# Computational Systems Biology
# Deep Learning in the Life Sciences
6.802 6.874 20.390 20.490 HST.506

# Lecture 2
# Optimizing Feedforward Networks

Manolis Kellis

Slides credit: David Gifford, Geoff Hinton, and more

# On tap today!

- What is Machine Learning
- Traditional Neural Networks
- How can we use gradients for optimization?
- How can we use gradients to train a deep neural network?
- What performance metrics should we use?
- How can we manage gradient optimization?
- How can we “regularize” a model to control parameter selection and thus model complexity?

# What is Machine Learning

# What is Machine Learning?

[Shalev-Shwartz and Ben-David, 2014]:

*"Learning is the process of converting experience into expertise or knowledge."*

[Mohri et al., 2012]:

*"Machine learning can be broadly defined as computational methods using experience to improve performance or to make accurate predictions."*

[Murphy, 2012]:

*"The goal of machine learning is to develop methods that can automatically detect patterns in data, and then to use the uncovered patterns to predict future data or other outcomes of interest."*

[Hastie et al., 2001]:

*"[...] state the learning task as follows: given the value of an input vector* ***x*** *, make a good prediction of the output* ***y*** *, denoted by* ***y^****"*

# What is Machine Learning?

A computer program is said to learn from **experience E**

with respect to some **class of tasks T**

and **performance measure P**,

if its performance at tasks in T, as measured by P, improves with experience E.

[Mitchell, 1997]

# What is Machine Learning?

A computer program is said to learn from
**experience E**

with respect to some
**class of tasks T**

and
**performance measure P,**

if its performance at tasks in T, as measured by P, improves with experience E.

[Mitchell, 1997]

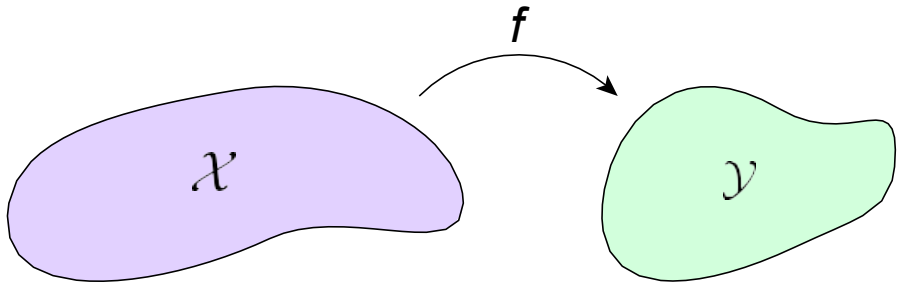
## Problem Set 1

- experience E: training set of images of handwritten digits with labels (training set)
- task T: classifying handwritten digits within new images (test set)
- performance measure P: percent of test set digits correctly classified in new images (test set)

# Linear Algebra and Machine Learning Notation

| | |
|---|---|
| $a, b, c_i$ | scalar (slanted, lower-case) |
| $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ | vector (bold, slanted, lower-case) |
| $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ | matrix (bold, slanted, upper-case) |
| $\mathbf{A}, \mathbf{B}, \mathbf{C}$ | tensor (bold, upright, upper-case) |
| $\mathrm{A}, \mathrm{B}, \mathrm{C}$ | set (calligraphic, slanted, upper-case) |
| $\mathrm{X}$ | input space or feature space dataset |
| $\boldsymbol{X}, \mathbf{X}$ | example matrix or tensor |
| $\boldsymbol{x}^{(i)}$ | $i$ th example of dataset, one row of $\boldsymbol{X}$ |
| $x_j^{(i)}, x_j$ | feature $j$ of example $\boldsymbol{x}^{(i)}$ |
| $\mathrm{Y}$ | label space |
| $\boldsymbol{y}^{(i)}$ | label of example $I$ |
| $\hat{\boldsymbol{y}}^{(i)}$ | predicted label of example $i$ |

# Terminology




Input $\boldsymbol{x} \in \mathrm{X}$:

- **features** (in machine learning)
- predictors (in statistics)
- independent variables (in statistics)
- regressors (in regression models)
- input variables
- covariates

Output $\boldsymbol{y} \in \mathrm{Y}$:

- **labels** (in machine learning)
- responses (in statistics)
- dependent variables (in statistics)
- regressand (in regression models)
- target variables

Training set $\mathrm{S}_{\text{training}} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{N} \in \{\mathrm{X}, \mathrm{Y}\}^{N}$, where $N$ is number of training examples

An example is a collection of features (and an associated label)

Training: use $\mathrm{S}_{\text{training}}$ to learn functional relationship $f : \mathrm{X} \rightarrow \mathrm{Y}$

# Terminology

$$f : \mathrm{X} \rightarrow \mathrm{Y}$$
$$f(\boldsymbol{x}; \boldsymbol{\theta}) = \hat{\boldsymbol{y}}$$

$\boldsymbol{\theta}$:

- **weights** and **biases** (intercepts)
- coefficients $\boldsymbol{\beta}$
- parameters

$f$:

- model
- hypothesis $h$
- classifier
- predictor
- discriminative models: P(Y|X)
- generative models: P(X, Y)

## Problem Set 1

$$\boldsymbol{x} \in [0, 1]^{784}$$
$$\hat{\boldsymbol{y}} \in [0, 1]^{10}$$
$$\boldsymbol{W} \in \mathrm{R}^{784 \times 10}$$
$$\boldsymbol{b} \in \mathrm{R}^{10}$$
$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}) = \varphi_{\text{softmax}}(\boldsymbol{W}^{\mathrm{R}} \boldsymbol{x} + \boldsymbol{b})$$

# Data in PS1

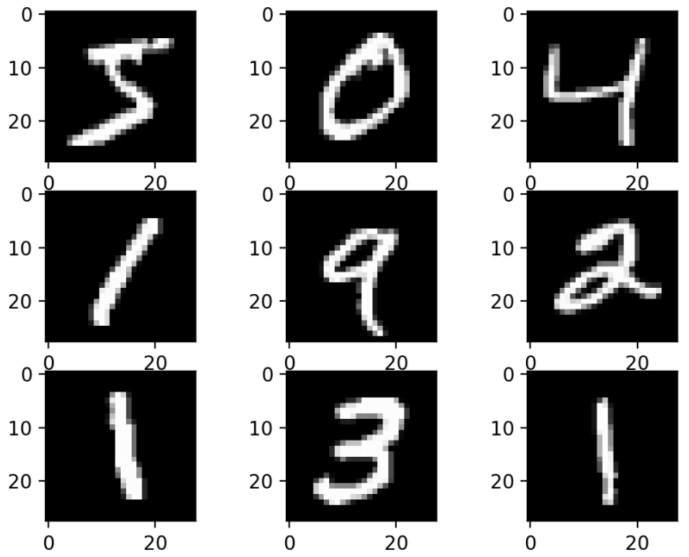
## Problem Set 1

input space:
$X = \{0, 1, \ldots, 255\}^{28\times 28}$

after rescaling:
$X^{I} = [0, 1]^{28\times 28}$

after flattening:
$X^{II} = [0, 1]^{784}$

**Classification**



# Data in PS1

## Problem Set

input space:
$X = \{0, 1, \ldots, 255\}^{28\times 28}$

after rescaling:
$X^{I} = [0, 1]^{28\times 28}$

after flattening:
$X^{II} = [0, 1]^{784}$

$$\overbrace{\boldsymbol{x}^{(i)} \in \mathcal{X}}$$

$$\begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ 28 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \cdots & 28 \end{array} \\ \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,28} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,28} \\ \vdots & \vdots & \ddots & \vdots \\ x_{28,1} & x_{28,2} & \cdots & x_{28,28} \end{bmatrix} \end{array}$$

integer-encoded label space:
$Y_i = \{0, 1, \ldots, 9\}$

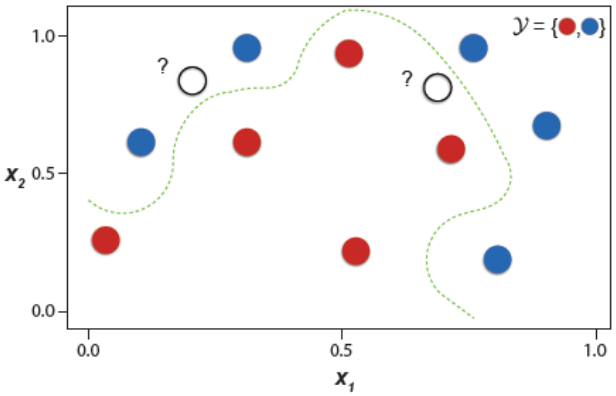
one-hot-encoded label space:
$Y_h = [0, 1]^{10}$

$$\overbrace{\boldsymbol{y}^{(i)} \in \mathcal{Y}_h}$$

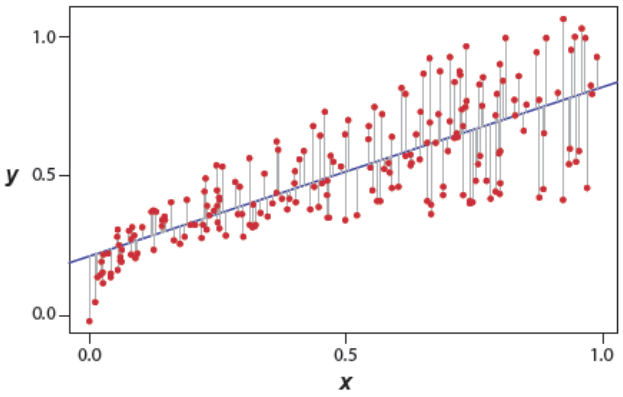
$$\begin{array}{cccc} 1 & 2 & \cdots & 10 \end{array}$$
$$\begin{bmatrix} y_1 & y_2 & \cdots & y_{10} \end{bmatrix}$$

# Types of Machine Learning

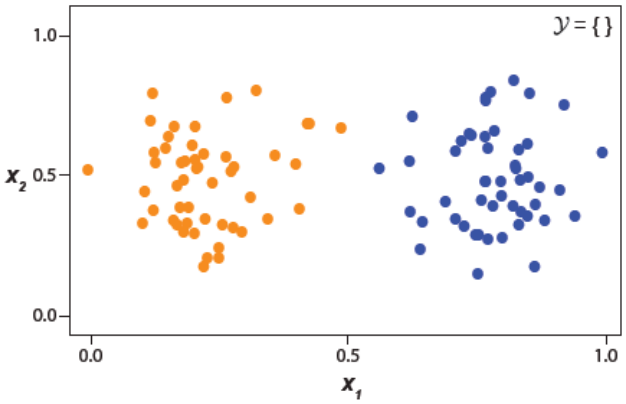
**Classification**




**Regression**




**Unsupervised learning**




| | |
|---|---|
| $Y \neq \emptyset$ | supervised or semi-supervised learning |
| $Y = R$ | Regression |
| $Y = R^K, K > 1$ | multivariate regression |
| $Y = \{0, 1\}$ | binary classification |
| $Y = \{1, \ldots, K\}$ | multi-class classification (integer encoding) |
| $Y = \{0, 1\}^K, K > 1$ | multi-label classification |
| $Y = \emptyset$ | unsupervised learning |

# Types of Machine Learning

## Problem Set 1

- task: every $\boldsymbol{x}$ has an associated $\boldsymbol{y}$ $\Rightarrow$ supervised learning
- subtask: $Y = \{0, \ldots, 9\}$ $\Rightarrow$ multi-class classification
- method: we use softmax regression (also known as multinomial logistic regression) as multi-class classification method

# Objective function

# Objective functions

An **objective function** J (Θ) is the function that you optimize when training machine learning models. It is usually in the form of (but not limited to) one or combinations of the following:

**Loss / cost / error function** $L(\hat{y}, y)$:

Classification

- 0-1 loss
- cross-entropy loss
- hinge loss

Regression

- mean squared error (MSE, $L_2$norm)
- mean absolute error (MAE, $L_1$norm)
- Huber loss (hybrid between $L_1$ and $L_2$ norm)

Probabilistic inference

- Kullback-Leibler divergence (KL divergence)

**Likelihood function / posterior**:

- negative log-likelihood (NLL) in maximum likelihood estimation (MLE)
- posterior in maximum a posteriori estimation (MAP)

**Regularizers and constraints**

- $L_1$ regularization $||\Theta||_1 = \lambda \sum_{i=1}^{N} |\theta_i|$
- $L_2$ regularization $||\Theta||_2^2 = \lambda \sum_{i=1}^{N} \theta_i^2$
- max-norm $||\Theta||_2^2 \leq c$

# Loss functions for classification

**0-1 loss**:

$$\mathcal{L}_{0\text{-}1}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=1}^{N} \mathbb{1}([\hat{y}^{(i)}] \neq y^{(i)}) = \sum_{i=1}^{N} \begin{cases} 1, & \text{for } \hat{y}^{(i)} \neq y^{(i)} \\ 0, & \text{for } \hat{y}^{(i)} = y^{(i)} \end{cases}$$

where $[x]$ is the function that rounds $x$ to the nearest integer.

**Binary cross-entropy loss** (for binary classification):

$\mathcal{L}_{\text{BCE}}$ = NLL (Negative Log Likelihood)
Likelihood is defined using the Bernoulli distribution

$$p(\hat{y}^{(i)}, y^{(i)}) = (\hat{y}^{(i)})^{y^{(i)}}(1 - \hat{y}^{(i)})^{(1-y^{(i)})}$$

$$\begin{aligned} \mathcal{L}_{\text{BCE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) &= \sum_{i=1}^{N} -y^{(i)} \log(\hat{y}^{(i)}) - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \\ &= \sum_{i=1}^{N} \begin{cases} -\log(\hat{y}^{(i)}), & \text{for } y^{(i)} = 1 \\ -\log(1 - \hat{y}^{(i)}), & \text{for } y^{(i)} = 0 \end{cases} \end{aligned}$$

# Loss functions for classification

**Binary cross-entropy loss** (for binary classification):

$$\mathcal{L}_{\text{BCE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=1}^{N} -y^{(i)} \log(\hat{y}^{(i)}) - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

$$= \sum_{i=1}^{N} \begin{cases} -\log(\hat{y}^{(i)}), & \text{for } y^{(i)} = 1 \\ -\log(1 - \hat{y}^{(i)}), & \text{for } y^{(i)} = 0 \end{cases}$$

| $\boldsymbol{y}$ | $\hat{\boldsymbol{y}}$ | $[\hat{\boldsymbol{y}}]$ | $\mathcal{L}_{0\text{-}1}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ | $\mathcal{L}_{\text{BCE}}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ |
|---|---|---|---|---|
| $[1, 0, 0]$ | $[0.9, 0.2, 0.4]$ | $[1, 0, 0]$ | 0 | 0.84 |
| $[1, 1, 0]$ | $[0.6, 0.4, 0.1]$ | $[1, 0, 0]$ | 1 | 1.53 |
| $[1, 0, 1]$ | $[0.1, 0.7, 0.3]$ | $[0, 1, 0]$ | 3 | 4.71 |

# Loss functions for classification

## Problem Set 1

**Categorical cross-entropy loss** (for multi-class classification with $K$ classes):

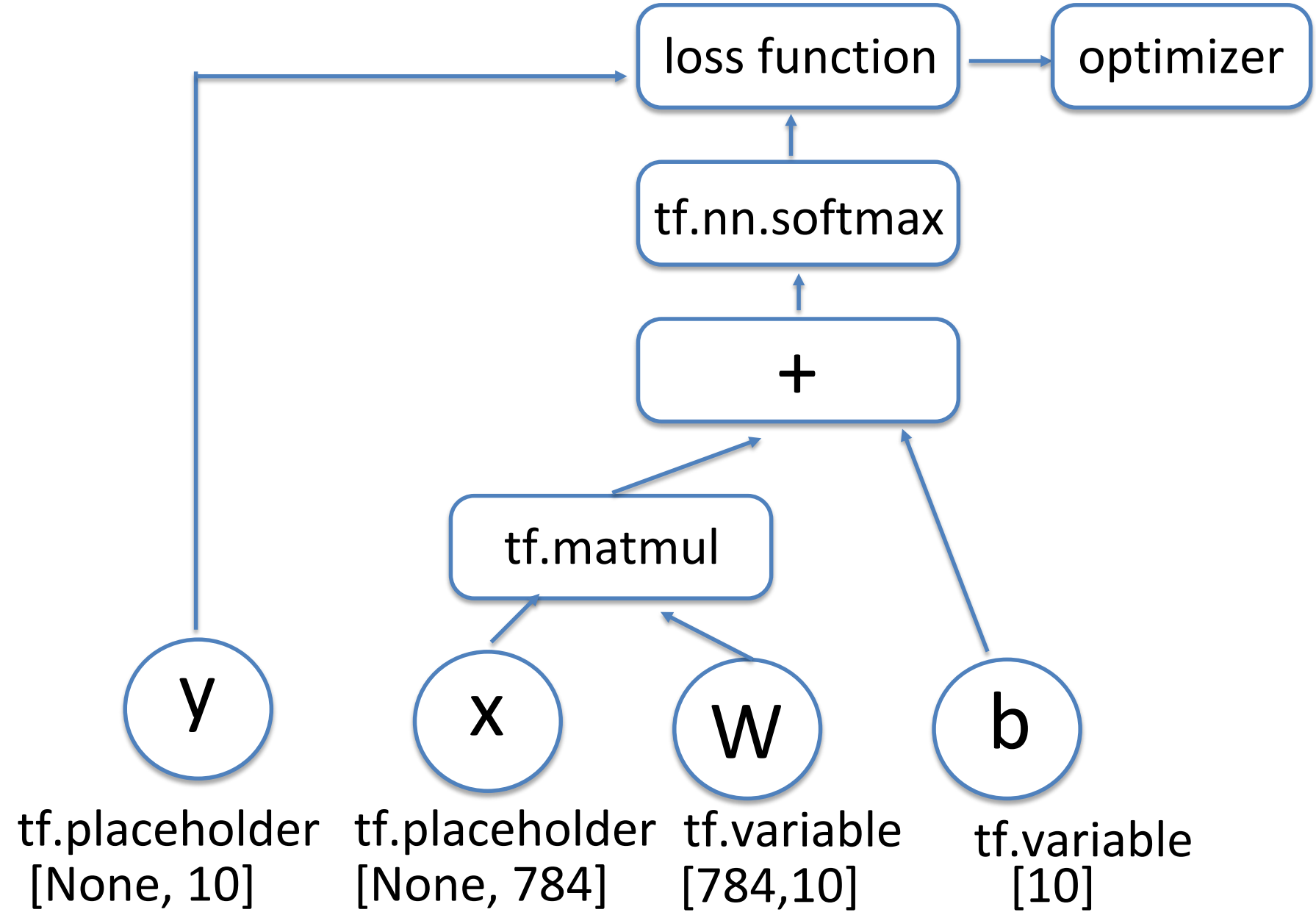
$$\mathcal{L}_{\text{CCE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{i=1}^{N} \sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)}),$$

where $\hat{y}_j^{(i)} = \dfrac{\exp(z_j^{(i)})}{\sum_{k=1}^{K} \exp(z_k^{(i)})}$ if softmax is used

note: $y_j{}^{(i)} = 1$ only if $\boldsymbol{x}^{(i)}$ belongs to class $j$ and otherwise $y_j{}^{(i)} = 0$

Probabilistic interpretation:
$L_{CCE}$ = NLL, if likelihood is defined using the categorical distribution

# Problem Set 1 Structure



# Loss functions for regression

**Mean squared error**:

$$\mathcal{L}_{\text{MSE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - \hat{y}^{(i)})^2$$

Probabilistic interpretation:
$L_{MSE}$ = NLL, under the assumptation that the noise is normally distributed, with constant mean and variance

**Mean absolute error**:

$$\mathcal{L}_{\text{MAE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{N} \sum_{i=1}^{N} |y^{(i)} - \hat{y}^{(i)}|$$

| $\boldsymbol{y}$ | $\hat{\boldsymbol{y}}$ | $\mathcal{L}_{\text{MSE}}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ | $\mathcal{L}_{\text{MAE}}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ |
|---|---|---|---|
| $[3.2, 1.2, 0.3]$ | $[3.1, 1.3, 0.4]$ | 0.01 | 0.1 |
| $[2.1, 0.1, -5.1]$ | $[2.0, -0.1, 1.2]$ | 13.25 | 2.2 |
| $[-0.1, 3.1, 0.5]$ | $[0.1, 3.3, -0.5]$ | 0.36 | 0.47 |

# Empirical risk minimization

**Expected risk** (loss) associated with hypothesis $h(\boldsymbol{x})$:

$$\mathcal{R}_{\text{exp}}(h) = \mathbb{E}(\mathcal{L}(h(\boldsymbol{x}), \boldsymbol{y})) = \int_{\mathcal{X}\times\mathcal{Y}} \mathcal{L}(h(\boldsymbol{x}), \boldsymbol{y}) p(\boldsymbol{x}, \boldsymbol{y}) d\boldsymbol{x} d\boldsymbol{y}$$

Minimize $R_{exp}(h)$ to find optimal hypothesis $h$:

$$h = \underset{h \in F}{\text{argmin}}\ R_{exp}(h)$$

Problem:

- distribution $p(\boldsymbol{x}, \boldsymbol{y})$ unknown
- F is too large (set of all functions from X to Y)

# Empirical risk minimization

**Empirical risk** associated with hypothesis $h(\boldsymbol{x})$:

$$\mathcal{R}_{\text{emp}}(h) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(h(\boldsymbol{x}^{(i)}), \boldsymbol{y}^{(i)})$$

Minimize $R_{emp}(h)$ to find $\hat{h}$:

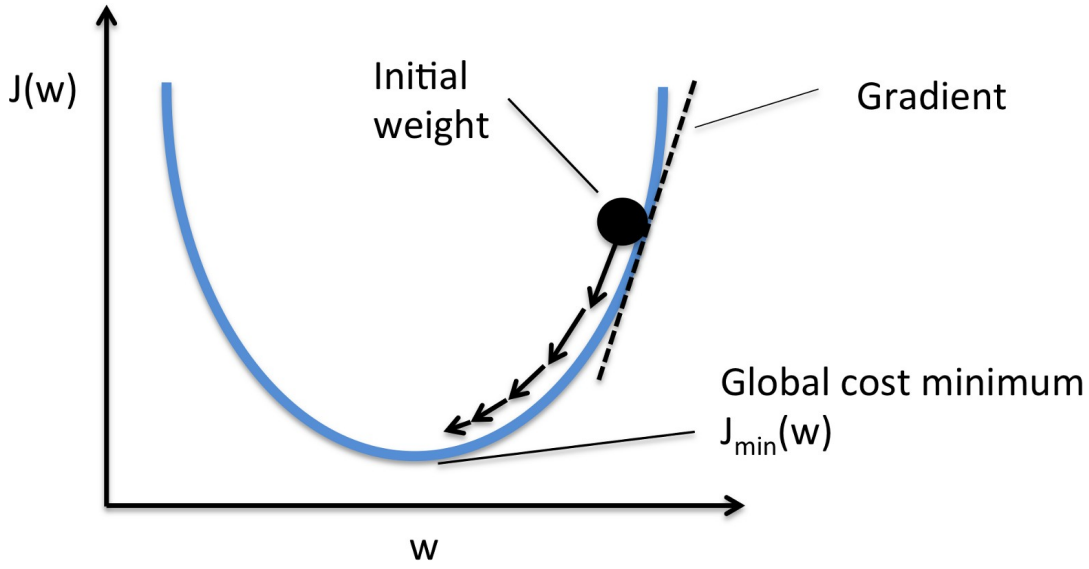
$$\hat{h} = \underset{h \in \mathrm{H}}{\operatorname{argmin}}\ R_{emp}(h)$$

In practice:

- instead of $p(\boldsymbol{x}, \boldsymbol{y})$, we use training set $S_{training}$
- instead of F, we use $H \subset F$, e.g., all polynomials of degree 5

# Optimizing objective function

**Gradient descent**




- initialize model parameters $\theta_0, \theta_1, \ldots, \theta_m$
- repeat until converge, for all $\theta_i$

$$\theta_i^t \leftarrow \theta_i^{t-1} - \lambda \frac{\partial}{\partial \theta_i^{t-1}} J(\Theta),$$

where the objective function $J(\Theta)$ is evaluated over all training data $\{(\mathbf{X}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N}$.

## Problem Set 1

**Stochastic Gradient Descent (SGD)**: in each step, randomly sample a mini-batch from the training data and update the parameters using gradients calculated from the mini-batch only.

