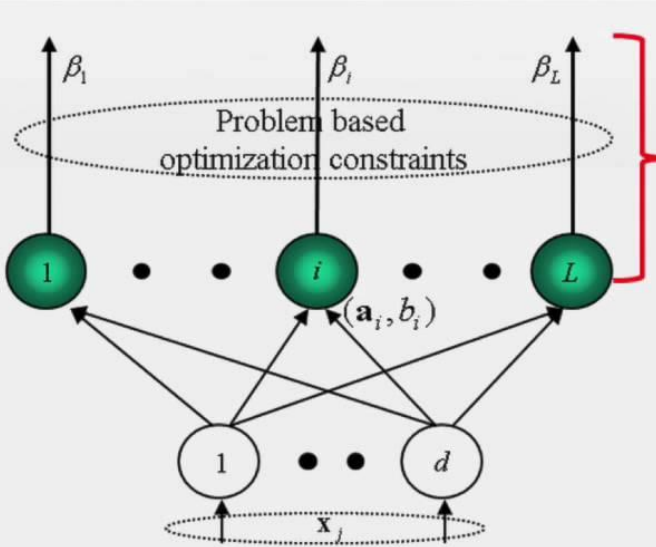
How it is taught in class and thought about



“Multi-layer”

What Rosenblatt was actually working on

“Extreme Learning Machines” : Rosenblatt’s idea?



Feature learning
Clustering
Regression
Classification

L Random Hidden Neurons (which need not be algebraic sum based) or other ELM feature mappings. Different type of output functions could be used in different neurons:

$$h_i(\mathbf{x}) = G_i(\mathbf{a}_i, b_i, \mathbf{x})$$

d Input Nodes

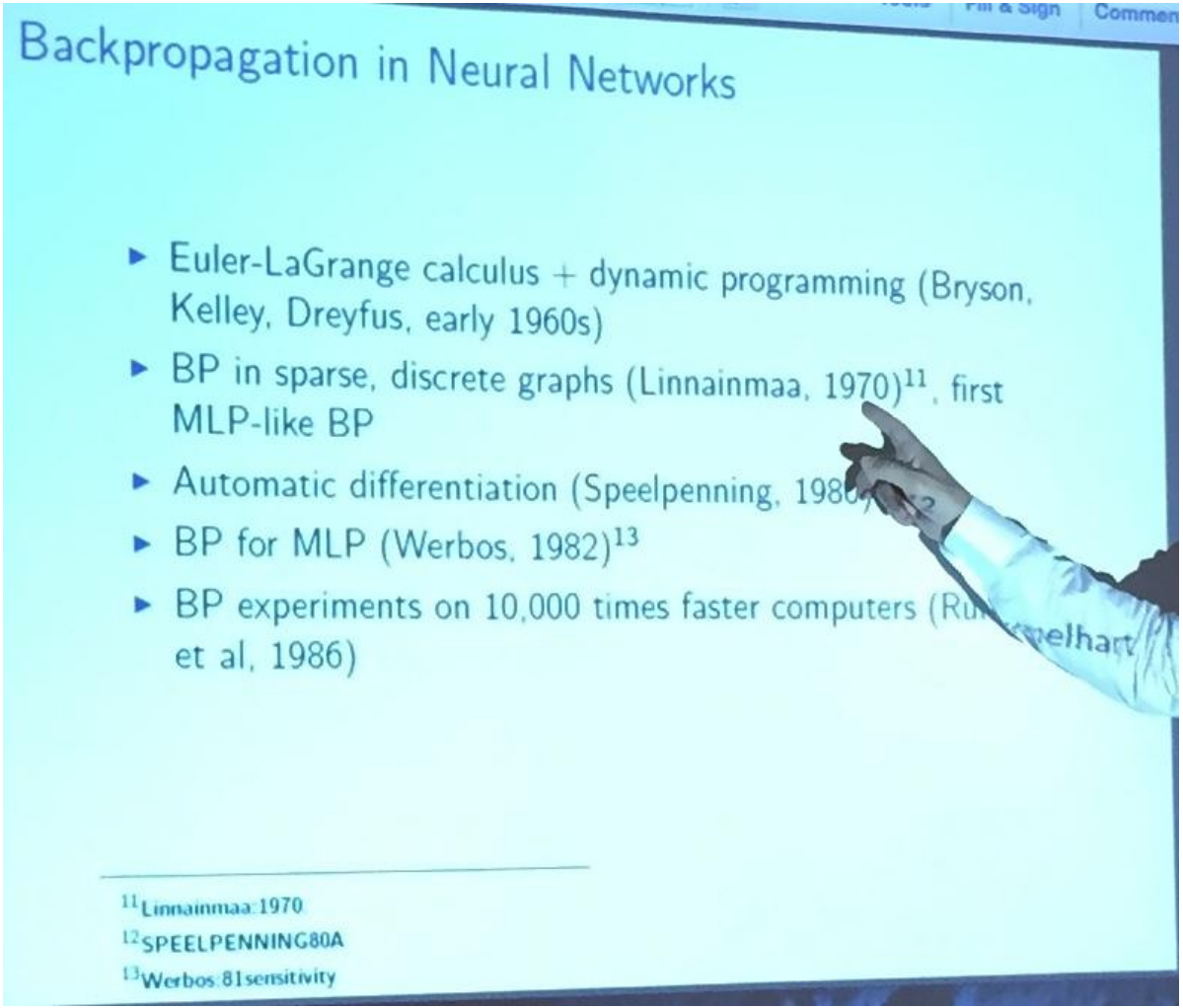
Extreme learning machines

INSTITUTE FOR NEURAL COMPUTATION
Technical Report #0403 2 July 2004
University of California, San Diego
Institute for Neural Computation
La Jolla, CA 92093-0523
inc2.ucsd.edu

Perceptrons

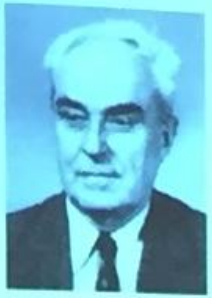
Robert Hecht-Nielsen
Computational Neurobiology, Institute for Neural Computation, ECE Department,
University of California, San Diego, La Jolla, California 92093-0407 USA rh-n@ucsd.edu

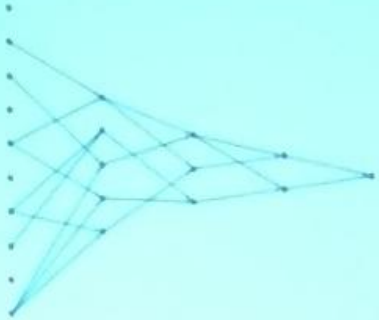
http://tdlc.ucsd.edu/documents/Perceptrons%20Monograph.pdf



Werbos, P.. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University, 1974

First Deep Learning





- Group Method of Data Handling (GMDH)[19,20]
- Deep multilayer networks of polynomial functions
- Layer-wise inductive training
- Learning regression models
- Sparse connections, pruning
- 8 layers already back in 1971[21]

[19] ivakhnenko1965

[20] ivakhnenko1968

[21] ivakhnenko1971

The 1980's: PDP group at UCSD

- Over several years in the 1980's, numerous scholars met at UCSD to to understand the mind (not solve AI necessarily)

- Most famous product is the "backpropagation algorithm"
  - Technically a "rediscovery," this led to solving the XOR problem and therefore showing that the limitations posed by Minsky and Papert's *Perceptrons* can be overcome
- *First remarketing efforts:*
  - Step 1. change "perceptron to multi-layer perceptron",
  - Step 2. …
  - Step 3. profit

letters to nature

Nature **323**, 533 - 536 (09 October 1986); doi:10.1038/323533a0

Learning representations by back-propagating errors

DAVID E. RUMELHART*, GEOFFREY E. HINTON† & RONALD J. WILLIAMS*

*Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
†Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA
†To whom correspondence should be addressed.

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

References

1. Rosenblatt, F. *Principles of Neurodynamics* (Spartan, Washington, DC, 1961).
2. Minsky, M. L. & Papert, S. *Perceptrons* (MIT, Cambridge, 1969).
3. Le Cun, Y. *Proc. Cognitiva* **85**, 599–604 (1985).
4. Rumelhart, D. E., Hinton, G. E. & Williams, R. J. in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Vol. 1: *Foundations* (eds Rumelhart, D. E. & McClelland, J. L.) 318–362 (MIT, Cambridge, 1986).

HNC: using ANN’s in real industry

- HNC started with building hardware (neurocomputers)
- Pivoted to focusing on software and algorithm development
- These efforts led to HNC going public in the mid 1990’s and making San Diego an analytics hub

The 1990’s

- In 1991, Sepp Hochreiter and Juergen Schmidhuber identified the fundamental problem of training multilayer with vanishing and exploding gradients

- Proposed a *pretraining* framework

- Proposed a novel architecture for additional constraints (LSTM)