# Training and Evaluation

# Training, validation, test sets

**Training set ($S_{training}$):**

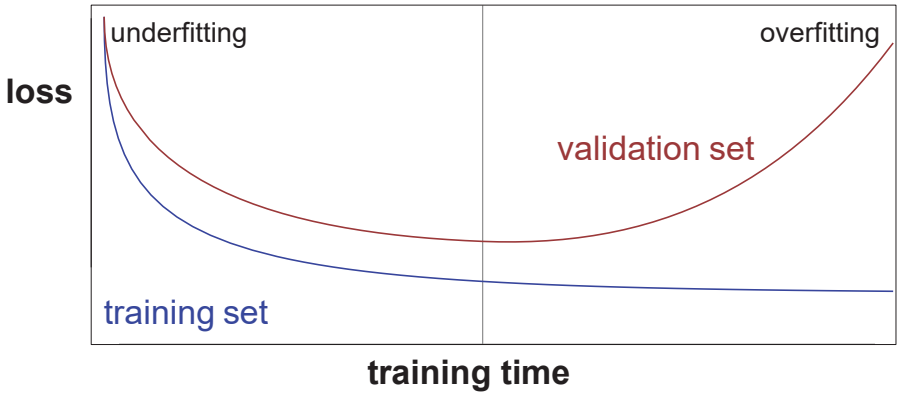
- set of examples used for learning
- usually 60 - 80 % of the data

**Validation set ($S_{validation}$):**

- set of examples used to tune the model hyperparameters
- usually 10 - 20 % of the data

**Test set ($S_{test}$):**

- set of examples used only to assess the performance of fully-trained model
- after assessing test set performance, model must not be tuned further
- usually 10 - 30 % of the data



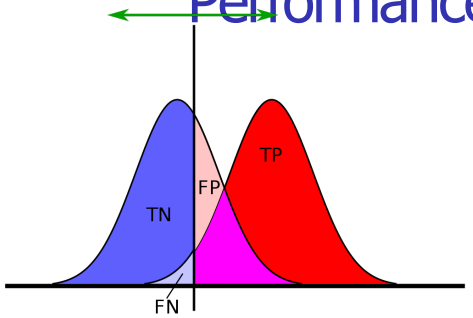
# Confusion matrix and derived metrics

| | | True condition | | |
|---|---|---|---|---|
| | Total population | Condition positive | Condition negative | $\text{Accuracy} = \frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$ |
| **Predicted condition** | Predicted condition positive | **True positive**, Power | **False positive**, Type I error | $\text{Precision} = \frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition positive}}$ |
| | Predicted condition negative | **False negative**, Type II error | **True negative** | |
| | | $\text{Recall, Sensitivity} = \frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$ | $\text{Specificity} = \frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$ | $F_1 \text{ score} = \frac{1}{\frac{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}{2}}$ |

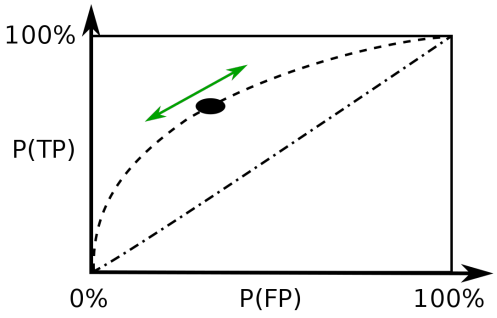
## Problem Set 1

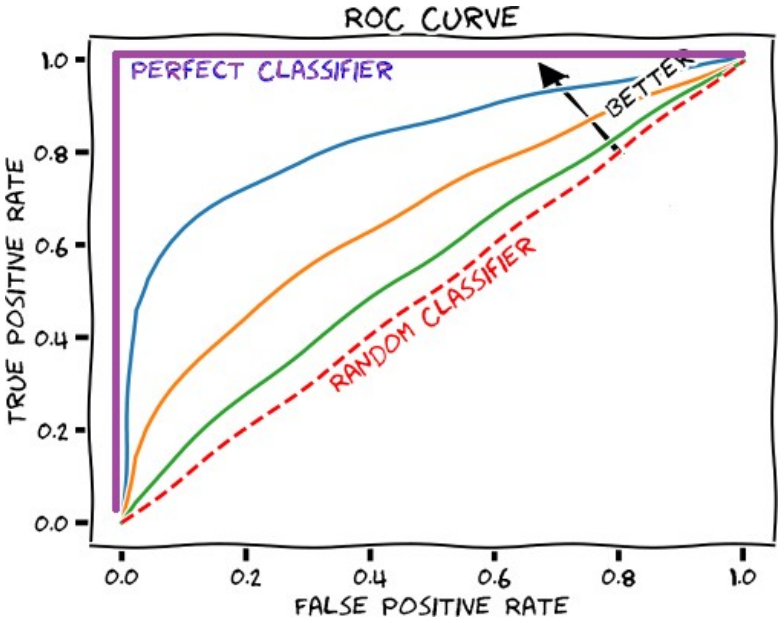
**Accuracy:** proportion of true predictions - (TP + TN) / (TP + FP + TN + FN)

# Receiver Operating Characteristic (ROC) Performance




| | |
|---|---|
| TP | FP |
| FN | TN |




## Area Under the ROC Curve (AuROC)

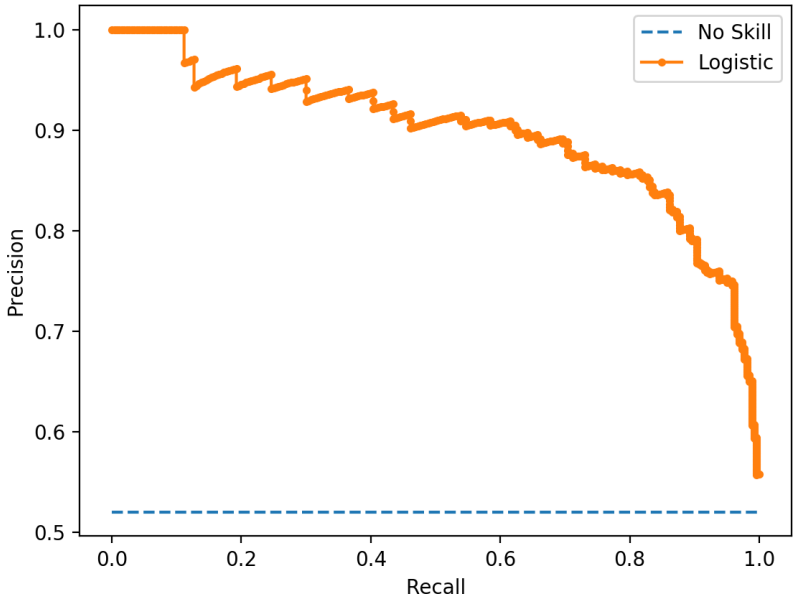
AuROC is a common metric for comparing classification methods

TPR = TP / (TP + FN)

FPR = FP / (FP + TN)

**Problematic when we have an unbalanced dataset (example more positives than negatives)**

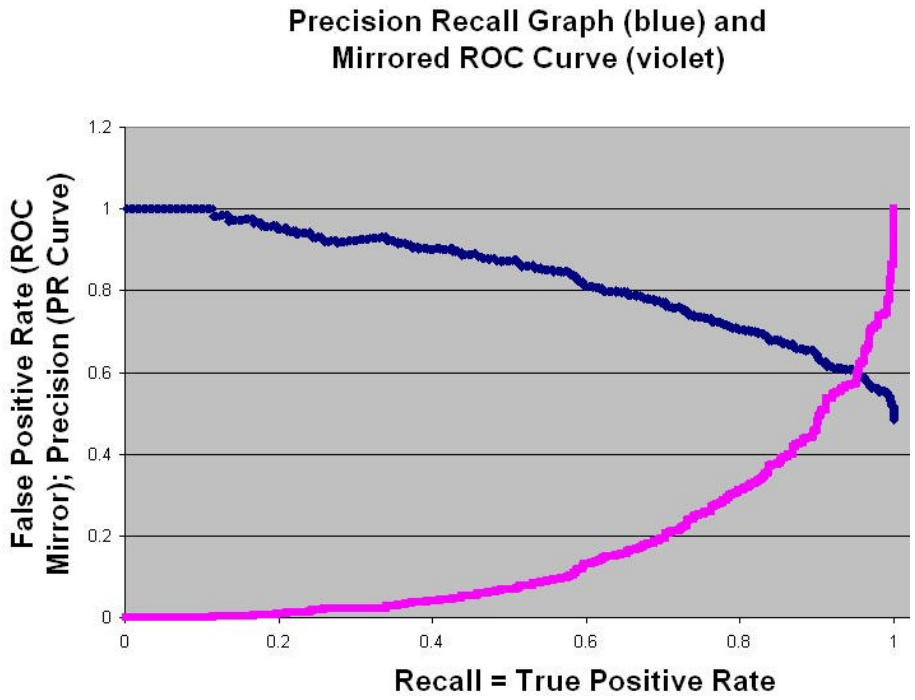
# Precision Recall Curve (PRC) Performance




## Area Under the PRC (AuPRC)

Precision = PPV = TP / (TP + FP) = 1 - FDR
Recall = TPR = TP / (TP + FN)
**Useful when datasets are unbalanced**

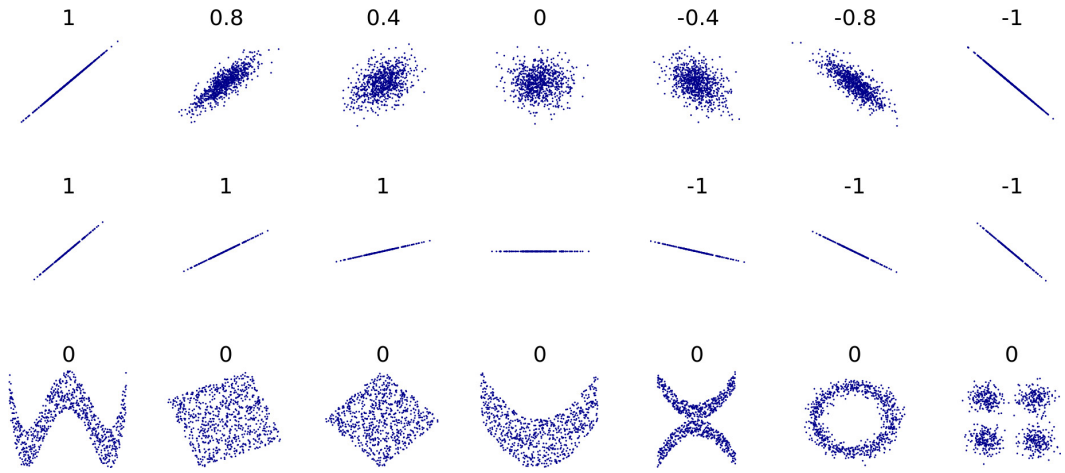
# ROC and PRC curves are complementary




## Recall

FPR = FP / (FP + TN)
Precision = PPV = TP / (TP + FP) = 1 - FDR
Recall = TPR = TP / (TP + FN)

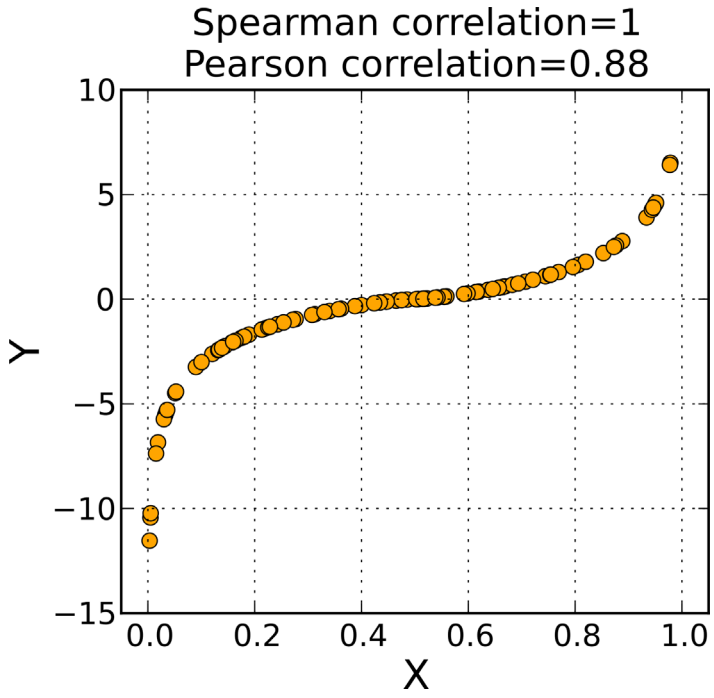
# Regression Metric 1 - Pearson Correlation




**Pearson correlation coefficient is $r$. $r^2$ is the fraction of linearly explained variance**

$$r = \frac{(x - \bar{x})}{||x||} \cdot \frac{(y - \bar{y})}{||y||}$$

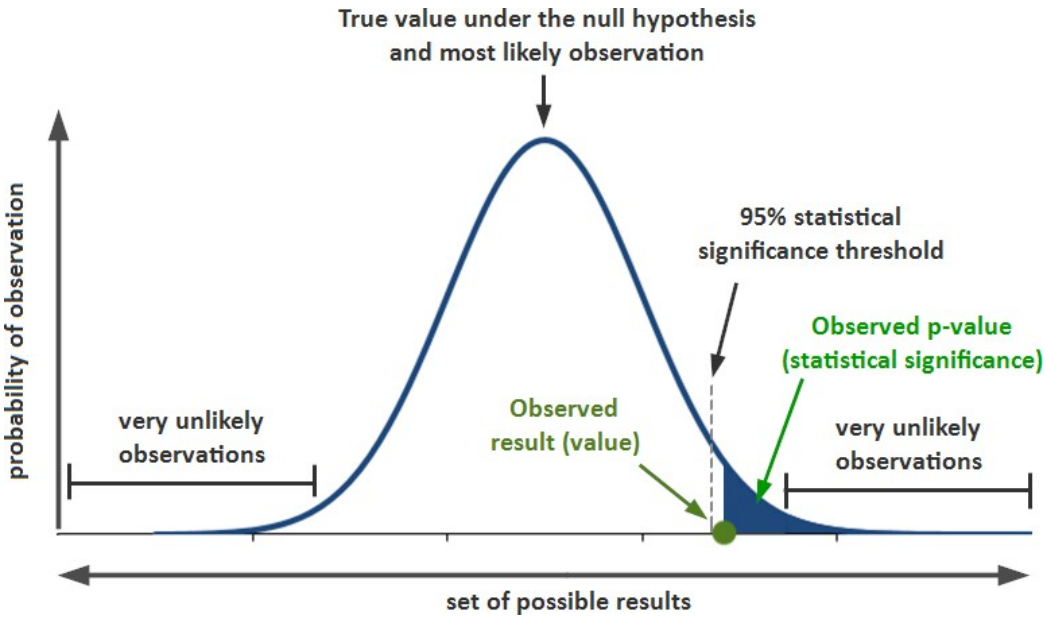
# Regression Metric 2 - Spearman Rank Correlation




## Pearson correlation of observation ranks

For ties assign fractional ranks by average rank in ascending order

# Correlation significance tests



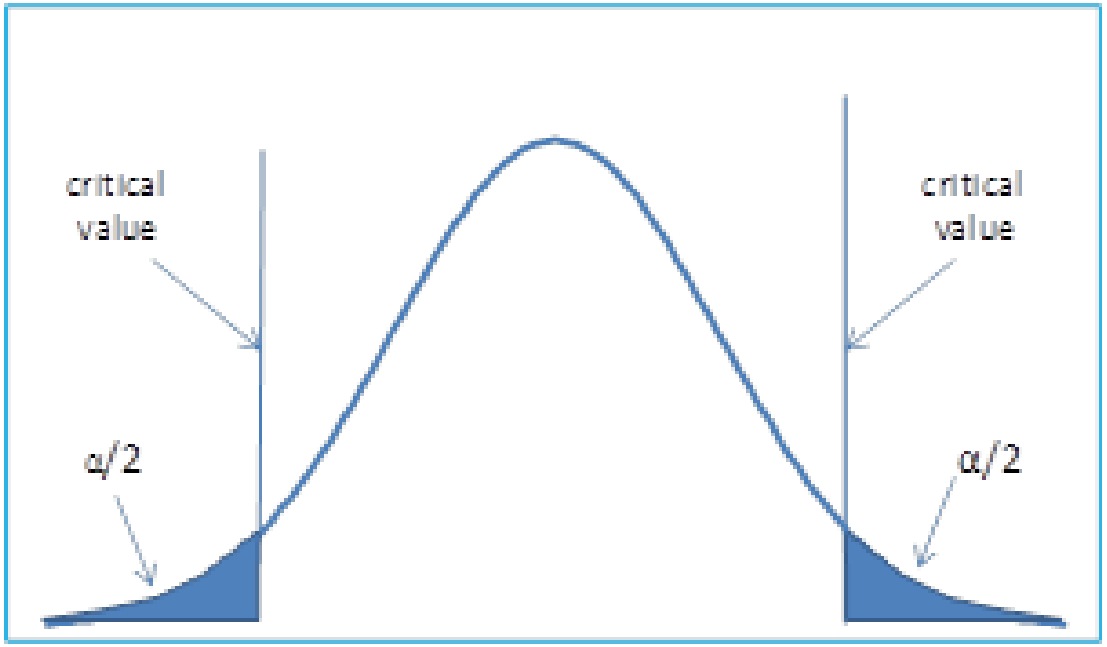

*t* is distributed as Student's t-distribution with $n-2$ degrees of freedom under the null hypothesis

*n* is the number of observations

$$t = r \frac{n-2}{1-r^2}$$

Alternatively we can permute values to observe the empirical distribution of null correlations

# One sided vs. two sided test




## Two sided tests are used when we are testing for a difference without regard to direction

Two sided tests allocate half the area to each direction
Thus they are more strict if you only wish to test in one direction

## Binomial test for probability that null model would produce observed results

$n$ is the number of observations in test set
$k$ is the number classified correctly test set
$p$ is the probability classifier will make correct choice at random
Probability that exactly $k$ observations are classified correctly by null
$Pr(x = k) = \binom{n}{k} p^k (1-p)^{n-k}$
Test that k or greater would have been classified correctly by null
$p = \sum_{i=k}^{n} Pr(x = i)$
This can be approximated by a Chi-squared test

# Multiple hypothesis correction is important

## If we ask $m$ questions we need to adjust our probability that the null is likely

$p_{single}$ Probability one test occurred by chance
$p_{corrected} \leq m * p_{single}$ from Boole's inequality results in the Bonferonni correction
$p_{single} \leq \frac{p_{corrected}}{m}$ Filter for significant events

## Benjamini-Hochberg uses a desired false discover rate to provide a relaxed bound

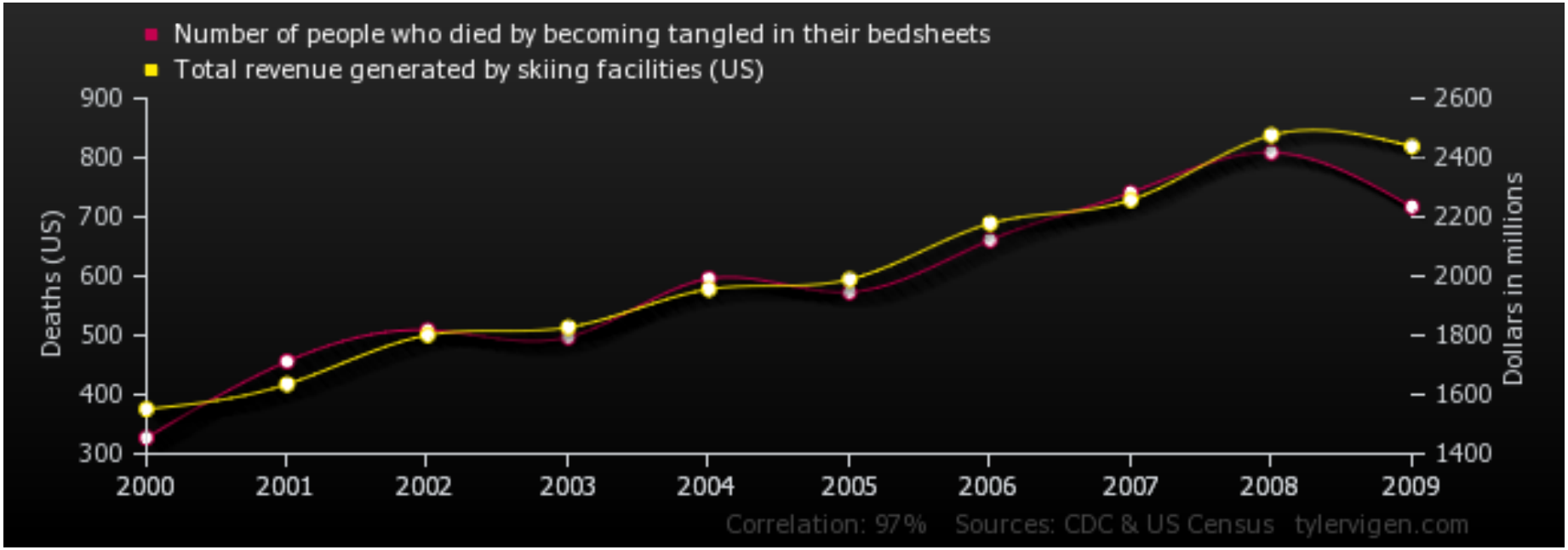
$\alpha$ is our desired false discovery rate (FDR)
$m$ is the number of tests $H_1 \ldots H_m$
$P_1 \ldots P_m$ are their p-values in ascending order

- Find the largest $k$ such that $P_k \leq \frac{k}{m}\alpha$
- Reject the null hypothesis for all $H_i$ for $i = 1, \ldots, k$

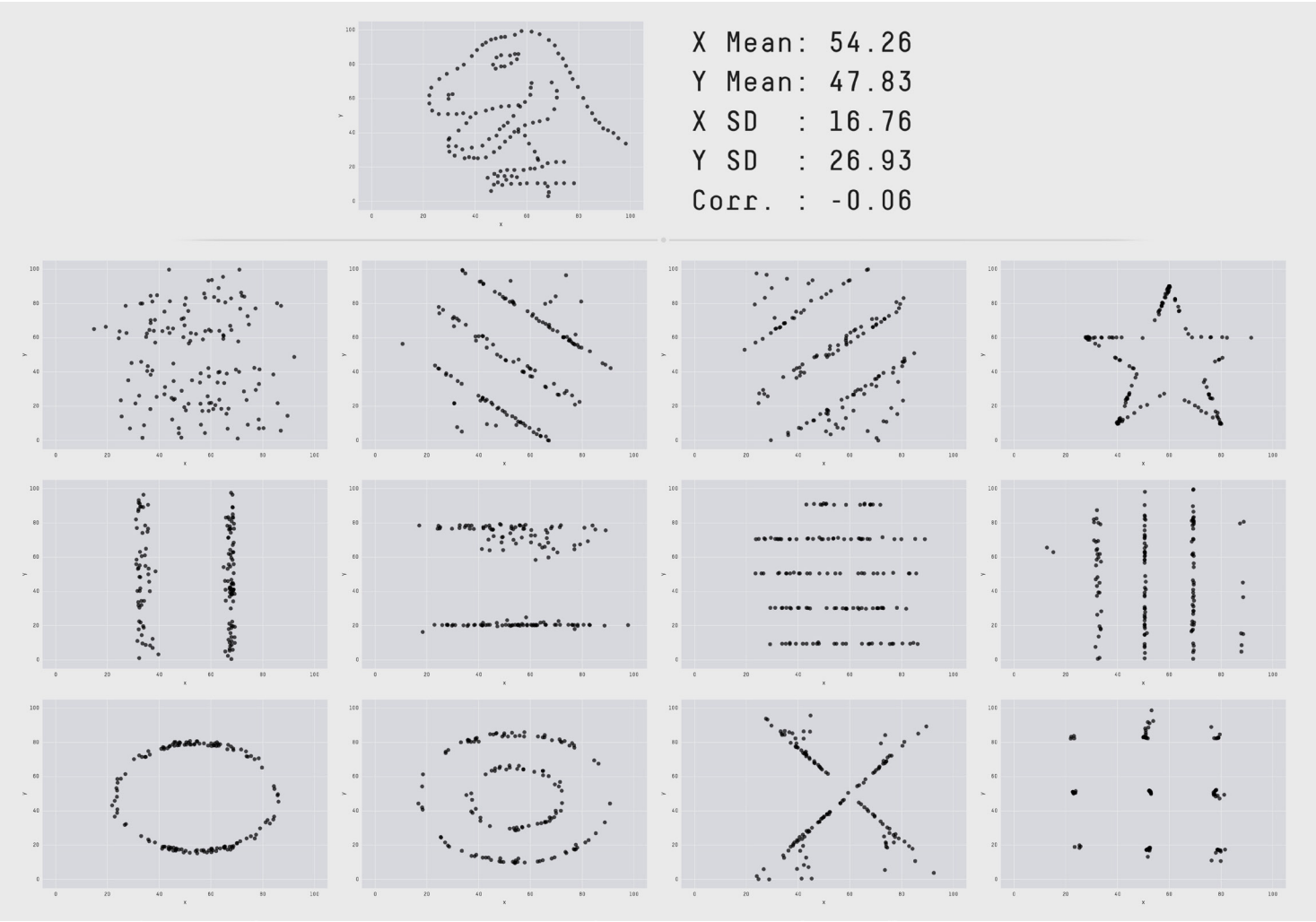
## Which transcription factors $TF_1 \ldots TF_5$ bind with a corrected significance of .05?

Single test p-values are $0.003, 0.006, 0.020, 0.045, 0.600$

# Correlation is not causation

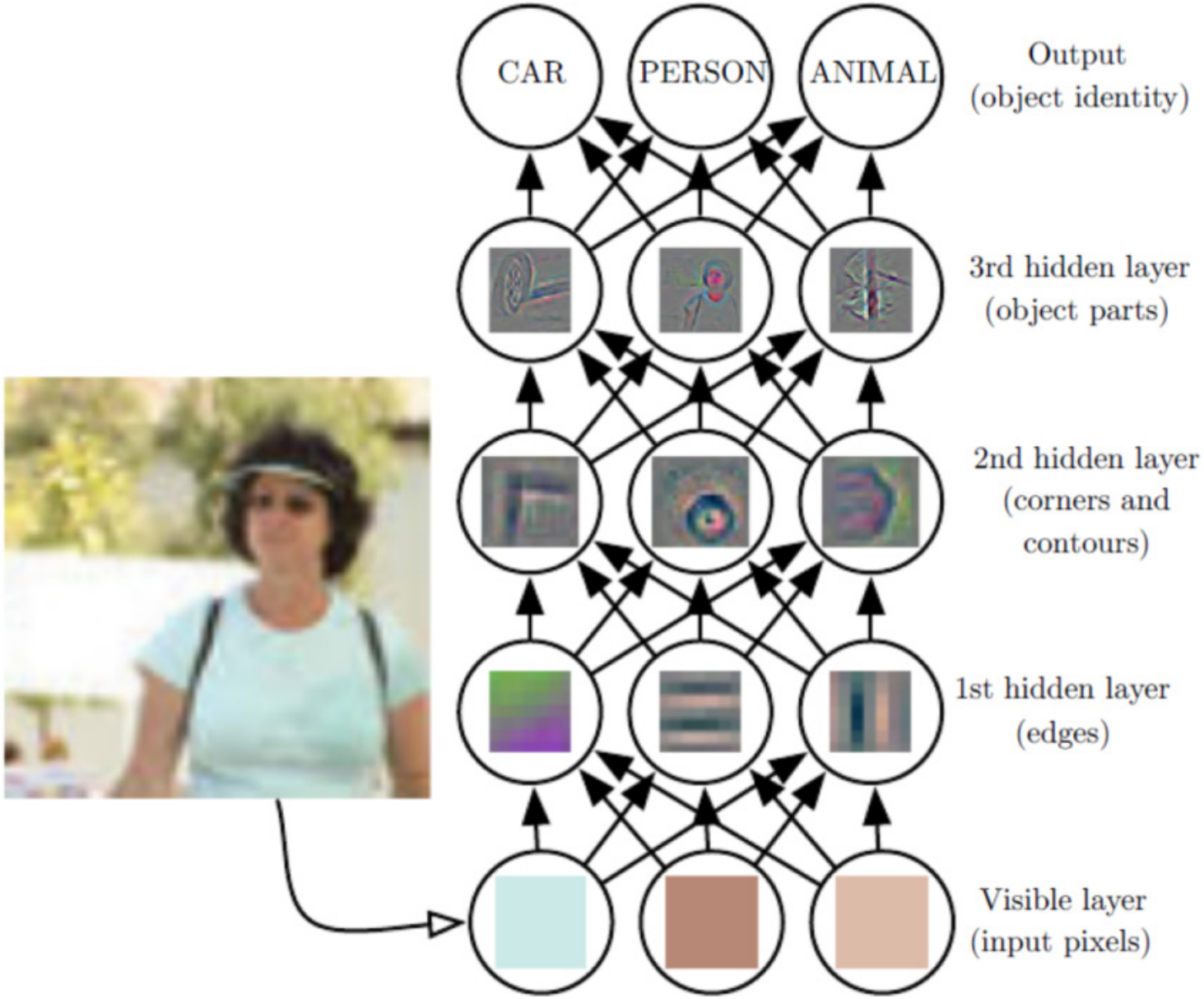


# The Datasaurus Dozen - J. Matejka, G. Fitzmaurice

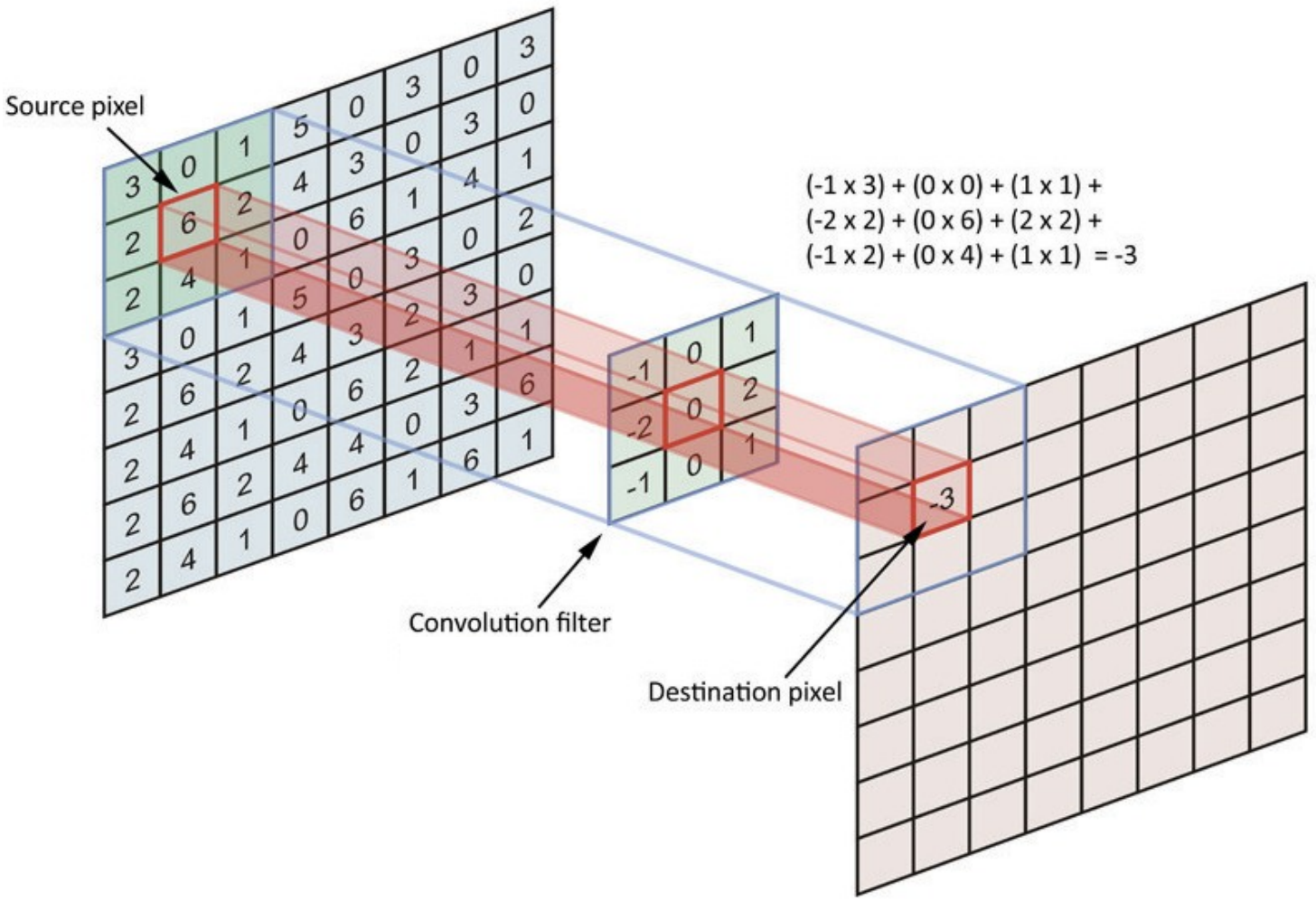


# Traditional Neural Networks

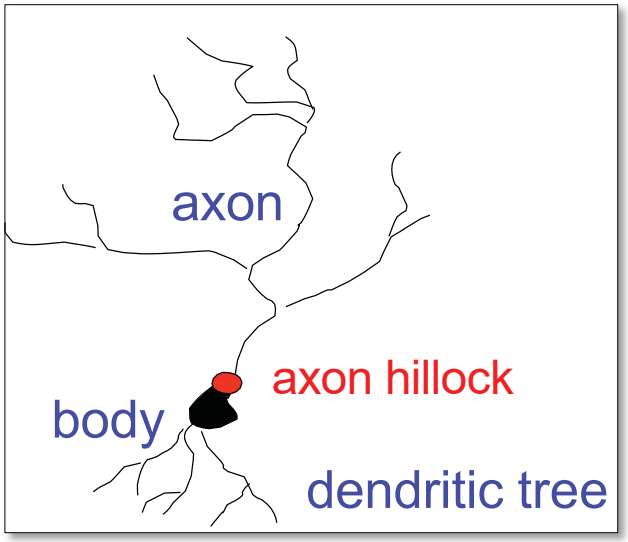
# **Deep** learning ➔ many **layers** of **abstraction**



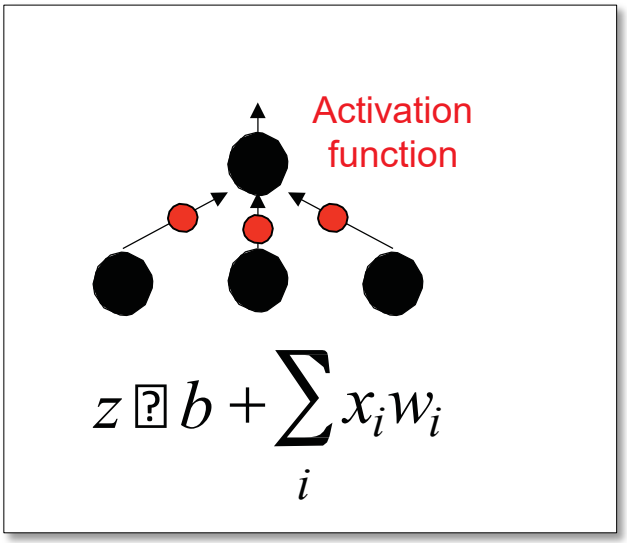
# Convolutional filter



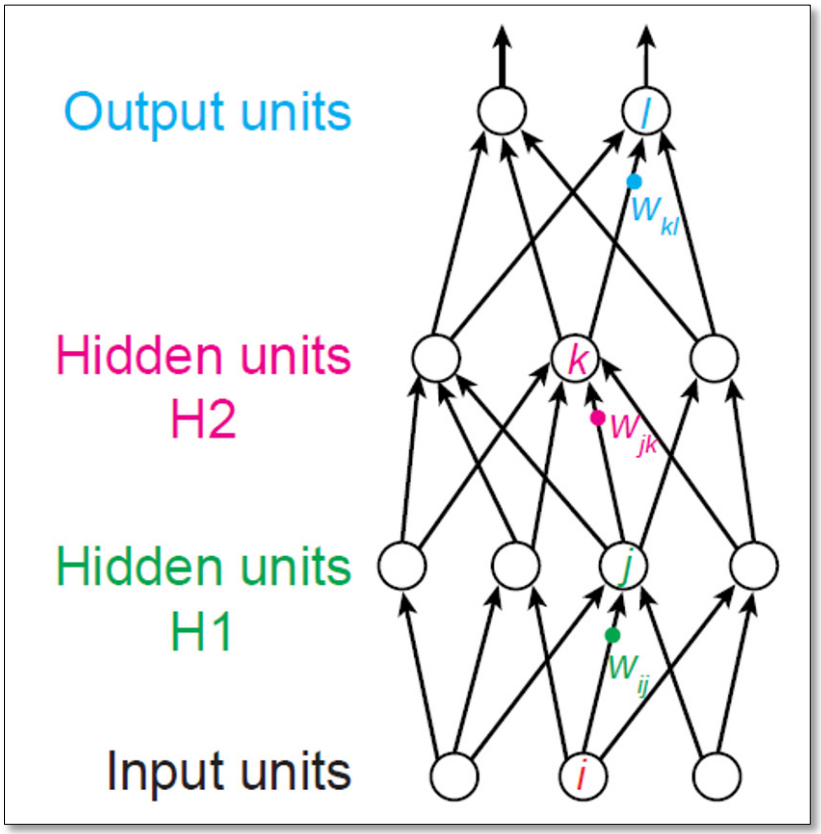
# How the brain works inspired artificial “neural" networks




Biological neuron




Artificial perceptron




Neural Network (e.g. 4-layers ‘deep’)

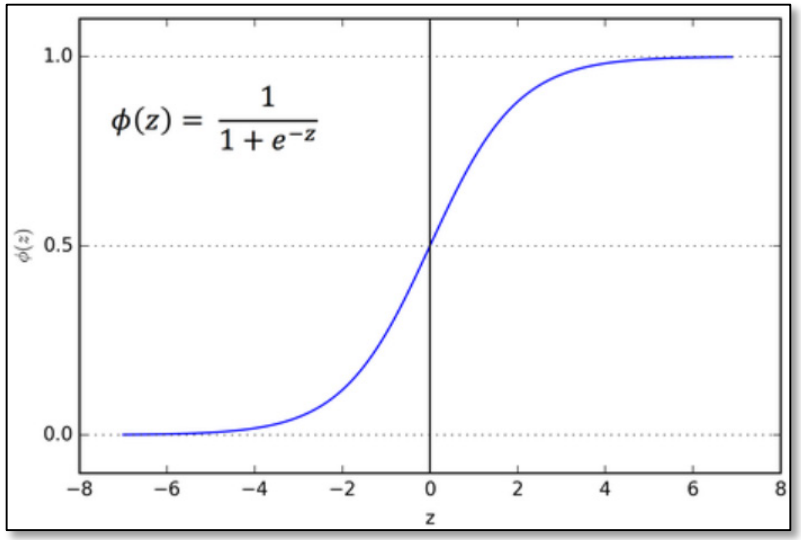
Deep multi-layer neural networks can ‘learn’ almost any function

# Learning non-linear functions: non-linear "activation" units

**Sigmoid unit**

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}.$$

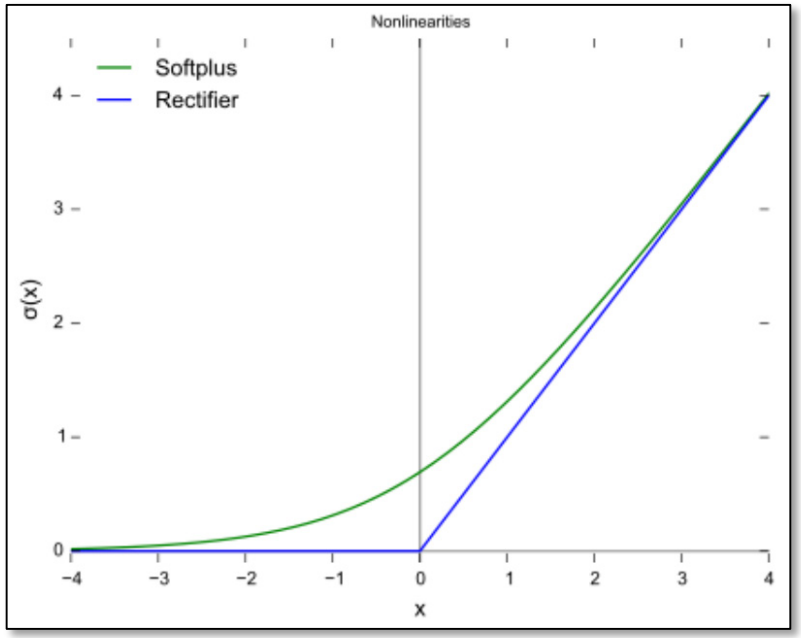
Maps to [0,1], saturation




**Softplus unit**

$$\mathrm{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}.$$

No saturation, smooth transition

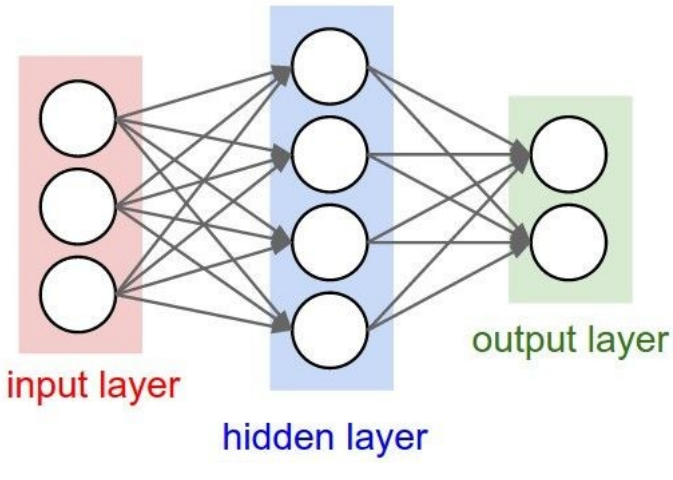


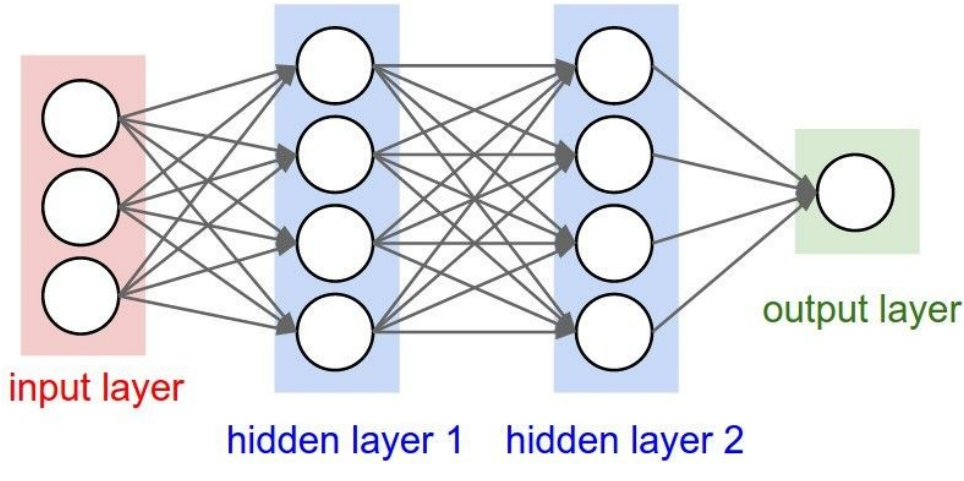

**Rectified linear unit (ReLU)**

$$f(x) = x^+ = \max(0, x)$$

Easy to optimize, generalizes well

# Each hidden layer has an activation function at its output

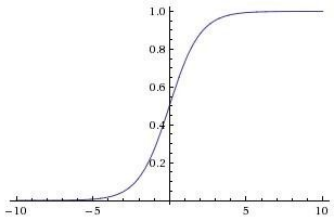


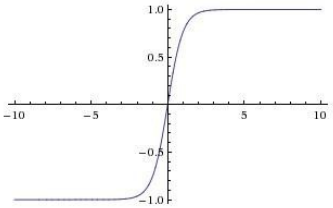
# Popular activation functions at node outputs

**Sigmoid**

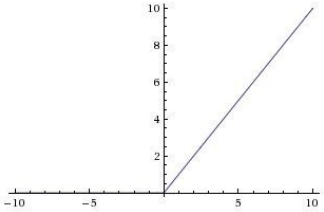
$$\sigma(x) = 1/(1 + e^{-x})$$




**tanh** tanh(x)

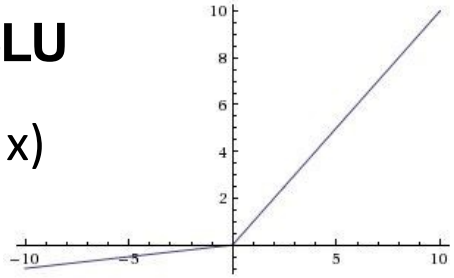



**ReLU** max(0,x)

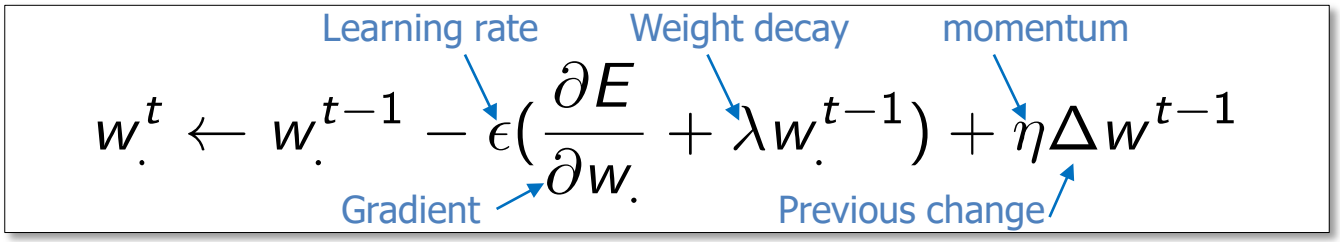



**Leaky ReLU**

max(0.1x, x)



# Gradient-based learning: use derivative to update weights



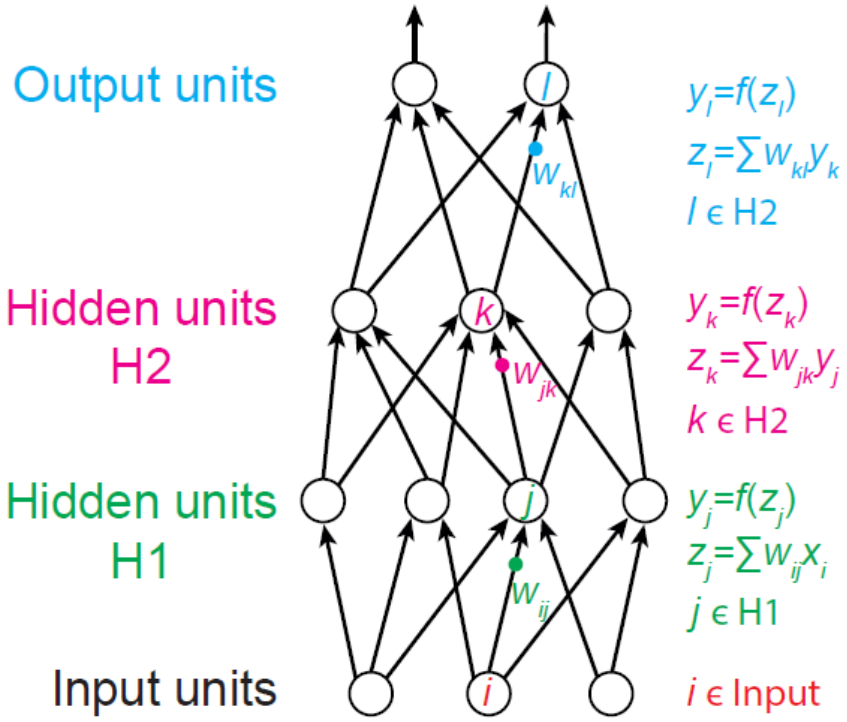

where

- Gradient descent: **∂E/∂w** = partial derivative of error E wrt w
- $\epsilon$ = **learning rate** (e.g. <0.1), needed to not overshoot the optimal solution
- $\lambda$ = **weight decay**, penalizes large weights to prevent overfitting
- $\eta$ = **momentum**, based on magnitude+sign of previous update ($\Delta w^{t-1}$); when direction of update is consistent ➔ faster convergence

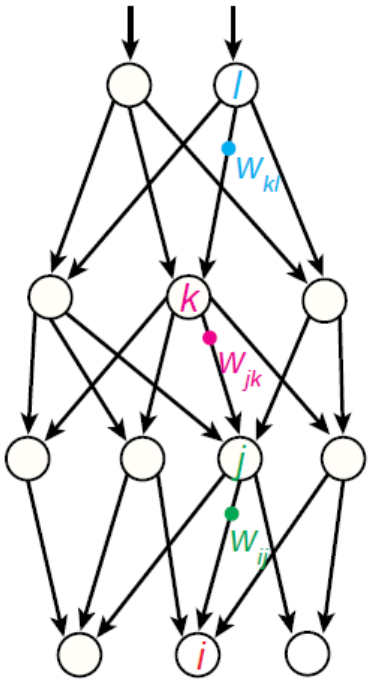
## Using only a subset of samples at a time:

- ***Stochastic*** **gradient descent (SGD)**: speed up computation
  - Randomly sample subset of samples
  - Update the weights using only that subset
- **On-line learning:** Update gradient using only 1 training data point each time

# Back-propagation of error across multiple layers




Forward pass




Backpropagation

$$\frac{\partial E}{\partial y_l} = y_l - t_l$$

$$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial z_l}$$

$$\frac{\partial E}{\partial w_{kl}} = \frac{\partial E}{\partial z_l}\frac{\partial z_l}{\partial w_{kl}}$$

$$\frac{\partial E}{\partial y_k} = \sum_{l \in \text{out}} \frac{\partial z_l}{\partial y_k}\frac{\partial E}{\partial z_l}$$

$$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k}$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial y_j} = \sum_{k \in \text{H2}} \frac{\partial z_k}{\partial y_j}\frac{\partial E}{\partial z_k}$$

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}}$$

$$\text{Update: } w_{\cdot}^{t} \leftarrow w_{\cdot}^{t-1} - \epsilon\left(\frac{\partial E}{\partial w_{\cdot}} + \lambda w_{\cdot}^{t-1}\right) + \eta \Delta w^{t-1}$$

[Rumelhart and Hintont, 1986, LeCun et al., 2015]

# The steepest gradient is not necessarily toward the optimum point

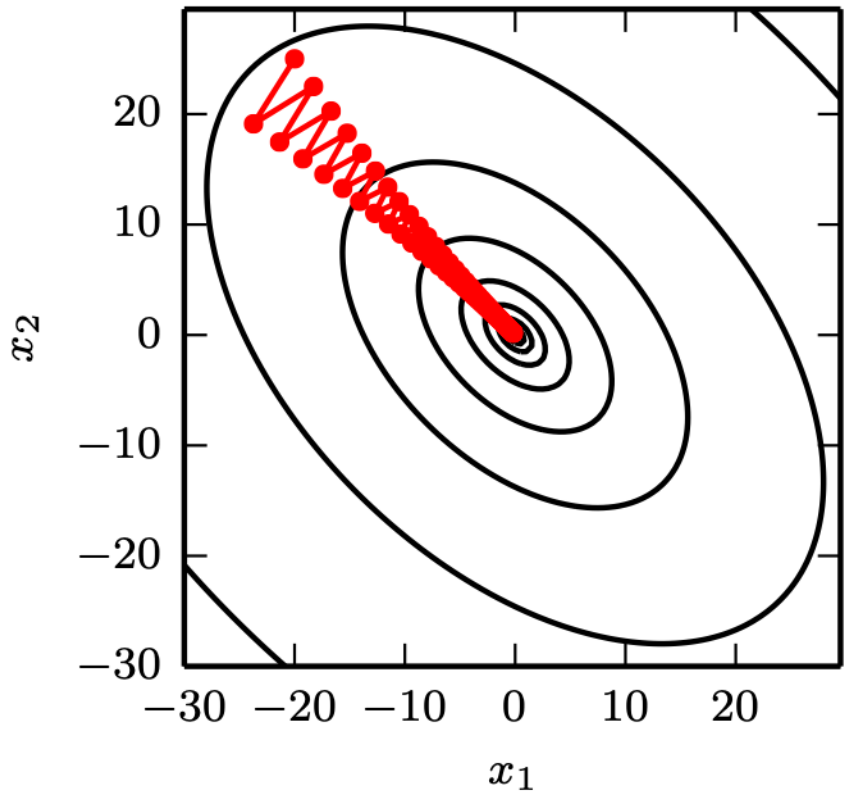
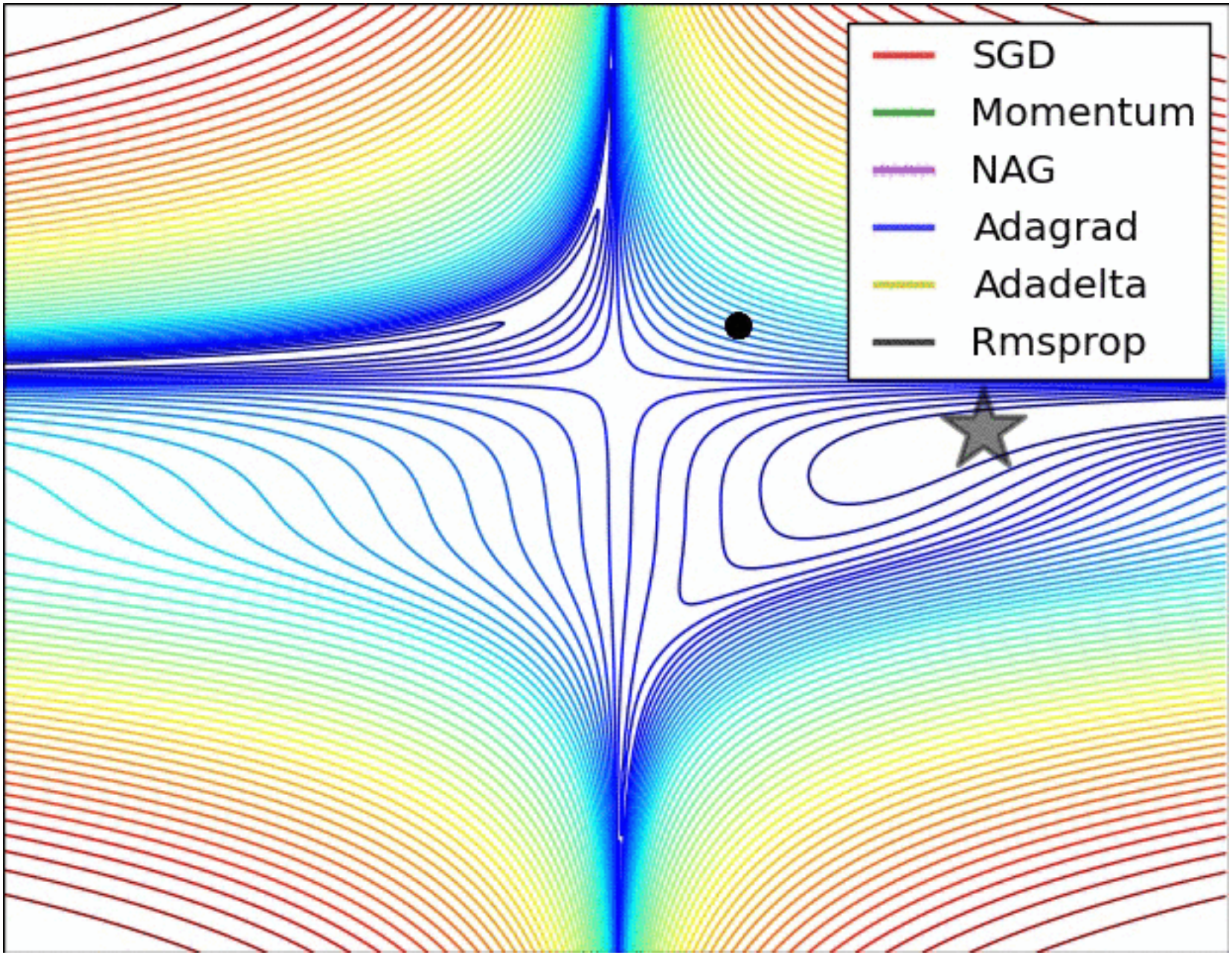
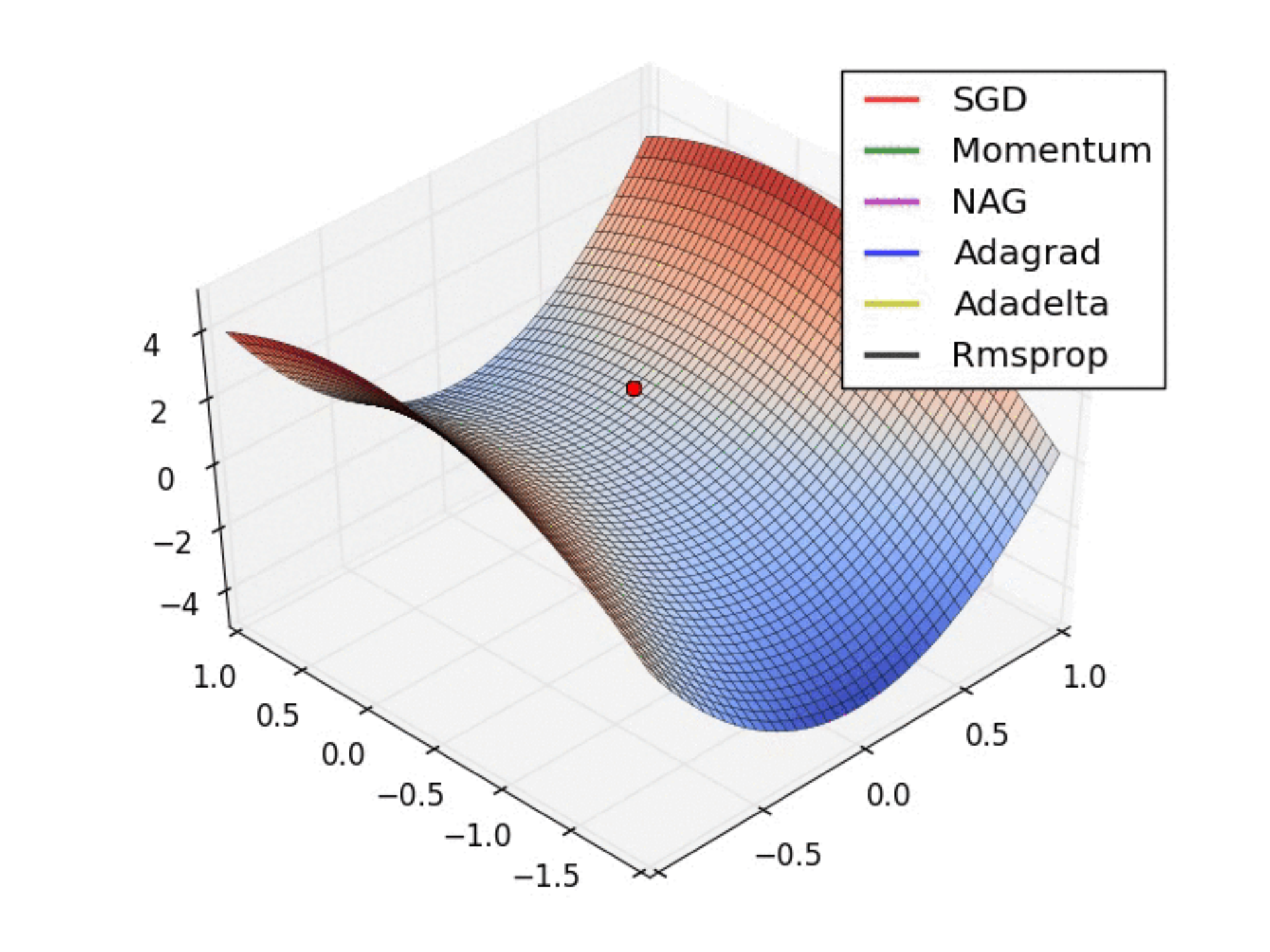



Figure 4.6

# Example gradient management methods

- Stochastic gradient descent (update on each training value)
- Mini-batch gradient descent (average gradient over a mini-batch)
- Momentum methods
  - Adam - exponentially decaying average of past gradients and parameter specific learning rates
  - Adadelta - parameters specific learning rates with fixed memory window


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5

# Gradient learning

Gradient based optimization needs a loss function to minimize

# Stochastic gradient descent updates parameters to reduce loss

$$L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta}).$$

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathbf{y} \sim \hat{p}_{\text{data}}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

$$\boldsymbol{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}).$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}$$

Learning rate

# $y' = w^T x$ find w to best approximate y

## What loss function should we use to pick w?

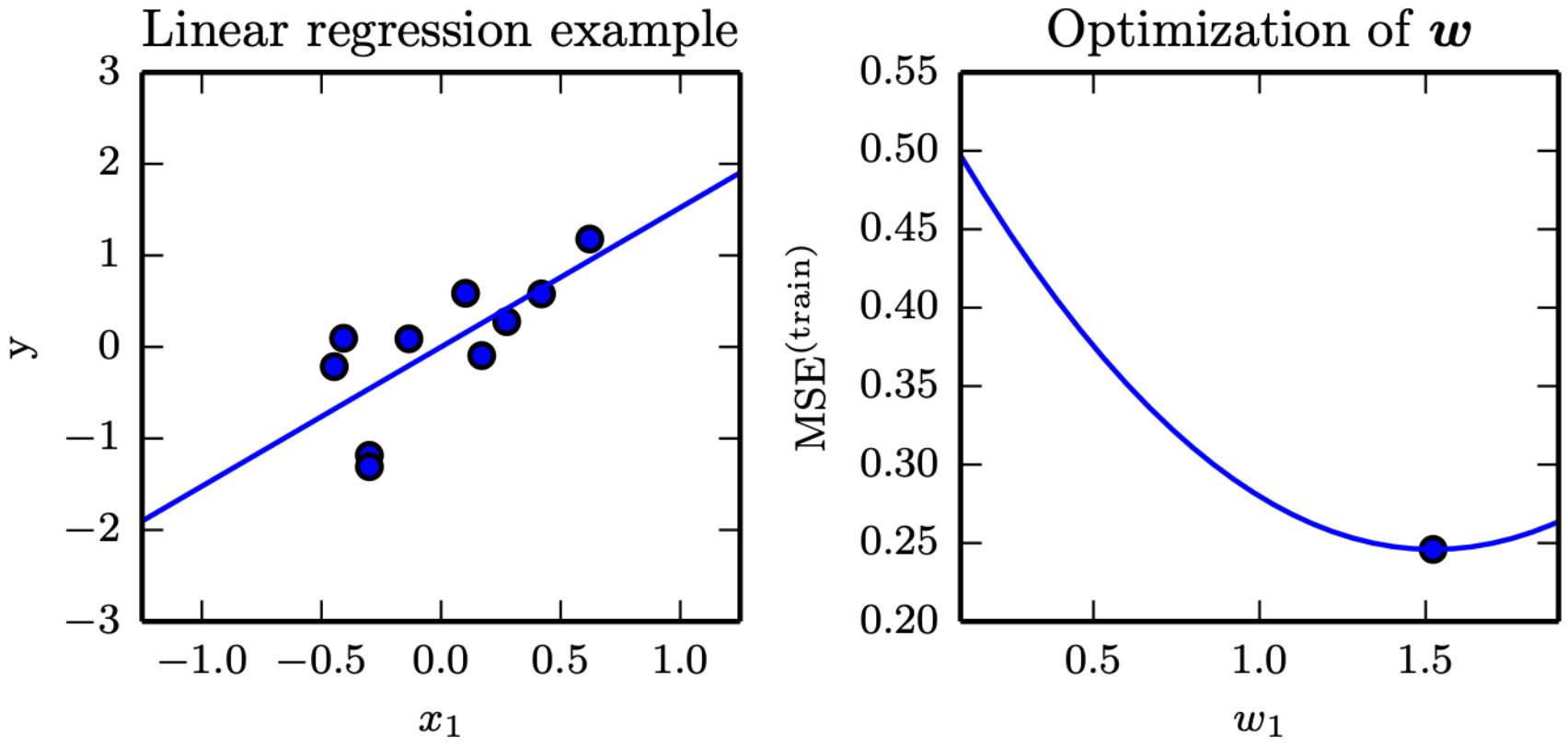



Figure 5.1

# Minimizing mean-square error maximizes Gaussian likelihood

$$f(x|\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$\log f(x|\theta) = C_1 - \frac{(x-\mu)^2}{C_2}$$

$$\arg\max_{\theta} f(x|\theta) = \arg\min_{\theta} (x-\mu)^2$$

# Gradient Descent for linear regression

$$\vec{w}, b = \arg\min_{\vec{w},b} \sum_{i=1}^{n} (y^{(i)} - \vec{w}^T \vec{x}^{(i)} - b)^2$$

$$\frac{\partial E}{\partial \vec{w}} = \frac{2}{n} \sum_{i=1}^{n} (\vec{w}^T \vec{x}^{(i)} + b - y^{(i)}) \vec{x}^{(i)}$$

$$\frac{\partial E}{\partial \vec{w}} = \frac{2}{n} \sum_{i=1}^{n} (\widehat{y}^{(i)} - y^{(i)}) \vec{x}^{(i)}$$

$$\frac{\partial E}{\partial b} = \frac{2}{n} \sum_{i=1}^{n} (\widehat{y}^{(i)} - y^{(i)})$$

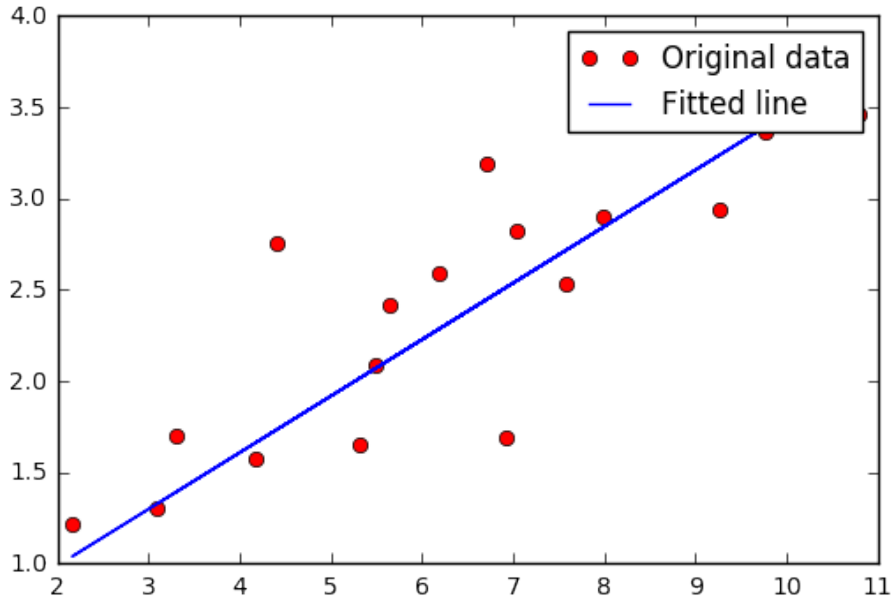
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}$$

# Linear regression can be solved by gradient descent

```
# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")

# Set model weights
W = tf.Variable(rng.randn(), name="weight")
b = tf.Variable(rng.randn(), name="bias")

# Construct a linear model
pred = tf.add(tf.mul(X, W), b)

# Mean squared error
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)

# Gradient descent
optimizer =
tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

# Cost is minimized after 1000 epochs

```
Epoch: 0050 cost= 0.195095107 W= 0.441748 b= -0.580876
Epoch: 0100 cost= 0.181448311 W= 0.430319 b= -0.498661
Epoch: 0150 cost= 0.169377610 W= 0.419571 b= -0.421336
Epoch: 0200 cost= 0.158700854 W= 0.409461 b= -0.348611
Epoch: 0250 cost= 0.149257123 W= 0.399953 b= -0.28021
Epoch: 0300 cost= 0.140904188 W= 0.391011 b= -0.215878
Epoch: 0350 cost= 0.133515999 W= 0.3826 b= -0.155372
Epoch: 0400 cost= 0.126981199 W= 0.374689 b= -
0.0984639
Epoch: 0450 cost= 0.121201262 W= 0.367249 b= -
0.0449408
Epoch: 0500 cost= 0.116088994 W= 0.360252 b=
0.00539905
Epoch: 0550 cost= 0.111567356 W= 0.35367 b= 0.052745
Epoch: 0600 cost= 0.107568085 W= 0.34748 b= 0.0972751
Epoch: 0650 cost= 0.104030922 W= 0.341659 b= 0.139157
Epoch: 0700 cost= 0.100902475 W= 0.336183 b= 0.178547
Epoch: 0750 cost= 0.098135538 W= 0.331033 b= 0.215595
Epoch: 0800 cost= 0.095688373 W= 0.32619 b= 0.25044
Epoch: 0850 cost= 0.093524046 W= 0.321634 b= 0.283212
Epoch: 0900 cost= 0.091609895 W= 0.317349 b= 0.314035
Epoch: 0950 cost= 0.089917004 W= 0.31332 b= 0.343025
Epoch: 1000 cost= 0.088419855 W= 0.30953 b= 0.370291
Optimization Finished!
Training cost= 0.0884199 W= 0.30953 b= 0.370291
```



# Example convex functions

- $c(x) = Mx + b$
- $c(x) = e^{c1(x)}$
- $c(x) = c1(x) + c2(x)$
- $c(x) = x^p$
- $c(x) = |x|$
- $c(x) = x \log x$
- $c(x) = \max( c1(x),\ c2(x) )$

Deep neural networks are typically non-convex functions

# Log convex functions

- They are not convex but their log is convex
- Example - sigmoid
- They can be optimized by convex solvers

We may not always find the best solution for non-convex functions

# Approximate Optimization

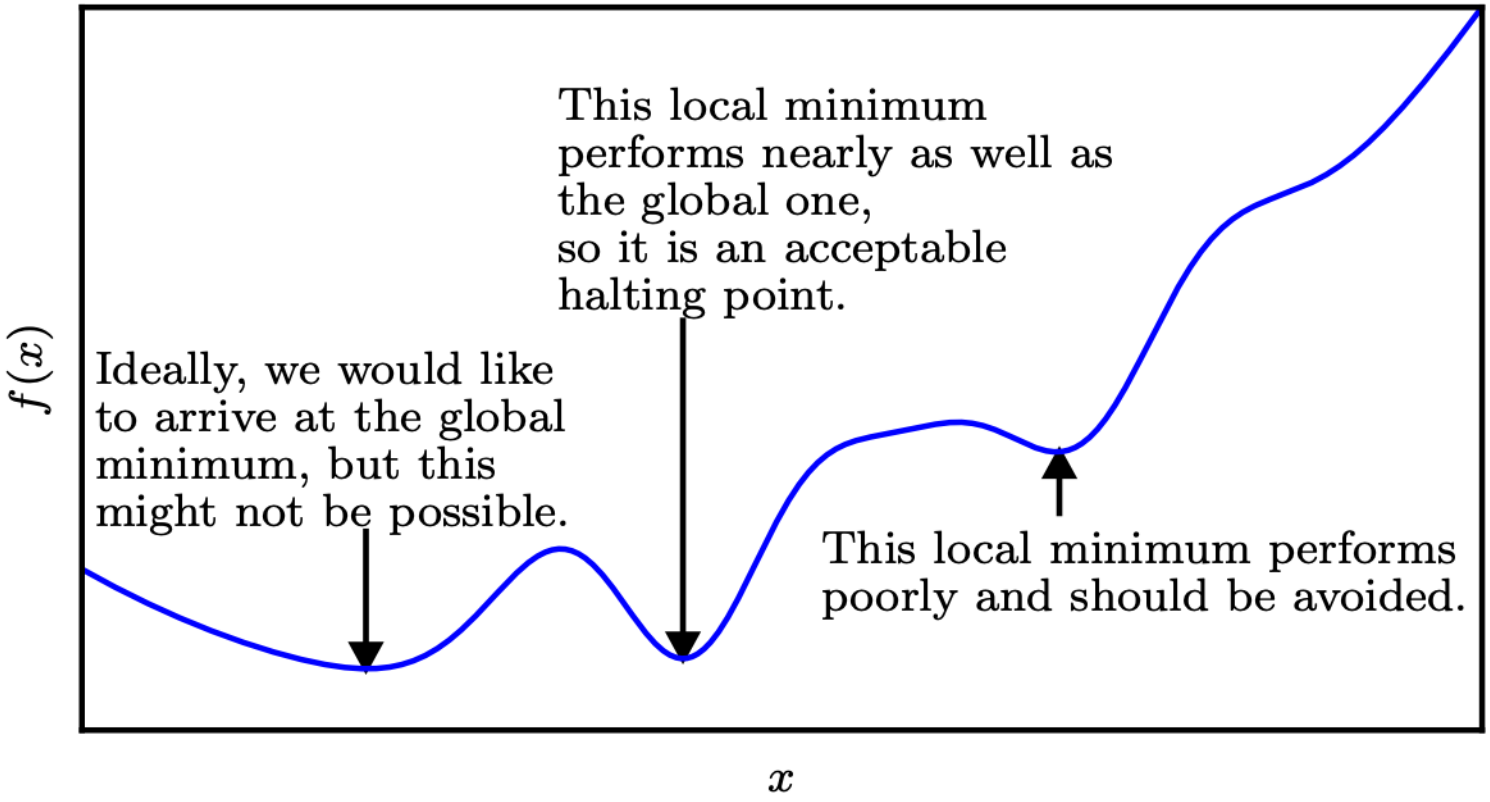
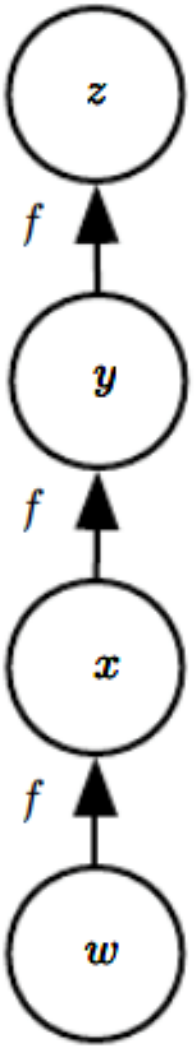



Figure 4.3

# Backpropagation

# Backpropagation computes gradients via the chain rule




$$\frac{\partial z}{\partial w}$$

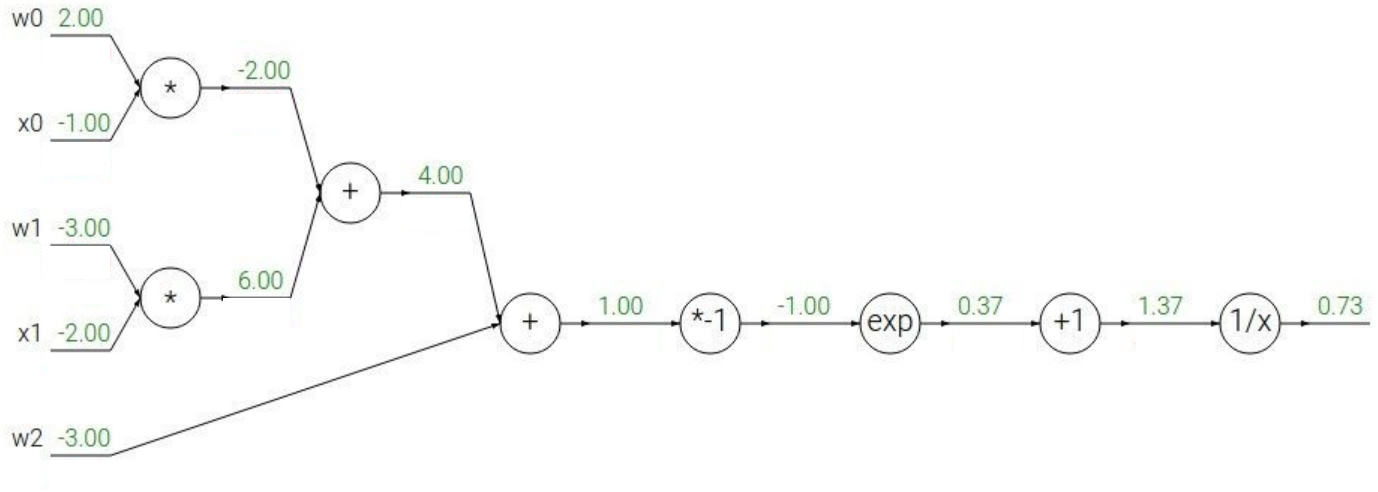
$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$

$$= f'(y)f'(x)f'(w)$$

$$= f'(f(f(w)))f'(f(w))f'(w)$$

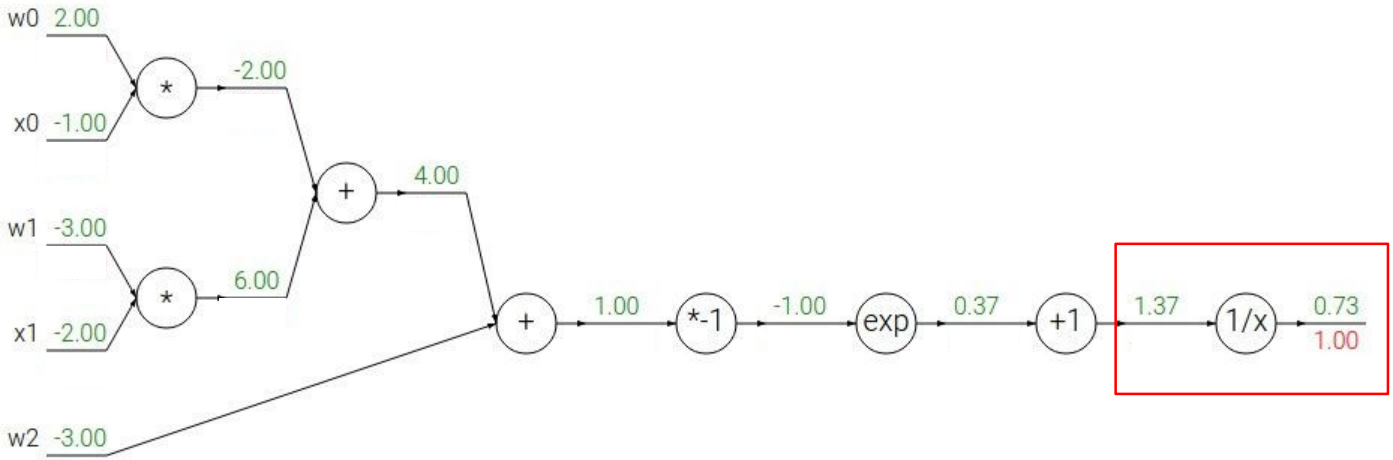
# Example of backpropagation of errors

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$




$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

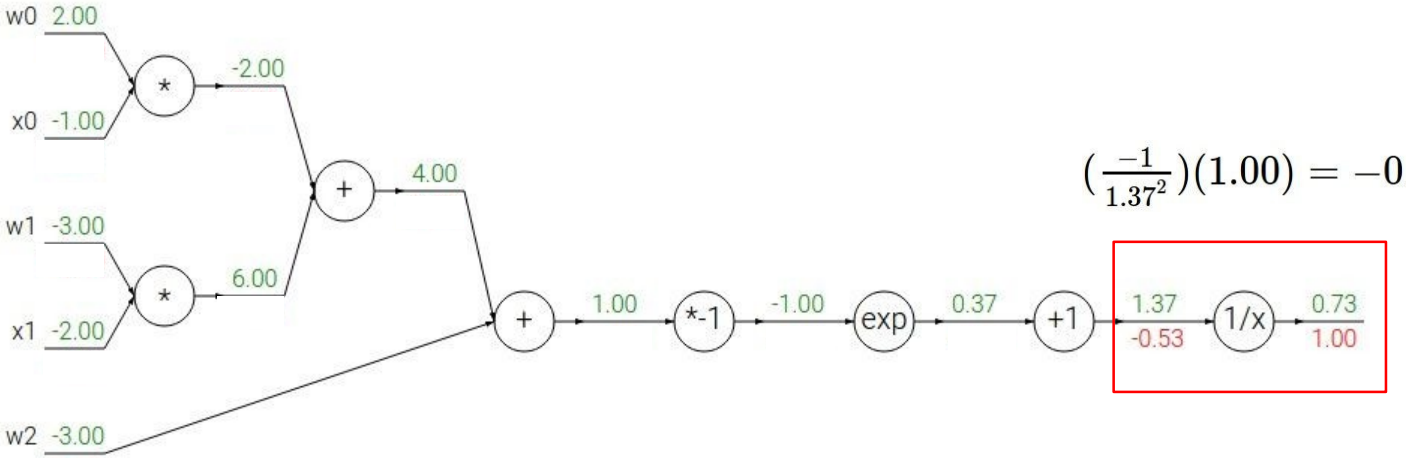
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$




$$(\frac{-1}{1.37^2})(1.00) = -0.53$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

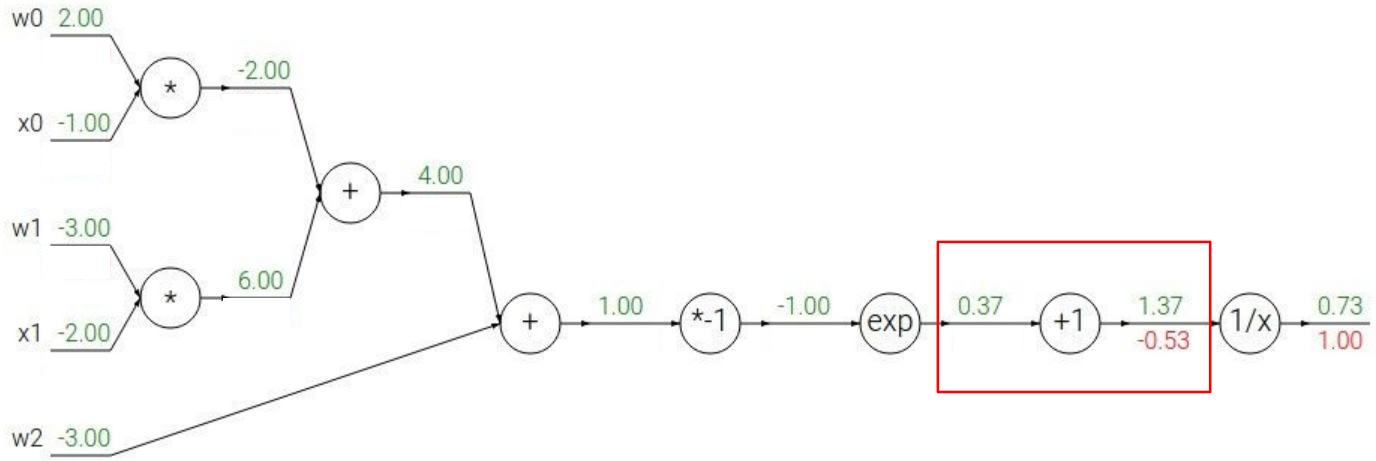
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$




$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

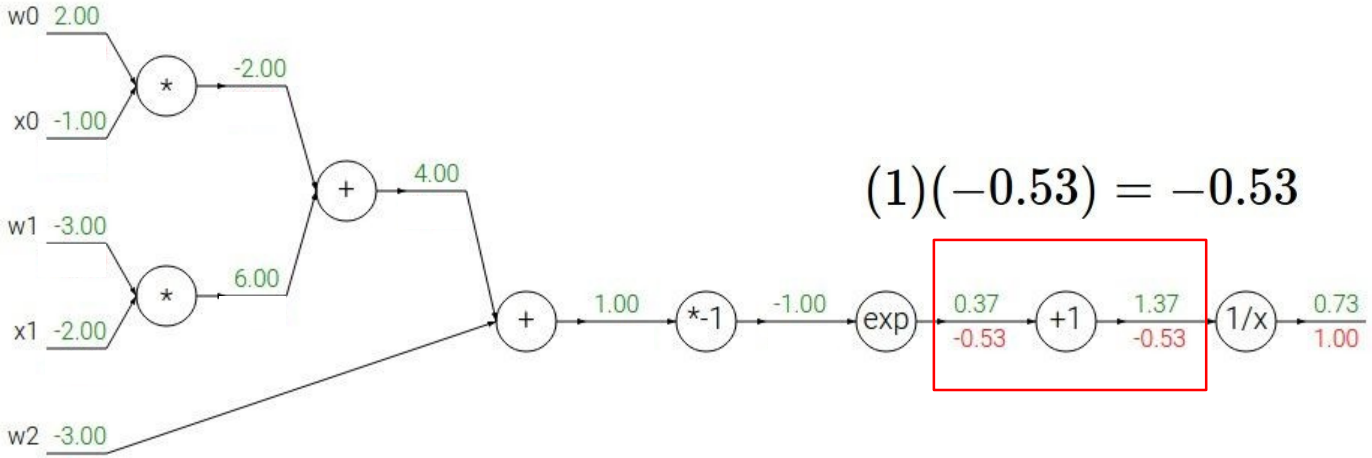
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$




$$(1)(-0.53) = -0.53$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

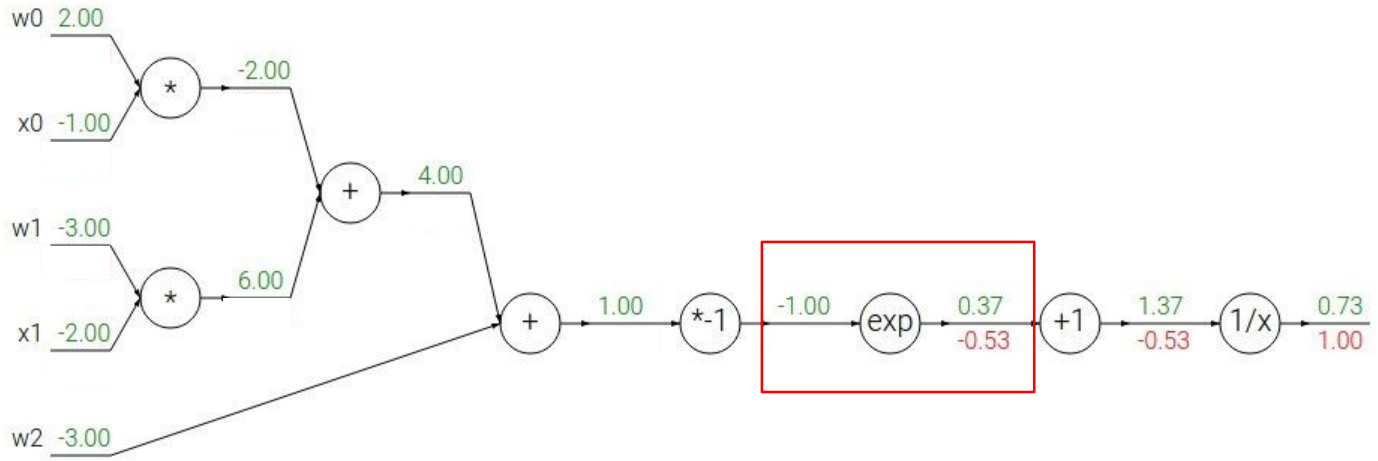
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$




$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

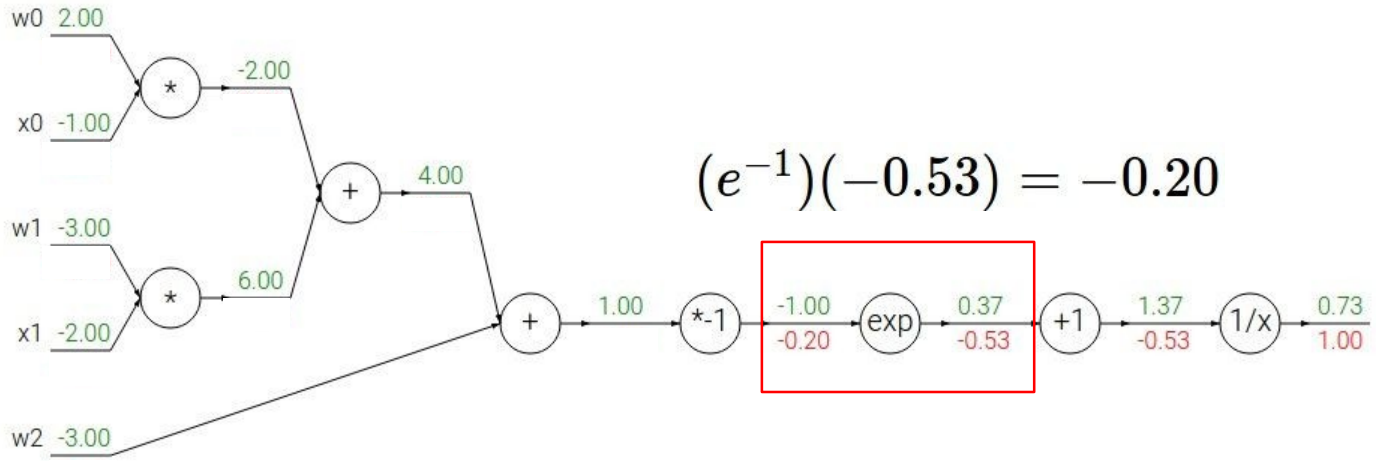
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

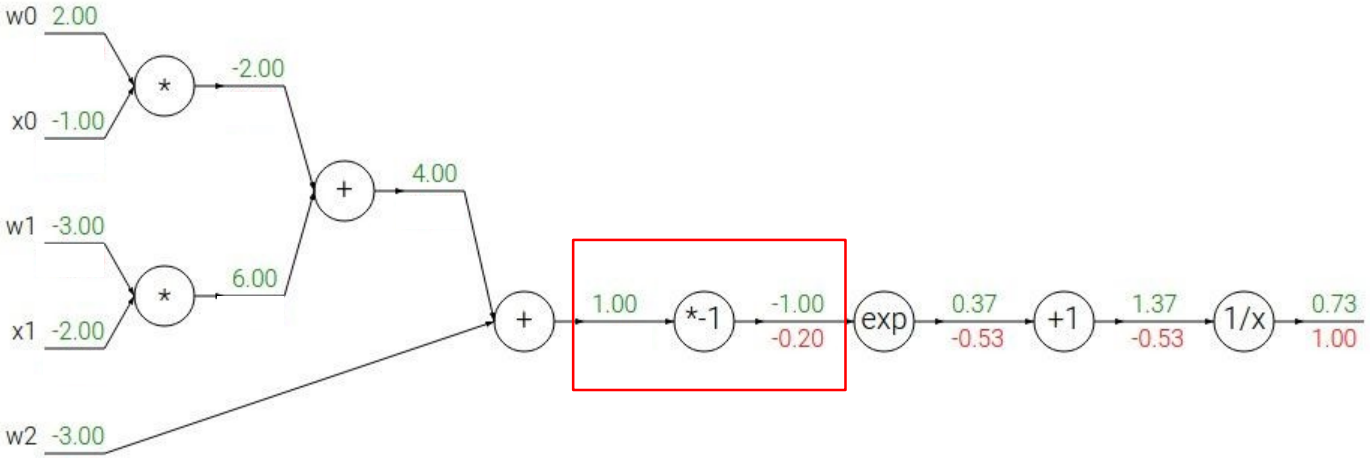



$$(e^{-1})(-0.53) = -0.20$$

| | | | | | |
|---|---|---|---|---|---|
| $f(x) = e^x$ | $\rightarrow$ | $\frac{df}{dx} = e^x$ | $f(x) = \frac{1}{x}$ | $\rightarrow$ | $\frac{df}{dx} = -1/x^2$ |
| $f_a(x) = ax$ | $\rightarrow$ | $\frac{df}{dx} = a$ | $f_c(x) = c + x$ | $\rightarrow$ | $\frac{df}{dx} = 1$ |

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$




$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

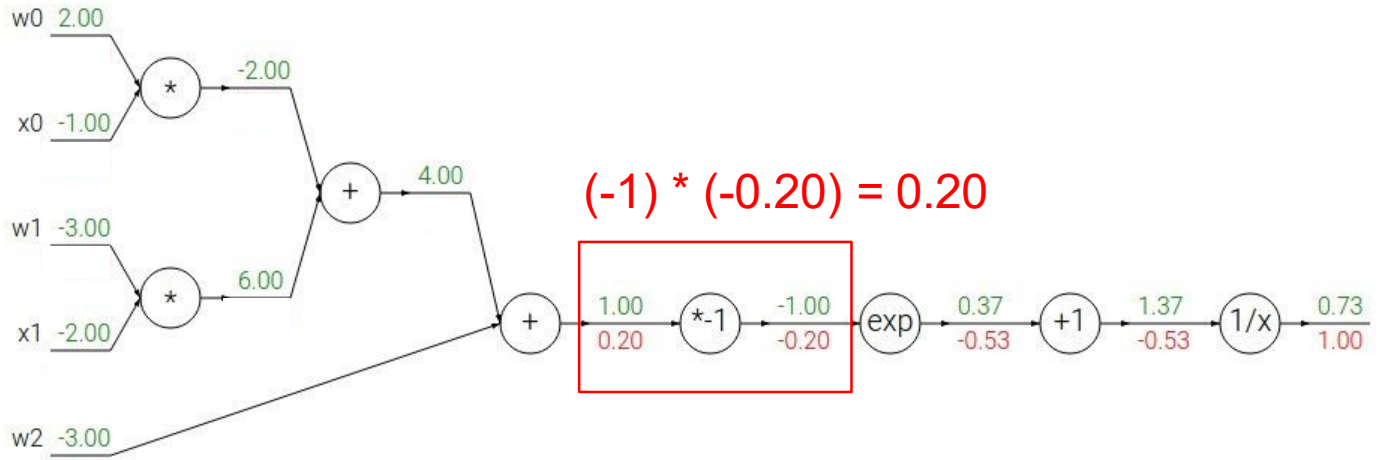
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$




(-1) * (-0.20) = 0.20

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

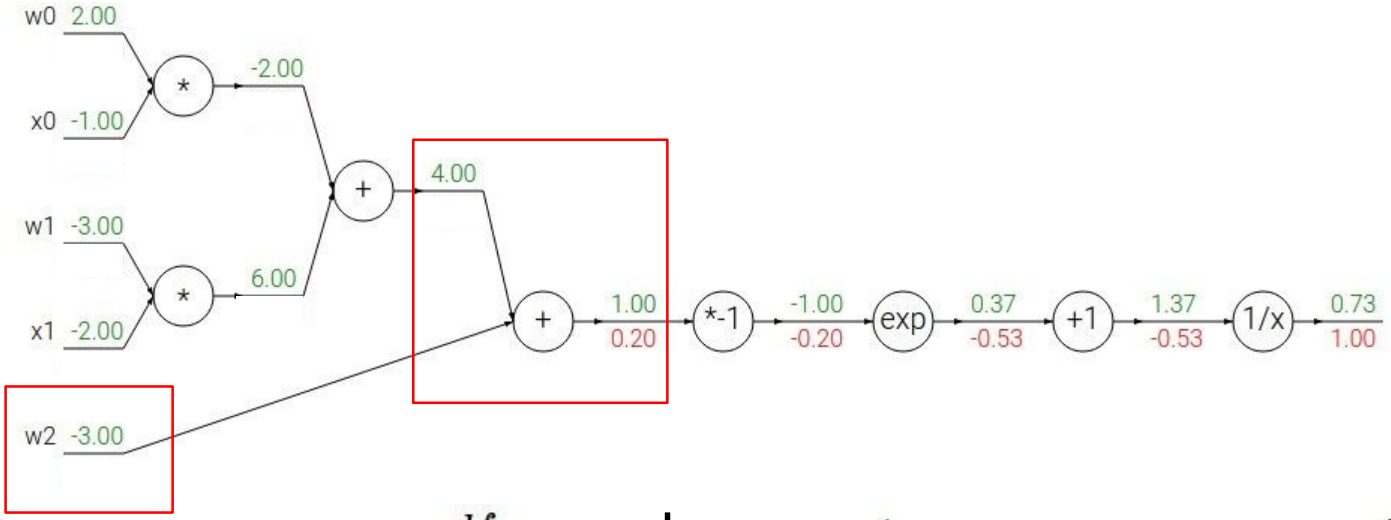
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

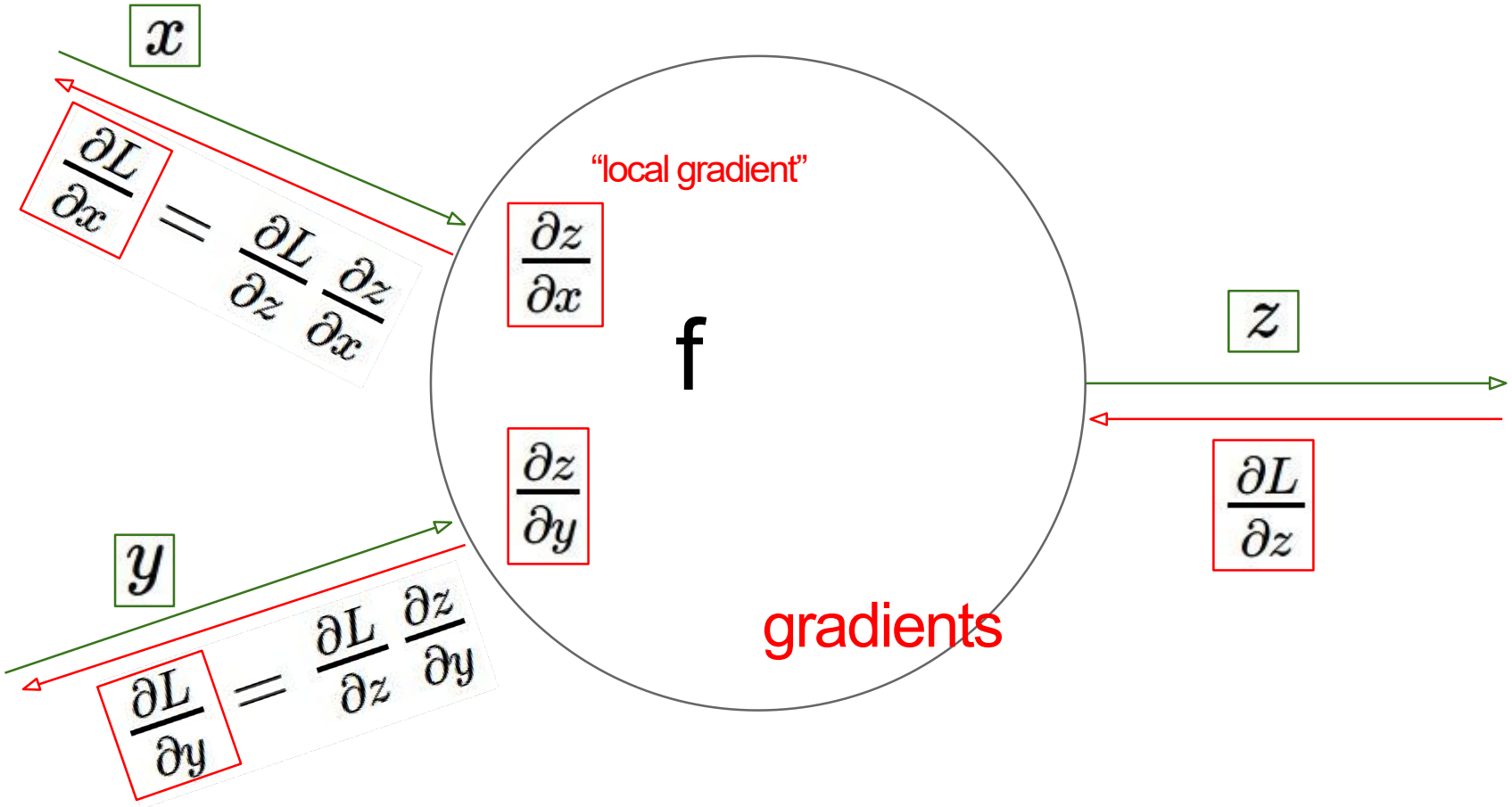



$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

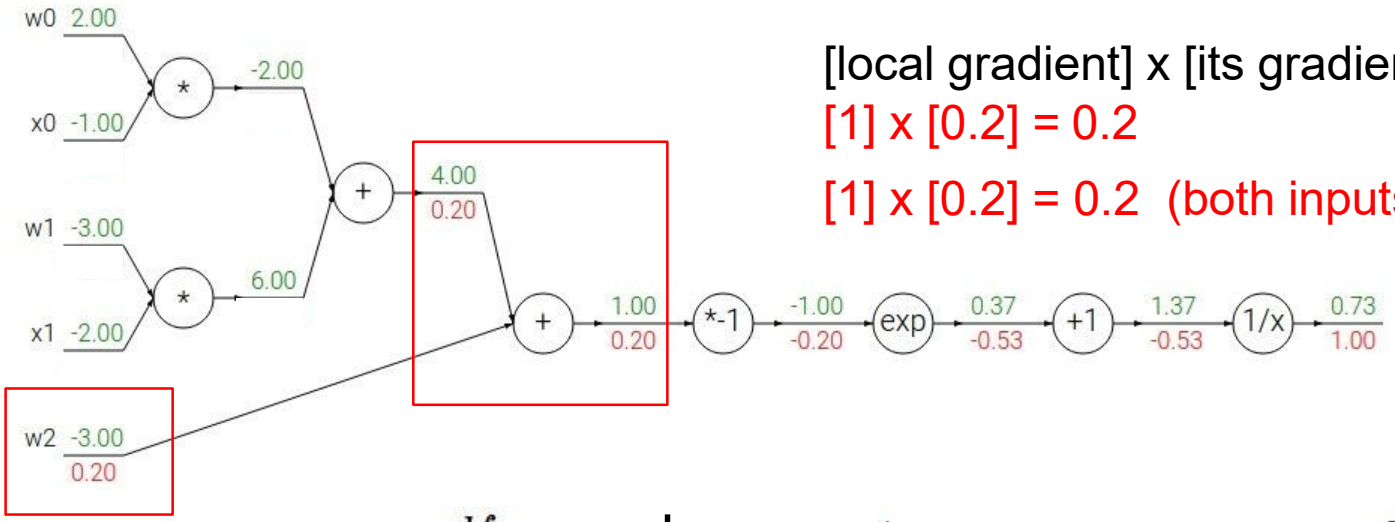
$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$


$x$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
$z$
$\frac{\partial L}{\partial z}$
$y$
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
gradients

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$




[local gradient] x [its gradient]

[1] x [0.2] = 0.2

[1] x [0.2] = 0.2 (both inputs!)

$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$

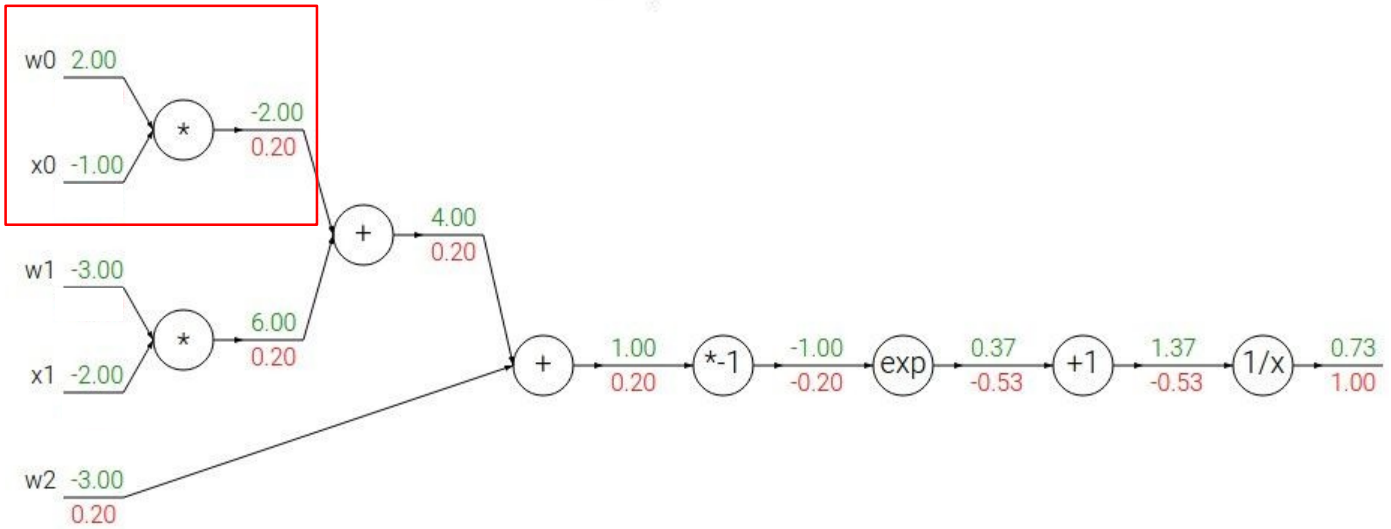
$f_a(x) = ax \rightarrow \frac{df}{dx} = a$

$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$




$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

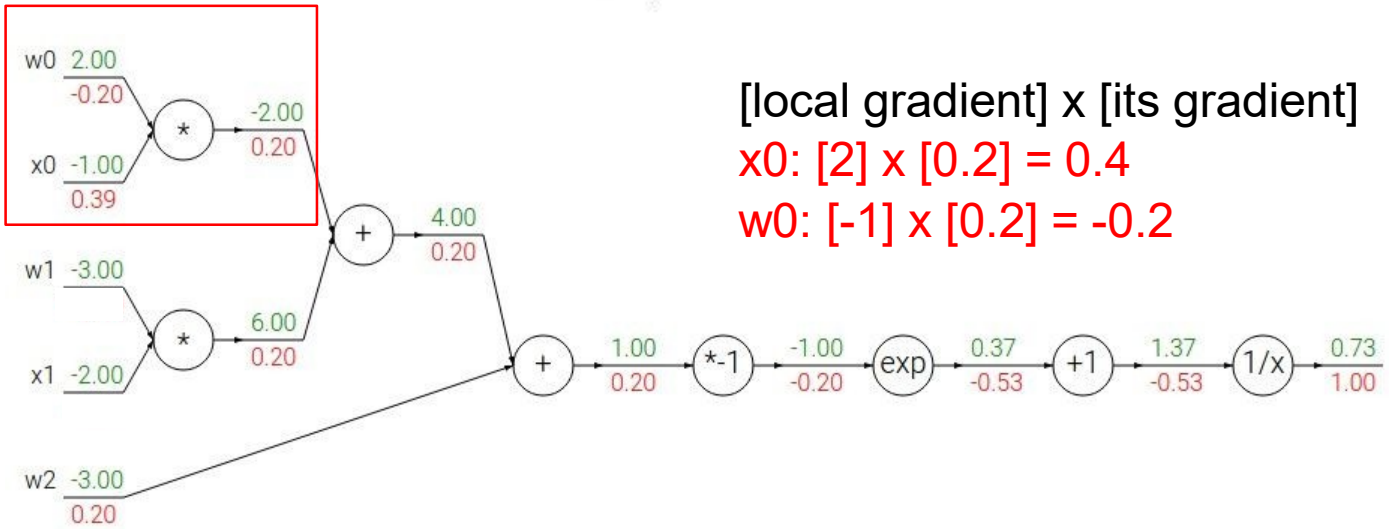
$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

Another example:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$




[local gradient] x [its gradient]

x0: [2] x [0.2] = 0.4

w0: [-1] x [0.2] = -0.2

$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

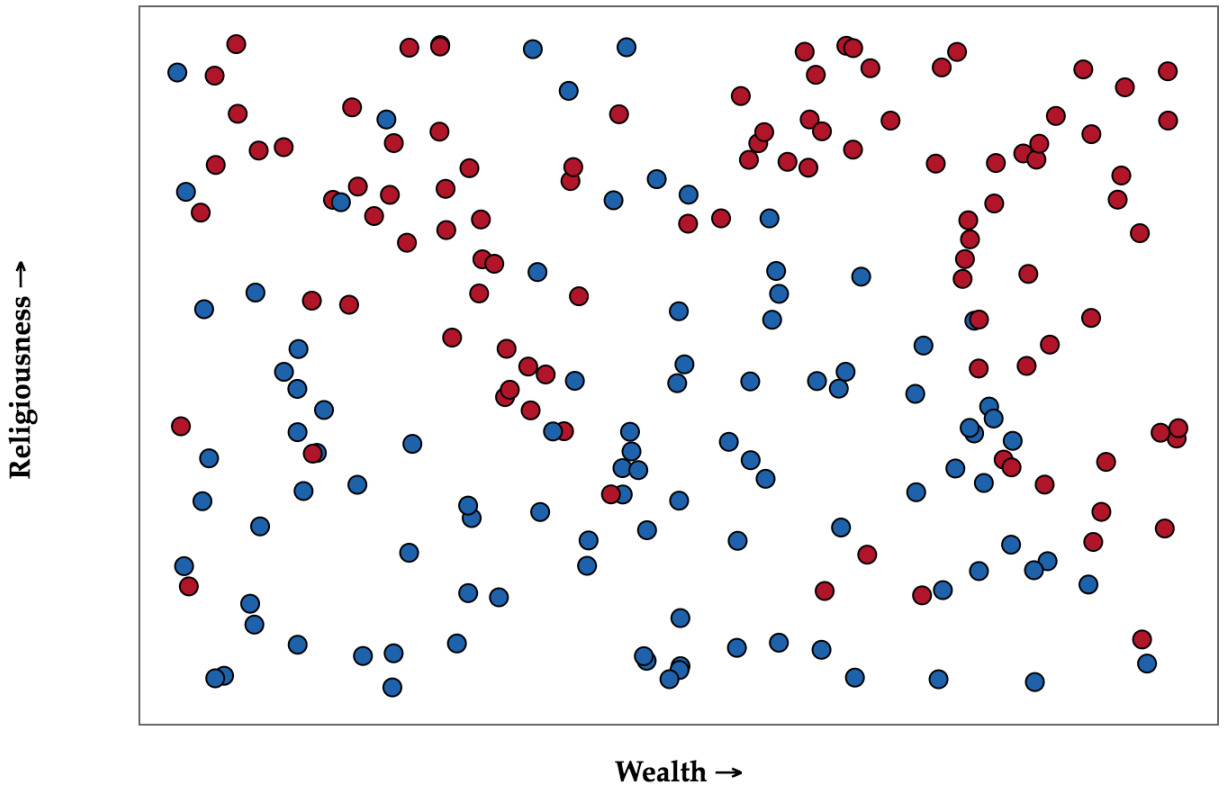
$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

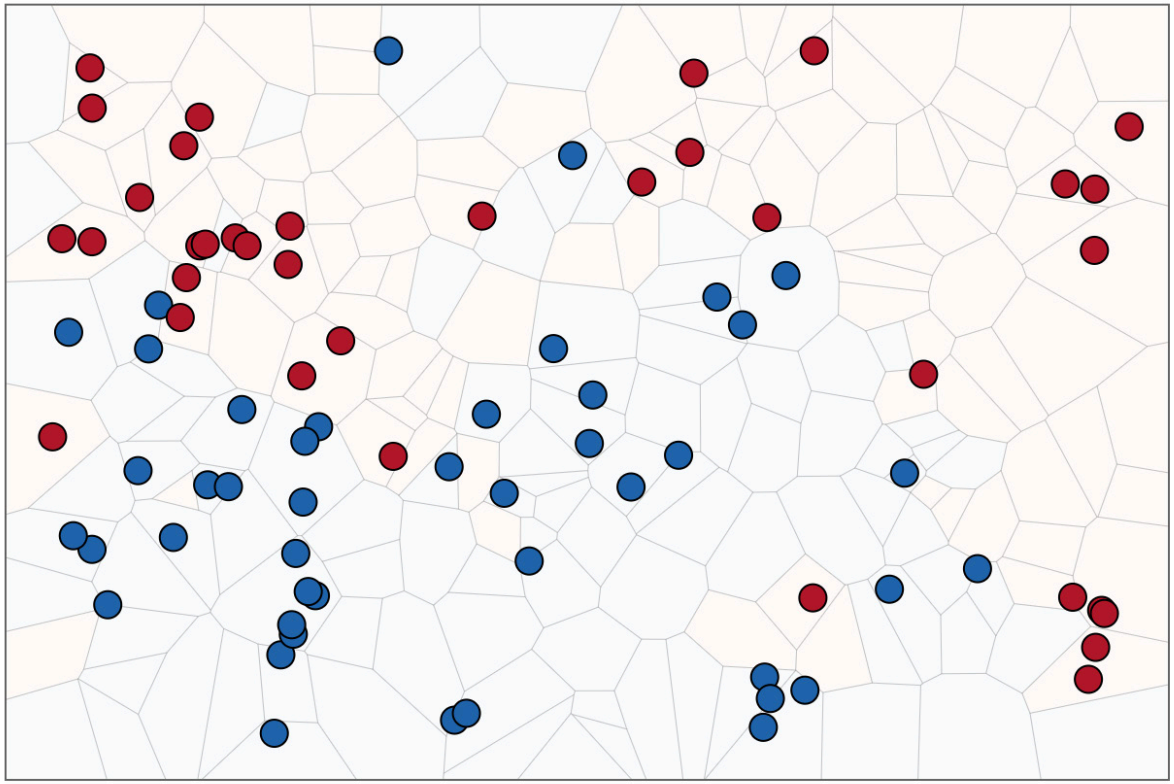
$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

# Controlling model complexity

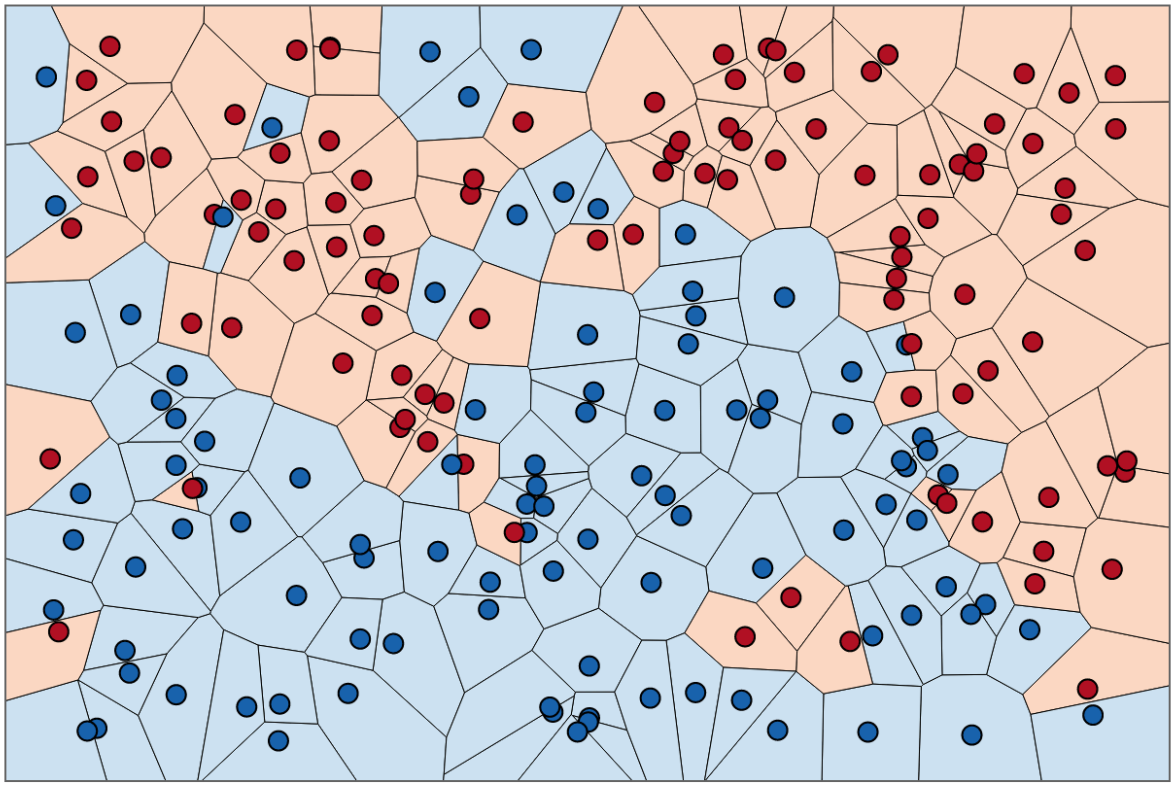
# Training set: observations of wealth and religiousness features with #blue or #red labels



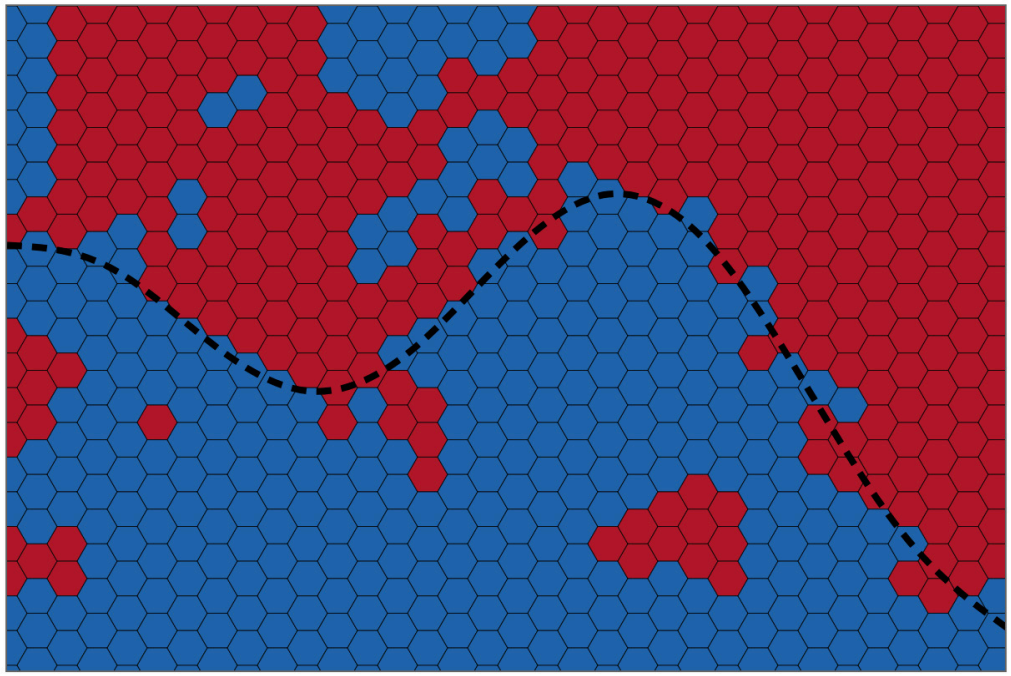
# Test set:  How well can we do?



# K-Nearest Neighbors (KNN) defines neighborhoods

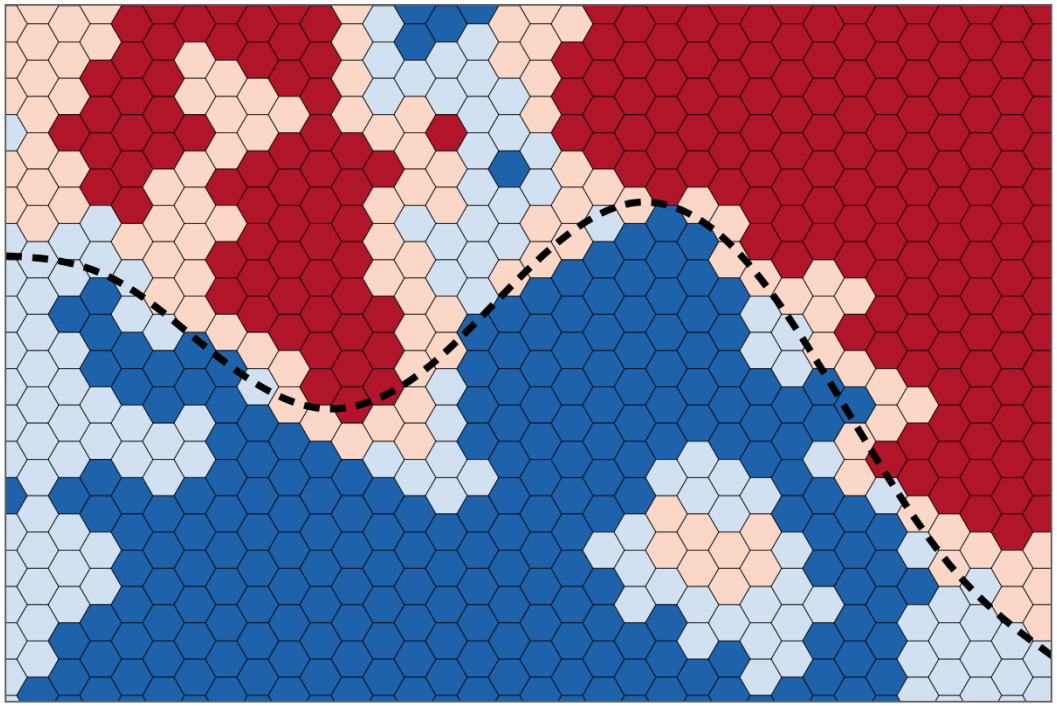


# K-Nearest Neighbors (KNN) k=1 classification results



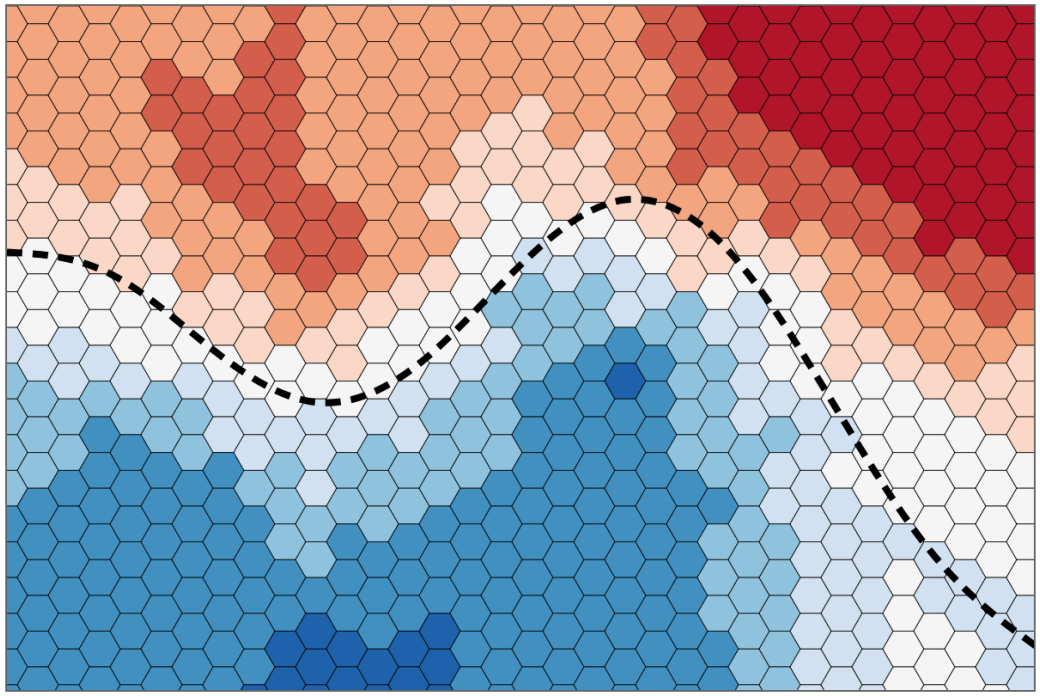
*k*-Nearest Neighbors: 1

# K-Nearest Neighbors (KNN) k=3 classification results



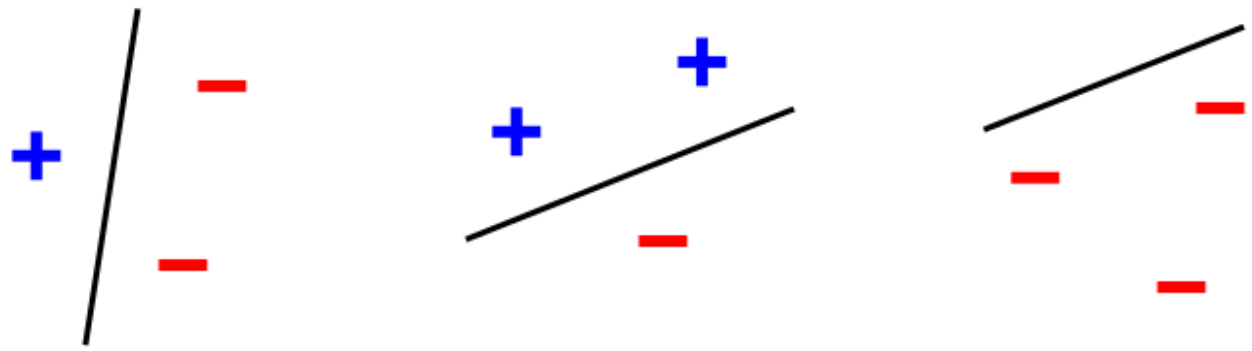
*k*-Nearest Neighbors: 3

# K-Nearest Neighbors (KNN) k=29 classification results

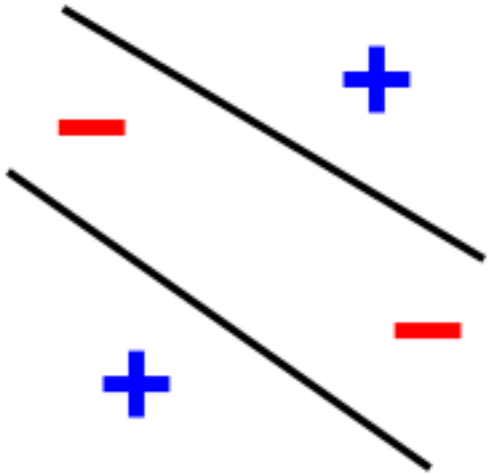


*k*-Nearest Neighbors: 29

A straight line can classify three points arbitrarily labeled



A straight line can not classify four points arbitrarily labeled



# Model capacity

The **capacity** (Vapnik-Chervonenkis dimension) of a model describes how many points can be correctly predicted when they are produced by an adversary

# The capacity of non-parametric models is defined by the size of their training set

- k-nearest neighbor (KNN) regression computes its output based upon the k “nearest” training examples
- Often the best method, and certainly a baseline to beat

The **generalizability** of a model describes its ability to perform well on previously unseen inputs

# We need to control model complexity for good generalization

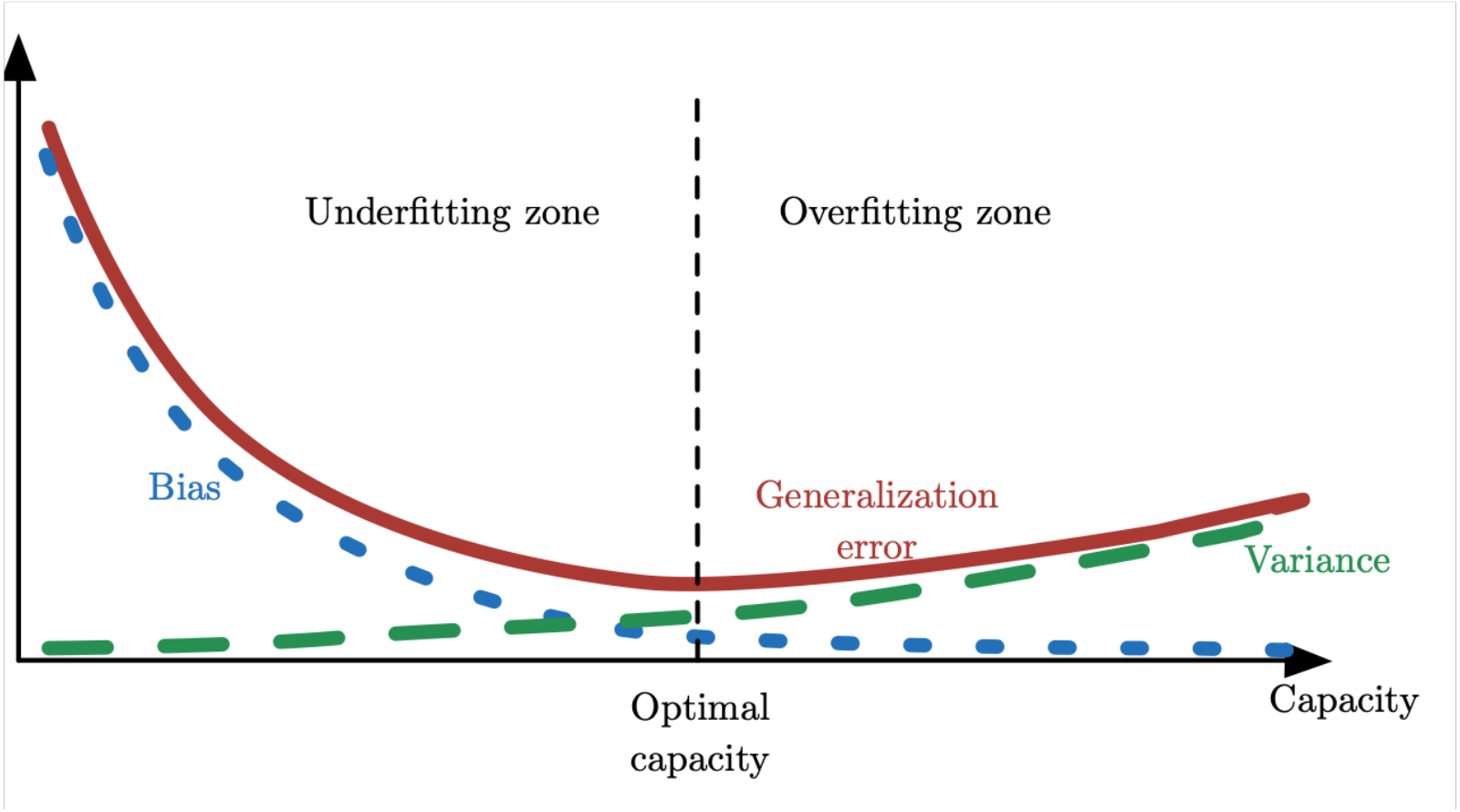
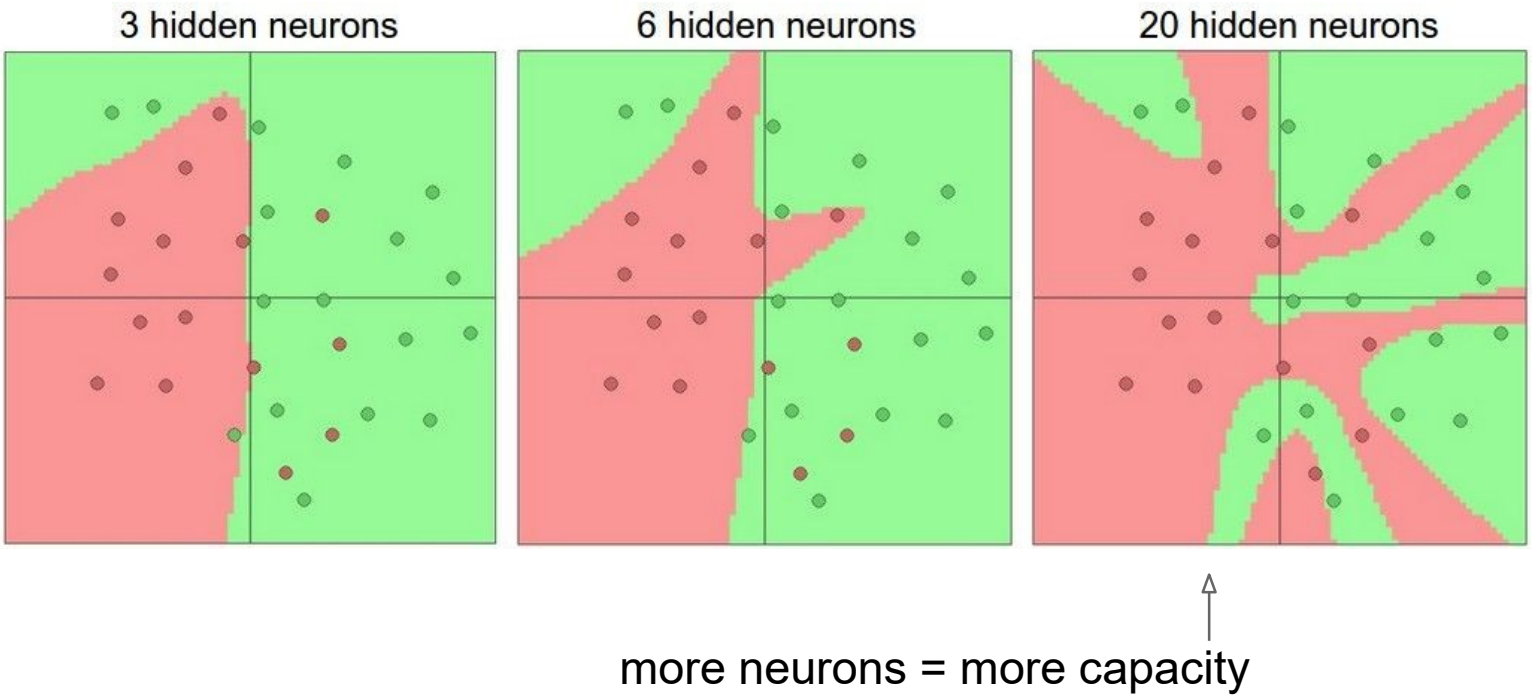



Figure 5.6

# Setting the number of layers and their sizes



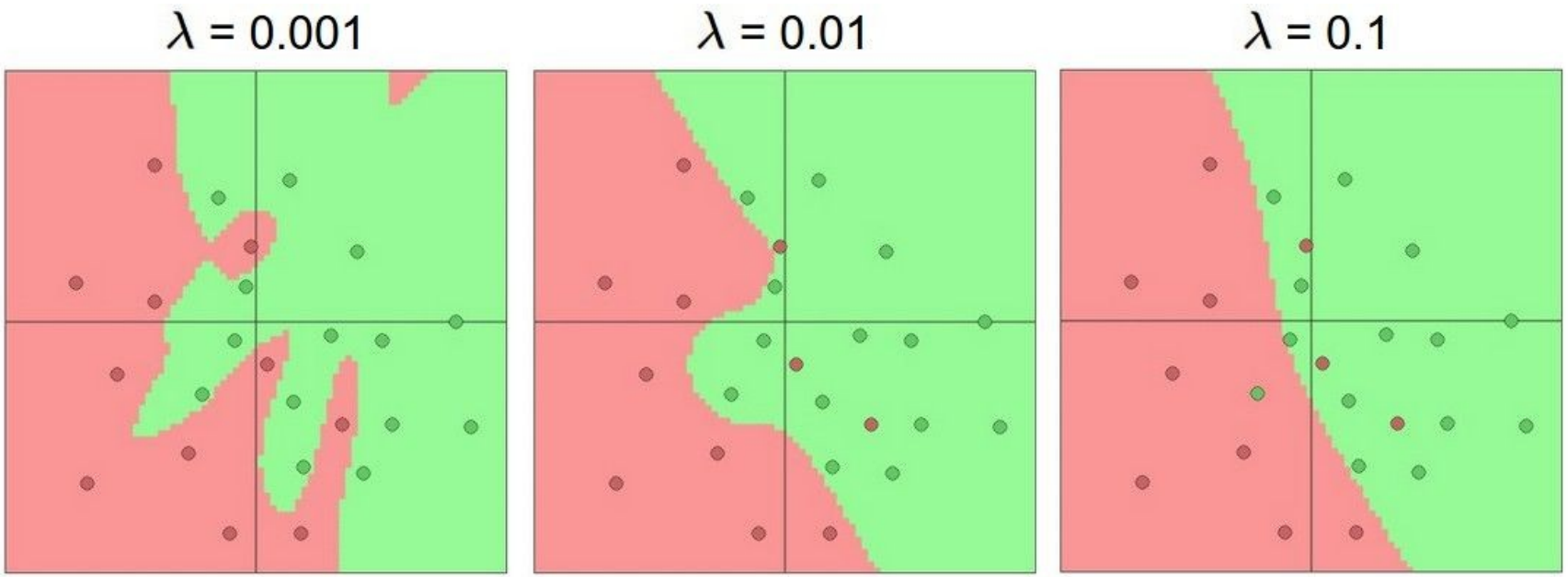
# We can regularize our parameters

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\frac{1}{2}\lambda \sum_k \sum_l W_{k,l}^2}_{\text{regularization loss}}$$

$$\frac{d}{dw}(\frac{1}{2}\lambda w^2) = \lambda w$$

http://cs231n.github.io/neural-networks-case-study/

# Parameter regularization instead of controlling model complexity




(you can play with this demo over at ConvNetJS:
http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

# Gradient of loss with respect to logits

$$z^i = W^T x^i + b \qquad p_k^i = \frac{e^{z_k^i}}{\sum_j e^{z_j^i}}$$

$$L_i = -\log(p_k^i) \; where k = y^i$$

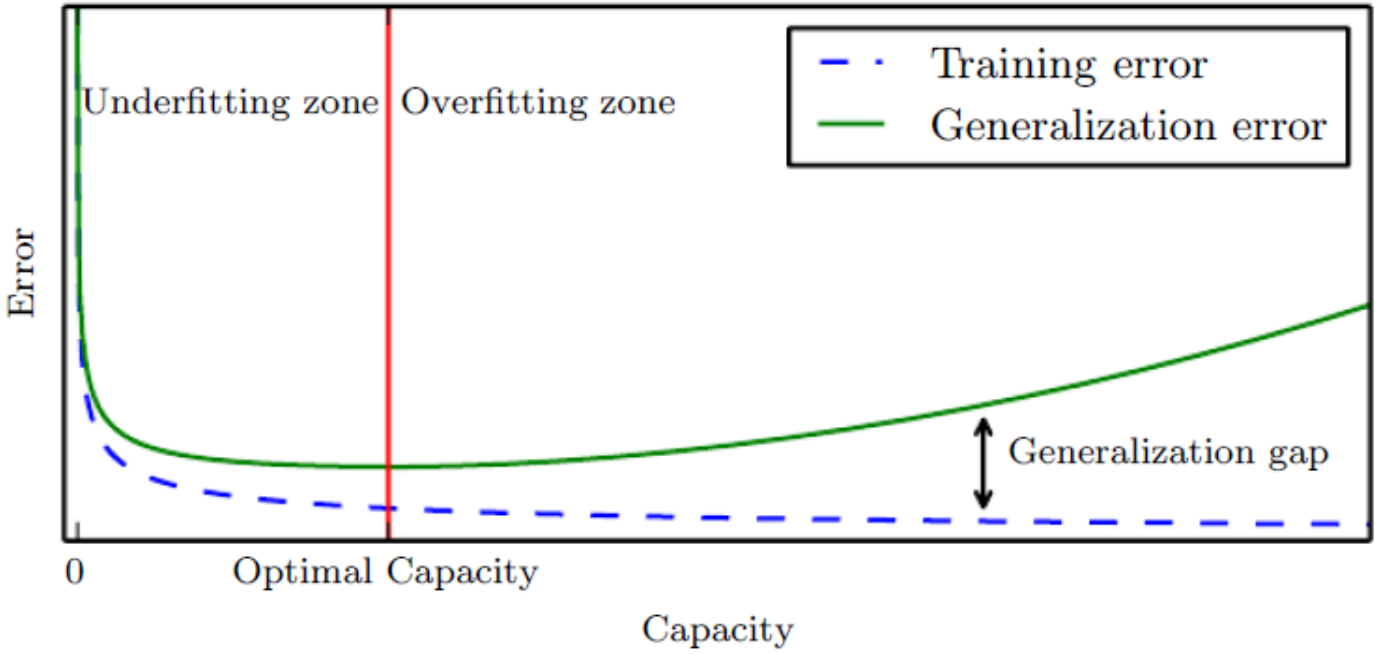
$$\frac{\partial L_i}{\partial z_j} = p_j^i - \mathbf{1}(\mathbf{y_i} = \mathbf{j}) \; Update \; weights \; for \; j$$

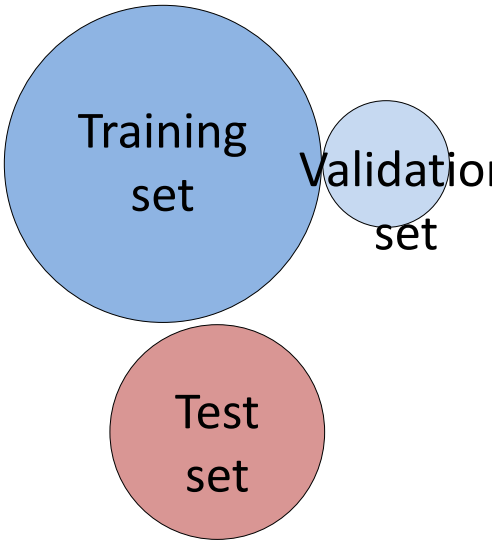
$$p^i = [0.6, 0.3, 0.1] \; then \; gradient \rightarrow [0.6, -0.4, 0.1]$$

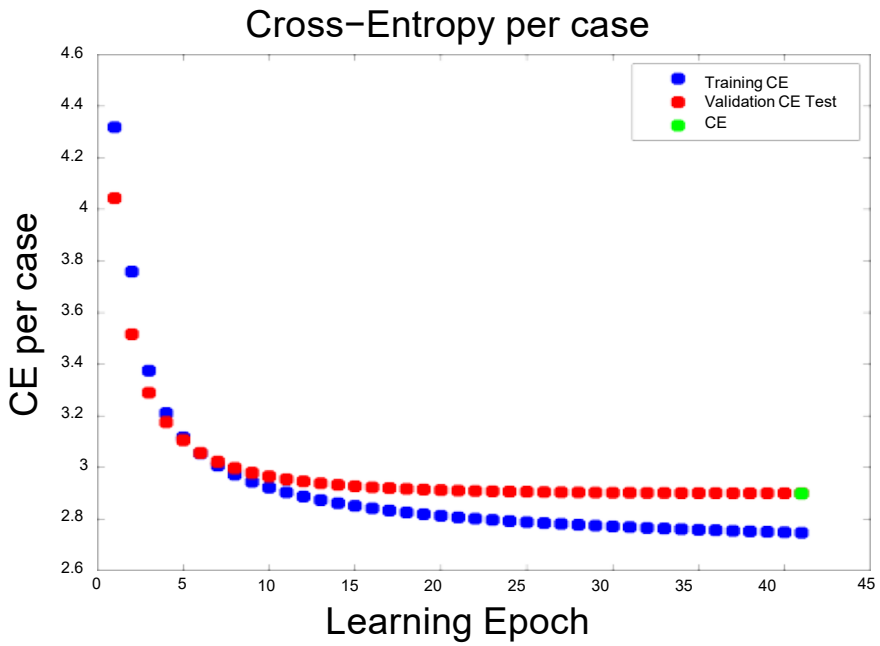
Correct Label

http://cs231n.github.io/neural-networks-case-study/

# Overfitting and its remedies: validation set, early stopping



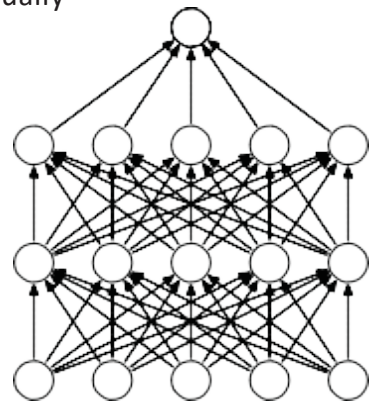




Leave out small "validation set"

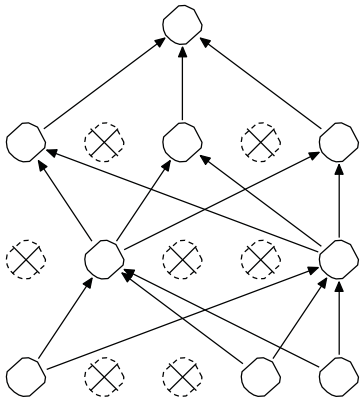
- not used to train the model
- used to evaluate model at each epoch/iteration (VCE, Validation cross-entropy)
- Stop when VCE increases, prevent overfitting

# Avoid overfitting: Regularization, Dropout training

Conceptually

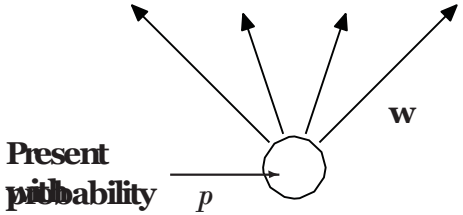


(a) Standard Neural Net

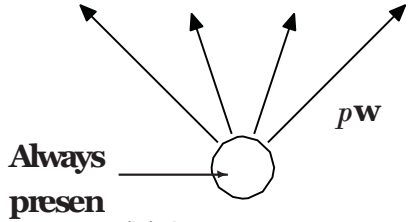


(b) After applying dropout.

In practice




(a) At training time

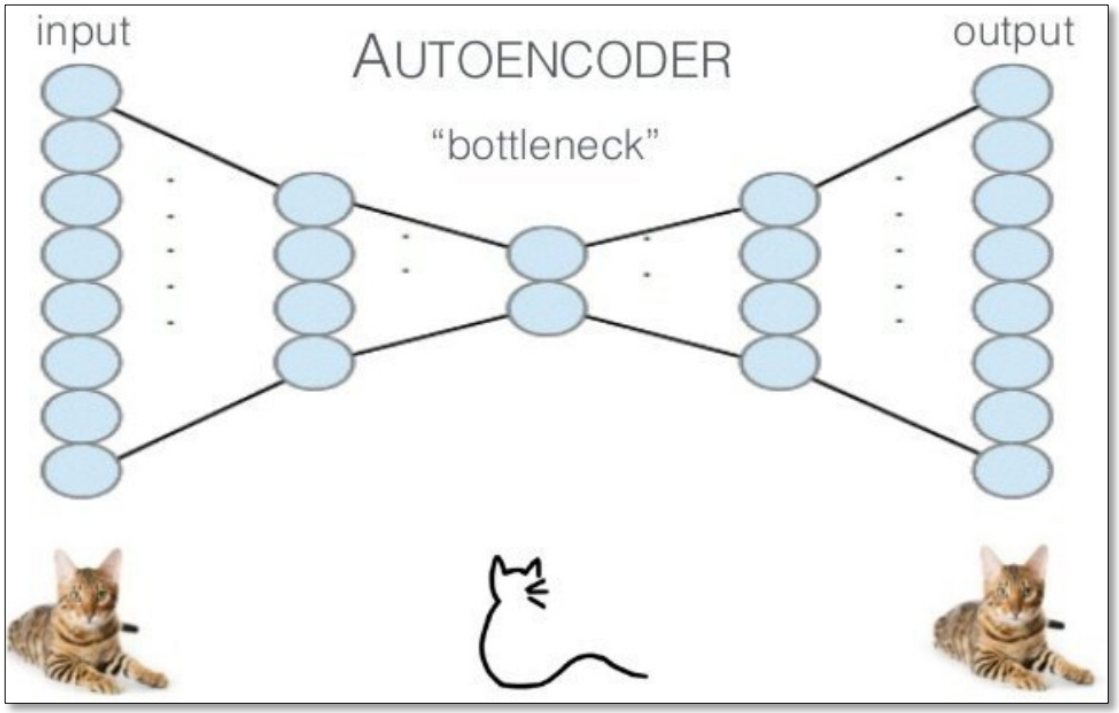



(b) At test time

[Srivastava et al., 2014]

- Regularization: recall linear (L1, lasso), quadratic (L2, ridge) or combination (elastic net) on parameters.
- Dropout is achieves parameter minimization in deep learning, by randomly dropping hidden units for different input points with some probability p.
- Train sub-network by back-propagation as usual.
- Equivalent to bagging (**b**ootstrap **agg**regat**ing**) an exponential number of models, each of which is missing some nodes
- Provides powerful regularization method, avoids overfitting in practice

# Autoencoder: dimensionality reduction with neural net




- Tricking a **supervised** learning algorithm to work in **unsupervised** fashion
- Feed input as output function to be learned. **But!** Constrain model complexity

- **Pretraining** with RBMs to learn representations for future supervised tasks. Use RBM output as "data" for training the next layer in stack
- After pretraining, "unroll" RBMs to create deep autoencoder
- Fine-tune using backpropagation

[Hinton *et al*, 2006]

# 4. Improving generalization

# Recurrent Neural Networks (RNNs) + Generalization

**Improving generalization**

- **More training data**
- **Tuning model capacity**
  - Architecture: # layers, # units
  - Early stopping: (validation set)
  - Weight-decay: L1/L2 regularization
  - Noise: Add noise as a regularizer
- **Bayesian prior on parameter distribution**
- **Why weight decay ⇔ Bayesian prior**
- **Variance of residual errors**

# Ways to reduce overfitting

- A large number of different methods have been developed.
  - Weight-decay
  - Weight-sharing
  - Early stopping
  - Model averaging
  - Bayesian fitting of neural nets
  - Dropout
  - Generative pre-training
- Many of these methods will be described in lecture 7.

# Reminder: Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains sampling error.
  - There will be accidental regularities just because of the particular training cases that were chosen.
- When we fit the model to the training set it cannot tell which regularities are real and which are caused by sampling error.
  - So it fits both kinds of regularity. If the model is very flexible it can model the sampling error really well.
- If you fitted the model to another training set drawn from the same distribution over cases, it would make different predictions on the test data. This is called “variance”.

# Preventing overfitting

- Approach 1: Get more data!
  - Almost always the best bet if data is cheap and you have enough compute power to train on more data.
- Approach 2: Use a model that has the right capacity:
  - enough to fit the true regularities.
  - not enough to also fit spurious regularities (if they are weaker).
- Approach 3: Average many different models.
  - Use models with different forms.
  - Or train the model on different subsets of the training data (this is called “bagging”).
- Approach 4: (Bayesian) Use a single neural network architecture, but average the predictions made by many different weight vectors.

# Get more data

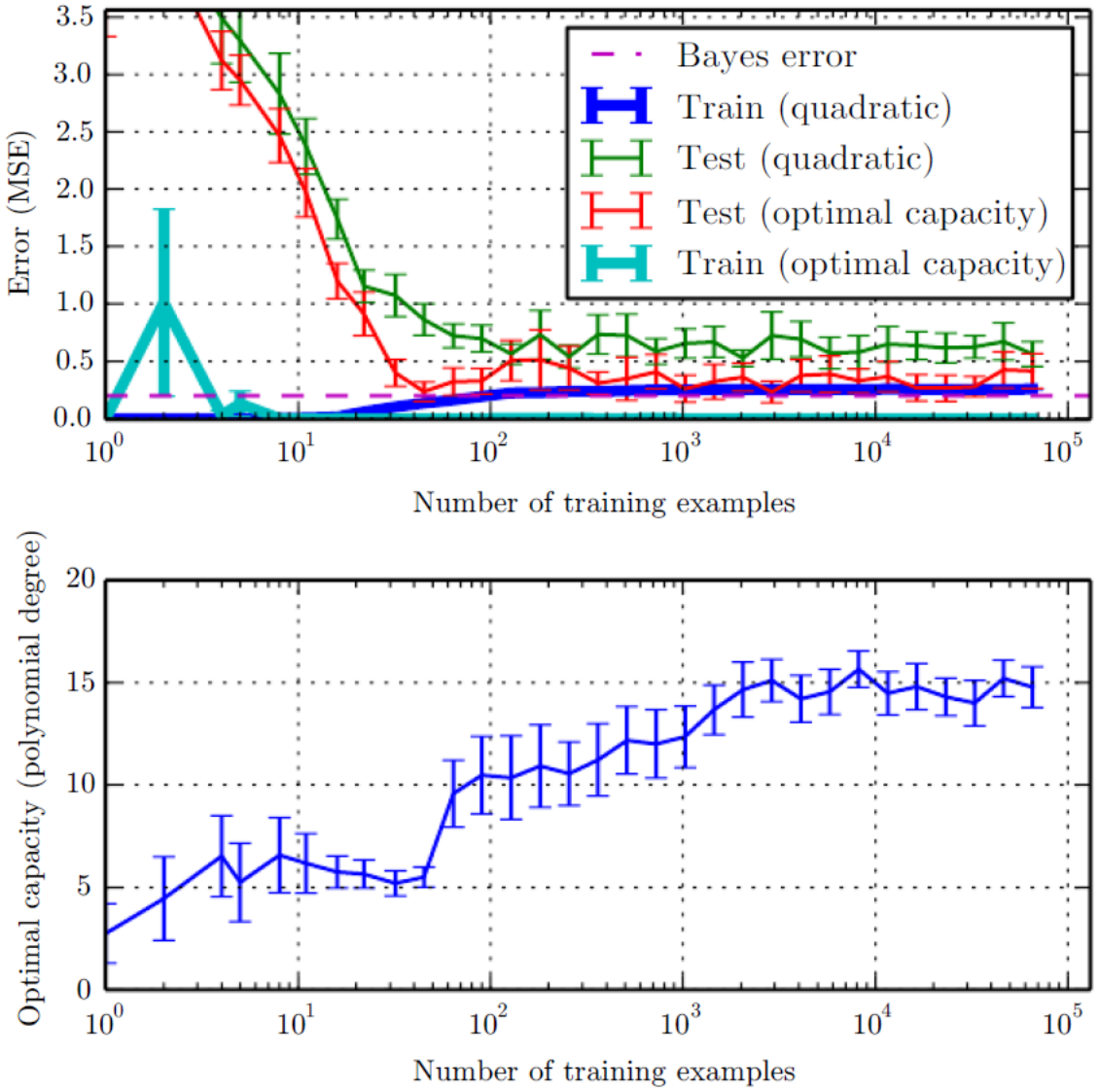



Figure 5.4: The effect of the training dataset size on the train and test error, as well as on the optimal model capacity. We constructed a synthetic regression problem based on adding a moderate amount of noise to a degree-5 polynomial, generated a single test set, and then generated several different sizes of training set. For each size, we generated 40 different training sets in order to plot error bars showing 95 percent confidence intervals. (Top)The MSE on the training and test set for two different models: a quadratic model, and a model with degree chosen to minimize the test error. Both are fit in closed form. For the quadratic model, the training error increases as the size of the training set increases. This is because larger datasets are harder to fit. Simultaneously, the test error decreases, because fewer incorrect hypotheses are consistent with the training data. The quadratic model does not have enough capacity to solve the task, so its test error asymptotes to a high value. The test error at optimal capacity asymptotes to the Bayes error. The training error can fall below the Bayes error, due to the ability of the training algorithm to memorize specific instances of the training set. As the training size increases to infinity, the training error of any fixed-capacity model (here, the quadratic model) must rise to at least the Bayes error. As the training (Bottom) set size increases, the optimal capacity (shown here as the degree of the optimal polynomial regressor) increases. The optimal capacity plateaus after reaching sufficient complexity to solve the task.

# 4. Improving generalization:

**a. Controlling model capacity**

➔ Architecture: # layers, # units

➔ Early stopping: (validation set)

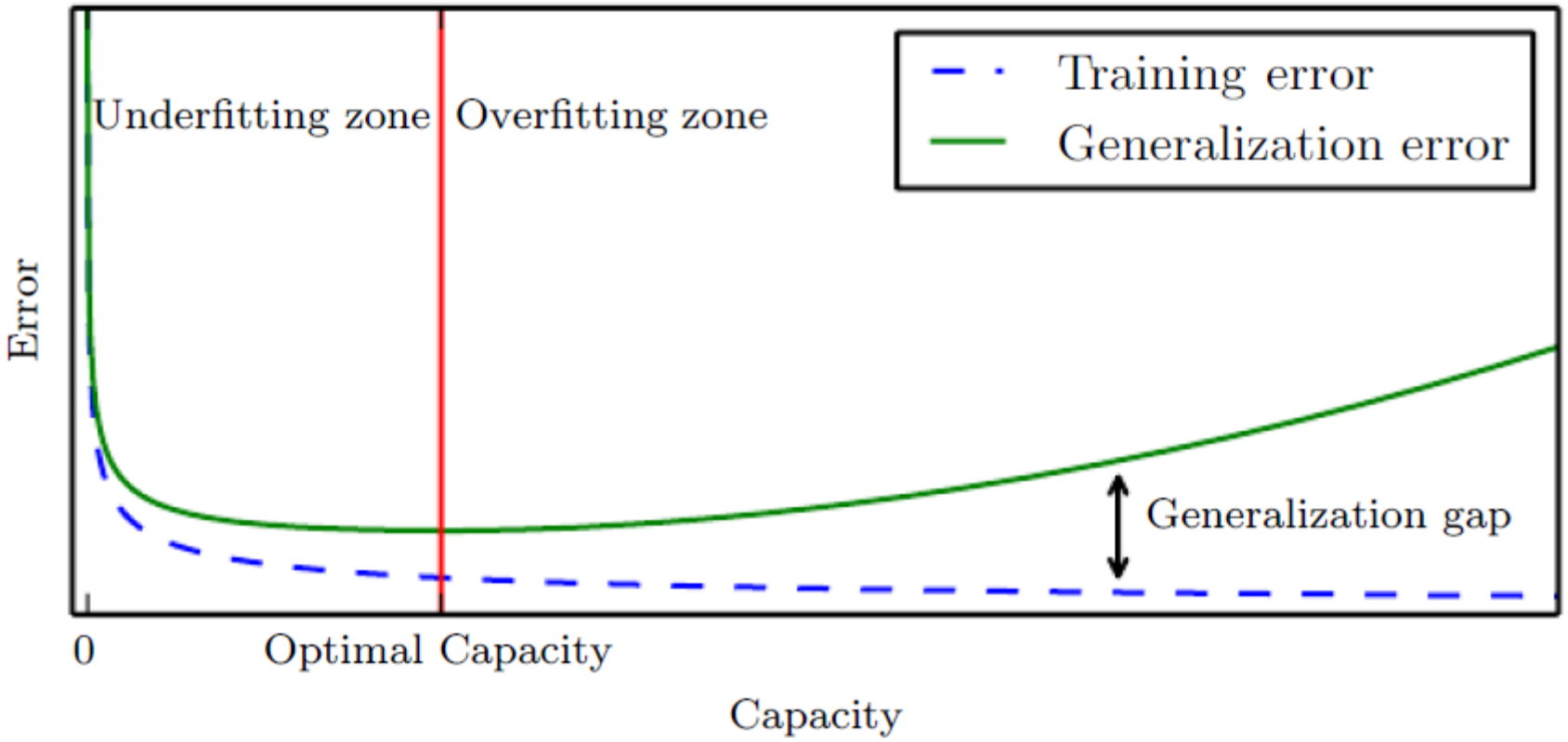
➔ Weight-decay: L1/L2 regularization

➔ Noise: Add noise

# Some ways to limit the capacity of a neural net

- The capacity can be controlled in many ways:
  - Architecture: Limit the number of hidden layers and the number of units per layer.
  - Early stopping: Start with small weights and stop the learning before it overfits.
  - Weight-decay: Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty).
  - Noise: Add noise to the weights or the activities.
- Typically, a combination of several of these methods is used.

# Effect of model capacity on generalization



# Tuning model capacity: Overfitting, underfitting

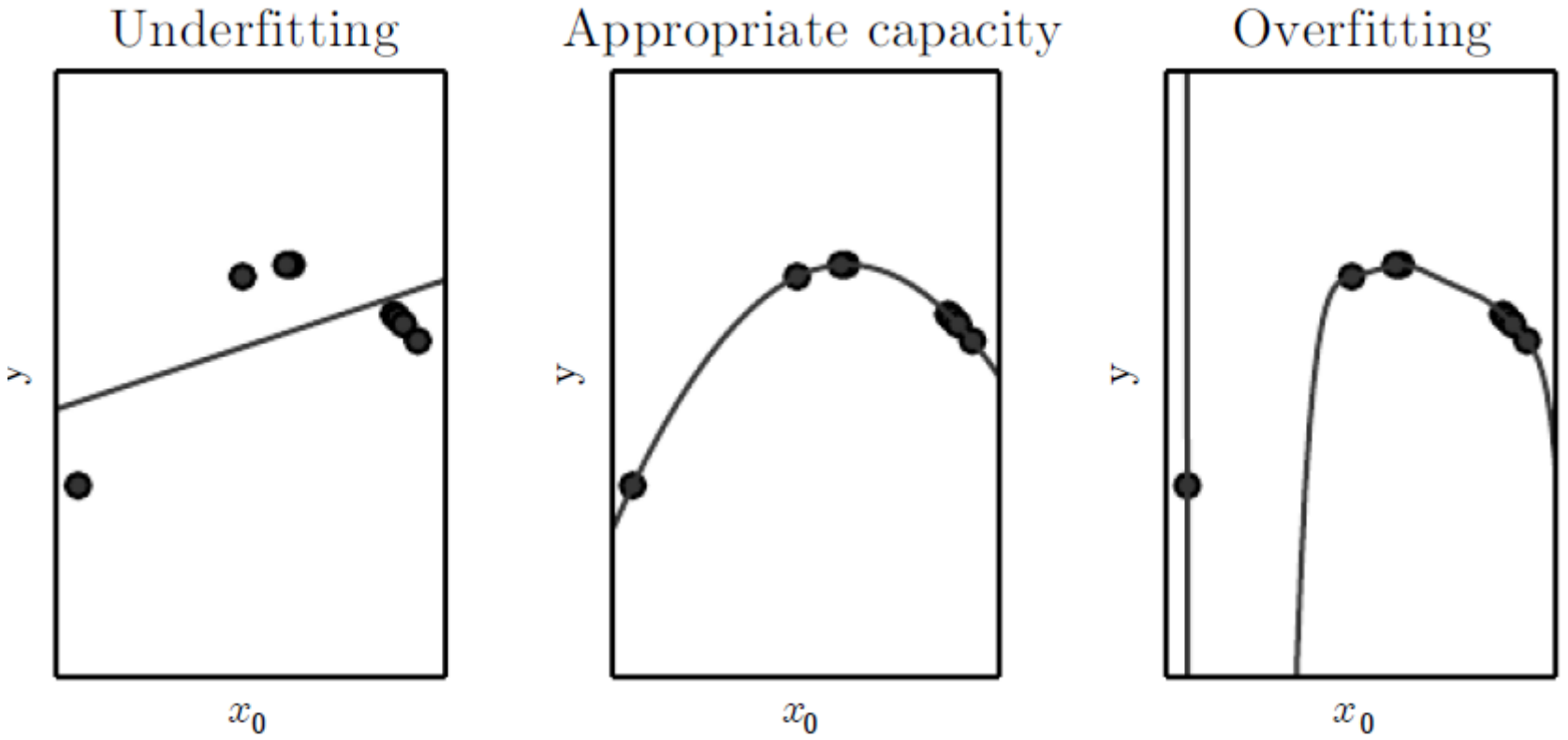



Figure 5.2: We fit three models to this example training set. The training data was generated synthetically, by randomly sampling x values and choosing y deterministically by evaluating a quadratic function. (Left)A linear function fit to the data suffers from underfitting—it cannot capture the curvature that is present in the data. (Center)A quadratic function fit to the data generalizes well to unseen points. It does not suffer from a significant amount of overfitting or underfitting. (Right)A polynomial of degree 9 fit to the data suffers from overfitting. Here we used the Moore-Penrose pseudoinverse to solve the underdetermined normal equations. The solution passes through all of the training points exactly, but we have not been lucky enough for it to extract the correct structure. It now has a deep valley in between two training points that does not appear in the true underlying function. It also increases sharply on the left side of the data, while the true function decreases in this area.

# 4a. Architecture hyperparams (# layers, # units)

# How to choose meta parameters that control capacity
## (like the number of hidden units or the size of the weight penalty)

- The wrong method is to try lots of alternatives and see which gives the best performance on the test set.
  - This is easy to do, but it gives a false impression of how well the method works.
  - The settings that work best on the test set are unlikely to work as well on a new test set drawn from the same distribution.
- An extreme example:
  Suppose the test set has random answers that do not depend on the input.
  - The best architecture will do better than chance on the test set.
  - But it cannot be expected to do better than chance on a new test set.

# Cross-validation: A better way to choose meta parameters

- Divide the total dataset into three subsets:
  - Training data is used for learning the parameters of the model.
  - Validation data is not used for learning but is used for deciding what settings of the meta parameters work best.
  - Test data is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data.
- We could divide the total dataset into one final test set and N other subsets and train on all but one of those subsets to get N different estimates of the validation error rate.
  - This is called N-fold cross-validation.
  - The N estimates are not independent.

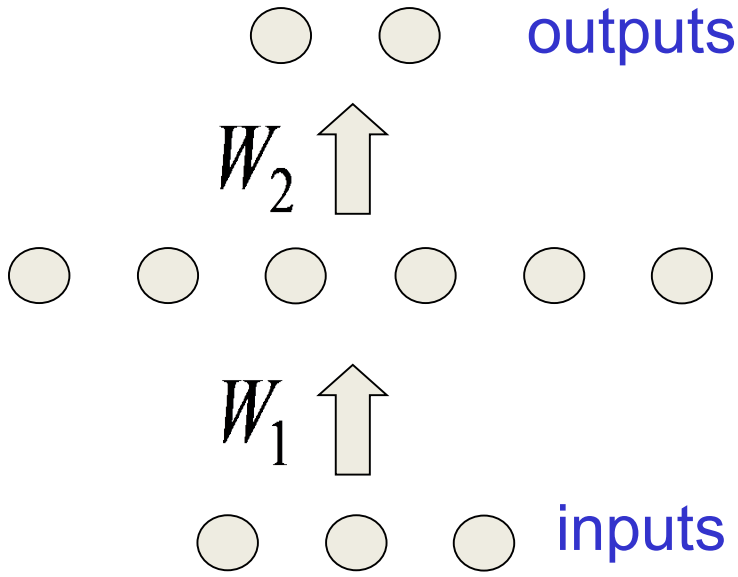
# 4b. Early stopping (training/validation/testing)

# Preventing overfitting by early stopping

- If we have lots of data and a big model, its very expensive to keep re-training it with different sized penalties on the weights or different architectures.
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse.
  - But it can be hard to decide when performance is getting worse.
- The capacity of the model is limited because the weights have not had time to grow big.
  - Smaller weights give the network less capacity. Why?

# Why early stopping works

- When the weights are very small, every hidden unit is in its linear range.
  - So even with a large layer of hidden units it's a linear model.
  - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.



# 4c. Weight regularization (L1, L2, Elastic Net)

# Effect of weight decay

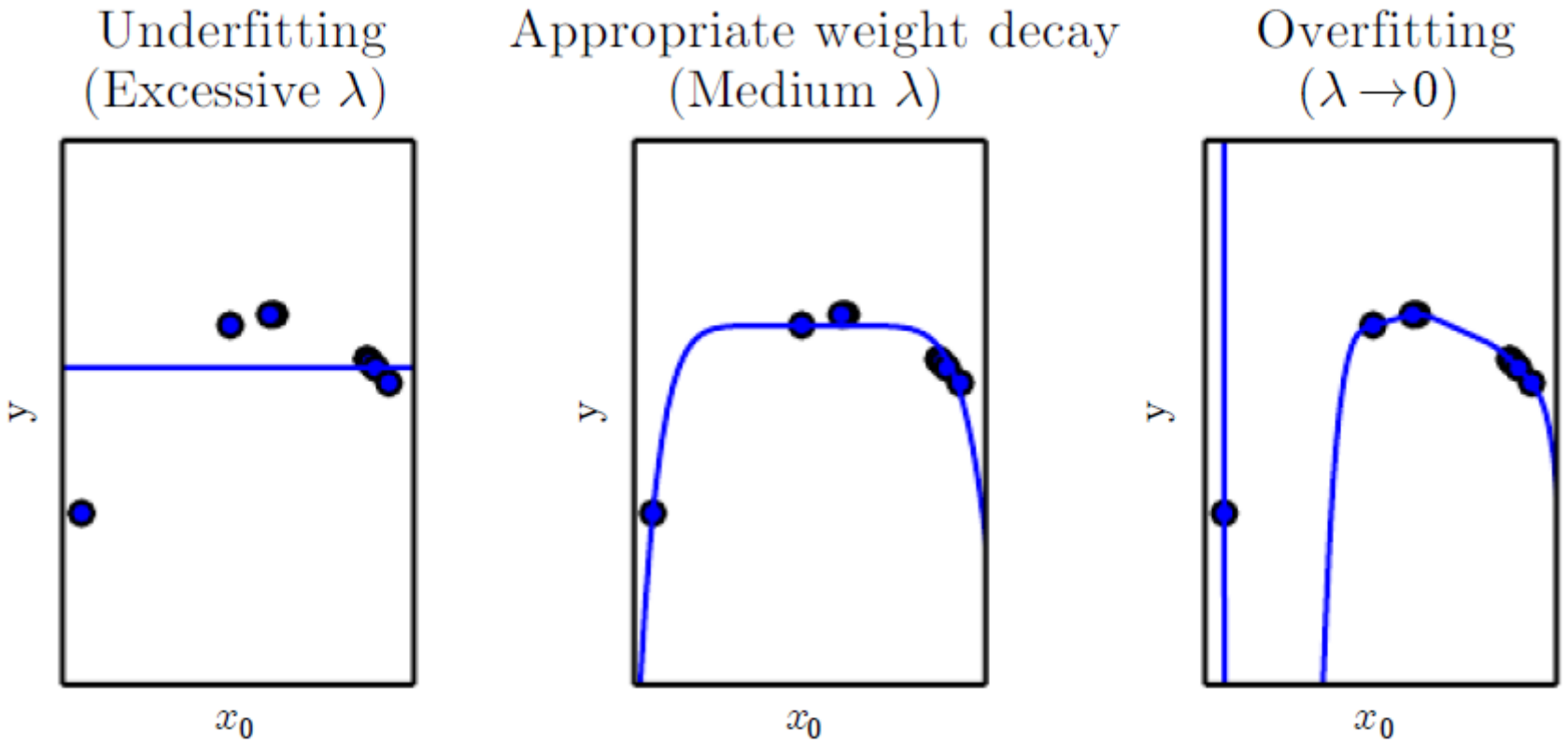


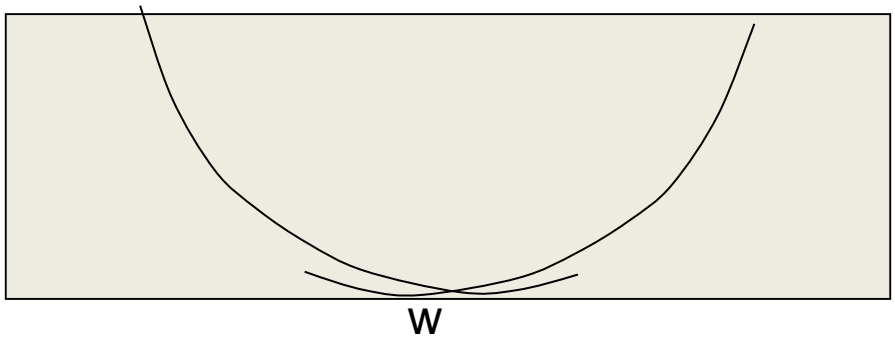

Figure 5.5. We fit a high-degree polynomial regression model to our example training set from figure 5.2. The true function is quadratic, but here we use only models with degree 9. We vary the amount of weight decay to prevent these high-degree models from overfitting. (Left)With very large λ, we can force the model to learn a function with no slope at all. This underfits because it can only represent a constant function. (Center)With a medium value of λ, the learning algorithm recovers a curve with the right general shape. Even though the model is capable of representing functions with much more complicated shape, weight decay has encouraged it to use a simpler function described by smaller coefficients. (Right)With weight decay approaching zero (i.e., using the Moore-Penrose pseudoinverse to solve the underdetermined problem with minimal regularization), the degree-9 polynomial overfits significantly, as we saw in figure 5.2

# Limiting the size of the weights

- The standard L2 weight penalty involves adding an extra term to the cost function that penalizes the squared weights.
  - This keeps the weights small unless they have big error derivatives.

$$C = E + \frac{\lambda}{2}\sum_i w_i^2$$

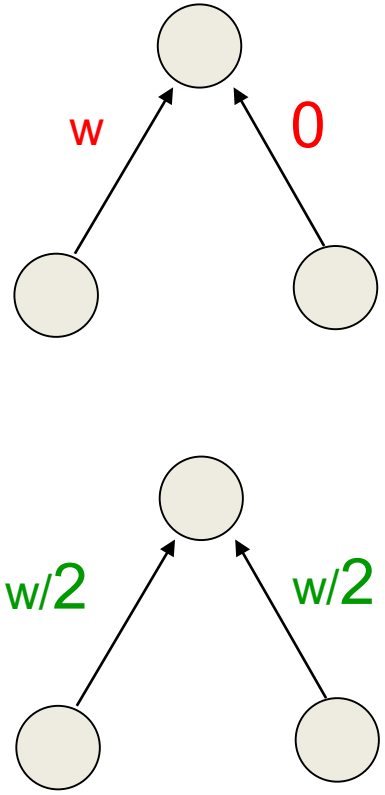
$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$




$$when \quad \frac{\partial C}{\partial w_i} = 0, \quad w_i = -\frac{1}{\lambda}\frac{\partial E}{\partial w_i}$$

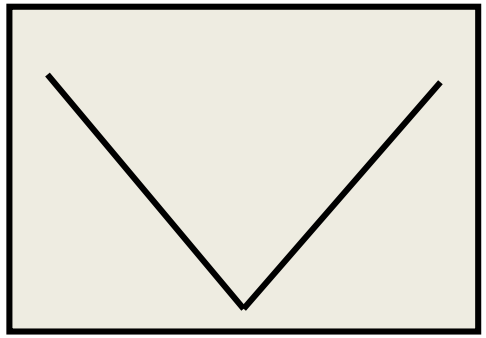
# The effect of L2 weight cost

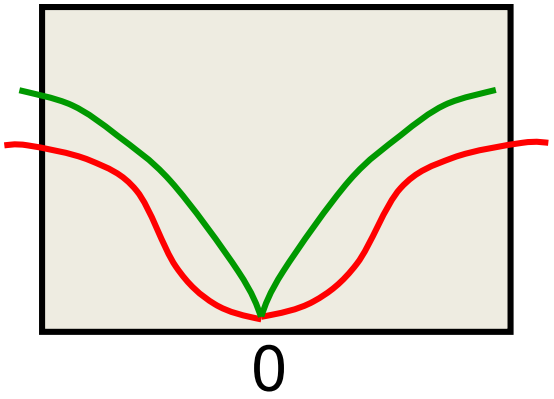
- It prevents the network from using weights that it does not need.
  - This can often improve generalization a lot because it helps to stop the network from fitting the sampling error.
  - It makes a smoother model in which the output changes more slowly as the input changes.
- If the network has two very similar inputs it prefers to put half the weight on each rather than all the weight on one.



# Other kinds of weight penalty

- Sometimes it works better to penalize the absolute values of the weights.
  - This can make many weights exactly equal to zero which helps interpretation a lot.
- Sometimes it works better to use a weight penalty that has negligible effect on large weights.
  - This allows a few large weights.



# Weight penalties vs weight constraints

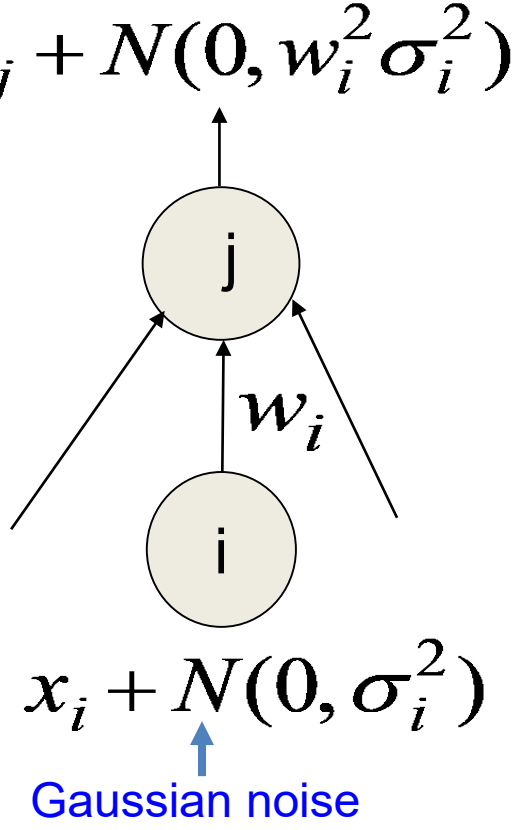
- We usually penalize the squared value of each weight separately.
- Instead, we can put a constraint on the maximum squared length of the incoming weight vector of each unit.
  - If an update violates this constraint, we scale down the vector of incoming weights to the allowed length.

- Weight constraints have several advantages over weight penalties.
  - Its easier to set a sensible value.
  - They prevent hidden units getting stuck near zero.
  - They prevent weights exploding.
- When a unit hits it's limit, the effective weight penalty on all of it's weights is determined by the big gradients.
  - This is more effective than a fixed penalty at pushing irrelevant weights towards zero.

# 4d. Adding noise (as a regularizer)

# L2 weight-decay via noisy inputs

- Suppose we add Gaussian noise to the inputs.
  - The variance of the noise is amplified by the squared weight before going into the next layer.
- In a simple net with a linear output unit directly connected to the inputs, the amplified noise gets added to the output.
- This makes an additive contribution to the squared error.
  - So minimizing the squared error tends to minimize the squared weights when the inputs are noisy.



output on one case →

$$y^{noisy} = \sum_i w_i x_i + \sum_i w_i \varepsilon_i \quad \text{where } \varepsilon_i \text{ is sampled from } N(0, \sigma_i^2)$$

$$E\left[(y^{noisy} - t)^2\right] = E\left[\left(y + \sum_i w_i \varepsilon_i - t\right)^2\right] = E\left[\left((y - t) + \sum_i w_i \varepsilon_i\right)^2\right]$$

$$= (y - t)^2 + E\left[2(y - t)\sum_i w_i \varepsilon_i\right] + E\left[\left(\sum_i w_i \varepsilon_i\right)^2\right]$$

$$= (y - t)^2 + E\left[\sum_i w_i^2 \varepsilon_i^2\right]$$

because $\varepsilon_i$ is independent of $\varepsilon_j$ and $\varepsilon_i$ is independent of $(y - t)$

$$= (y - t)^2 + \sum_i w_i^2 \sigma_i^2$$

So $\sigma_i^2$ is equivalent to an L2 penalty

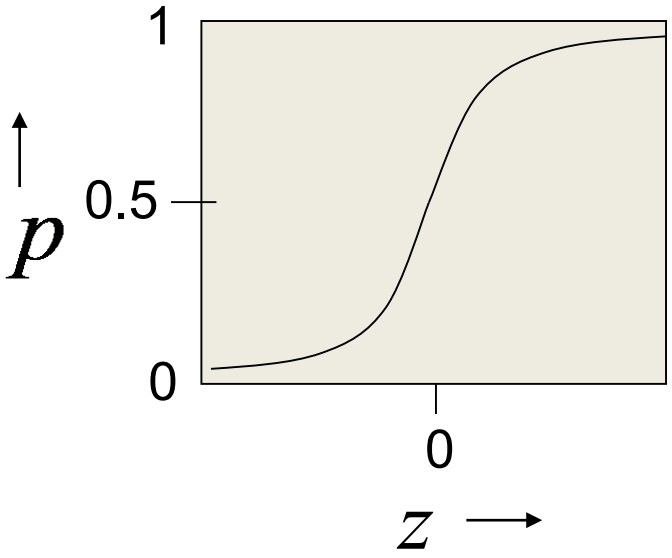
# Noisy weights in more complex nets

- Adding Gaussian noise to the weights of a multilayer non-linear neural net is not exactly equivalent to using an L2 weight penalty.
  - It may work better, especially in recurrent networks.
  - Alex Graves' recurrent net that recognizes handwriting, works significantly better if noise is added to the weights.

# Using noise in the activities as a regularizer

- Suppose we use backpropagation to train a multilayer neural net composed of logistic units.
  - What happens if we make the units binary and stochastic on the forward pass, but do the backward pass as if we had done the forward pass "properly"?
- It does worse on the training set and trains considerably slower.
  - But it does significantly better on the test set! (unpublished result).

$$p(s=1) = \frac{1}{1+e^{-z}}$$



# 4e. Prior distribution on params (Bayesian fitting)

# The Bayesian framework

- The Bayesian framework assumes that we always have a prior distribution for everything.
  - The prior may be very vague.
  - When we see some data, we combine our prior distribution with a likelihood term to get a posterior distribution.
  - The likelihood term takes into account how probable the observed data is given the parameters of the model.
    - It favors parameter settings that make the data likely.
    - It fights the prior
    - With enough data the likelihood terms always wins.

# A coin tossing example

- Suppose we know nothing about coins except that each tossing event produces a head with some unknown probability p and a tail with probability 1-p.
  - Our model of a coin has one parameter, p.

- Suppose we observe 100 tosses and there are 53 heads. What is p?

- The frequentist answer (also called maximum likelihood): Pick the value of p that makes the observation of 53 heads and 47 tails most probable.
  - This value is p=0.53

# A coin tossing example: the math

probability of a particular sequence containing 53 heads and 47 tails.

$$P(D) = p^{53}(1-p)^{47}$$

$$\frac{dP(D)}{dp} = 53p^{52}(1-p)^{47} - 47p^{53}(1-p)^{46}$$

$$= \left(\frac{53}{p} - \frac{47}{1-p}\right)\left[p^{53}(1-p)^{47}\right]$$

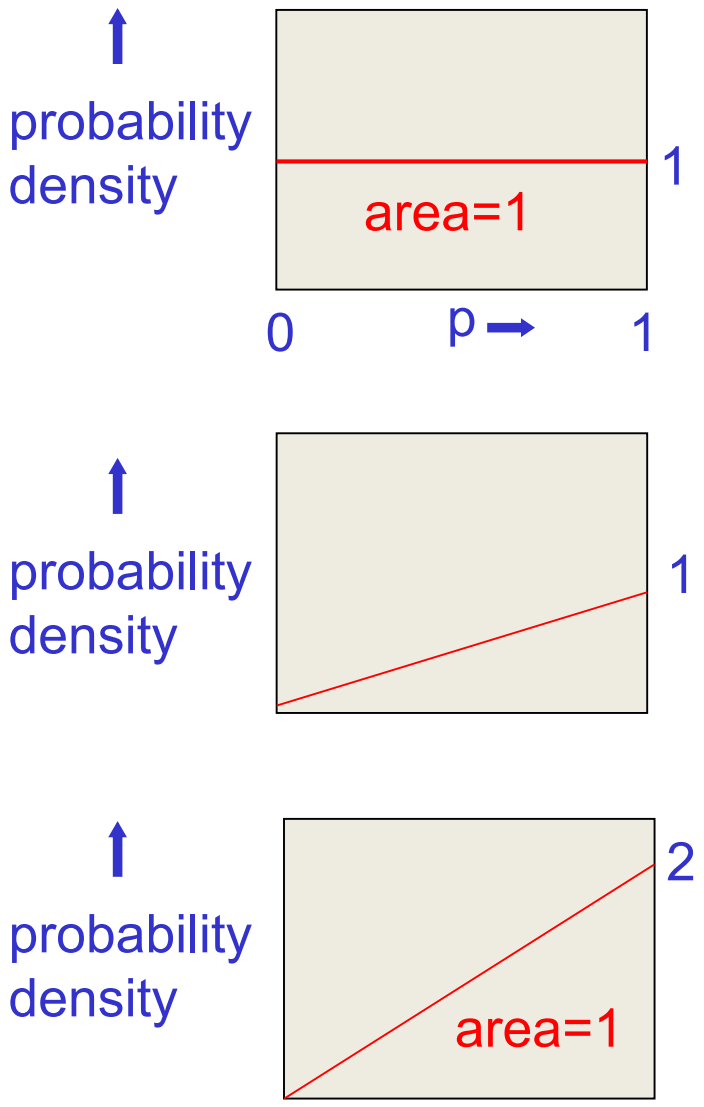
$$= 0 \;\; if \;\; p = .53$$

# Some problems with picking the parameters that are most likely to generate the data

- What if we only tossed the coin once and we got 1 head?
  - Is p=1 a sensible answer?
  - Surely p=0.5 is a much better answer.
- Is it reasonable to give a single answer?
  - If we don't have much data, we are unsure about p.
  - Our computations of probabilities will work much better if we take this uncertainty into account.

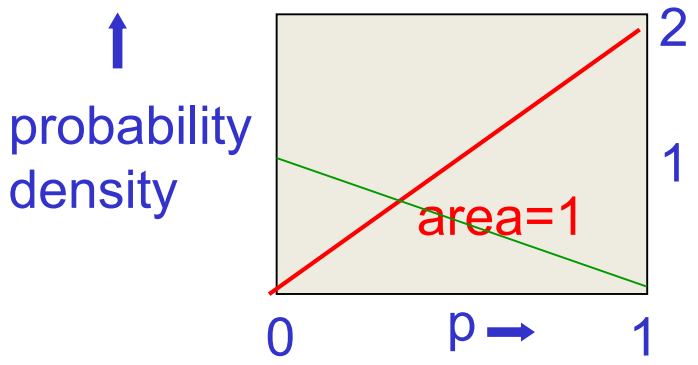
# Using a distribution over parameter values

- Start with a prior distribution over p. In this case we used a uniform distribution.
- Multiply the prior probability of each parameter value by the probability of observing a head given that value.
- Then scale up all of the probability densities so that their integral comes to 1. This gives the posterior distribution.

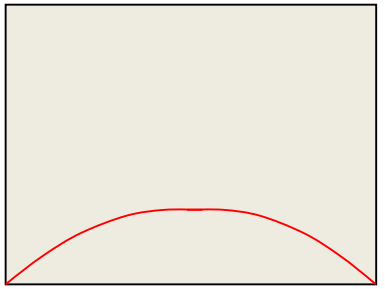


# Lets do it again: Suppose we get a tail

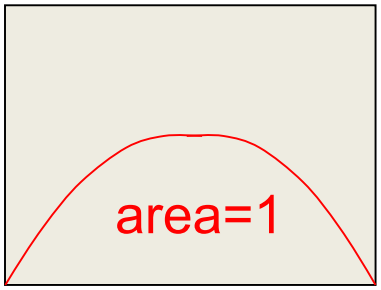
- Start with a prior distribution over p.




- Multiply the prior probability of each parameter value by the probability of observing a tail given that value.

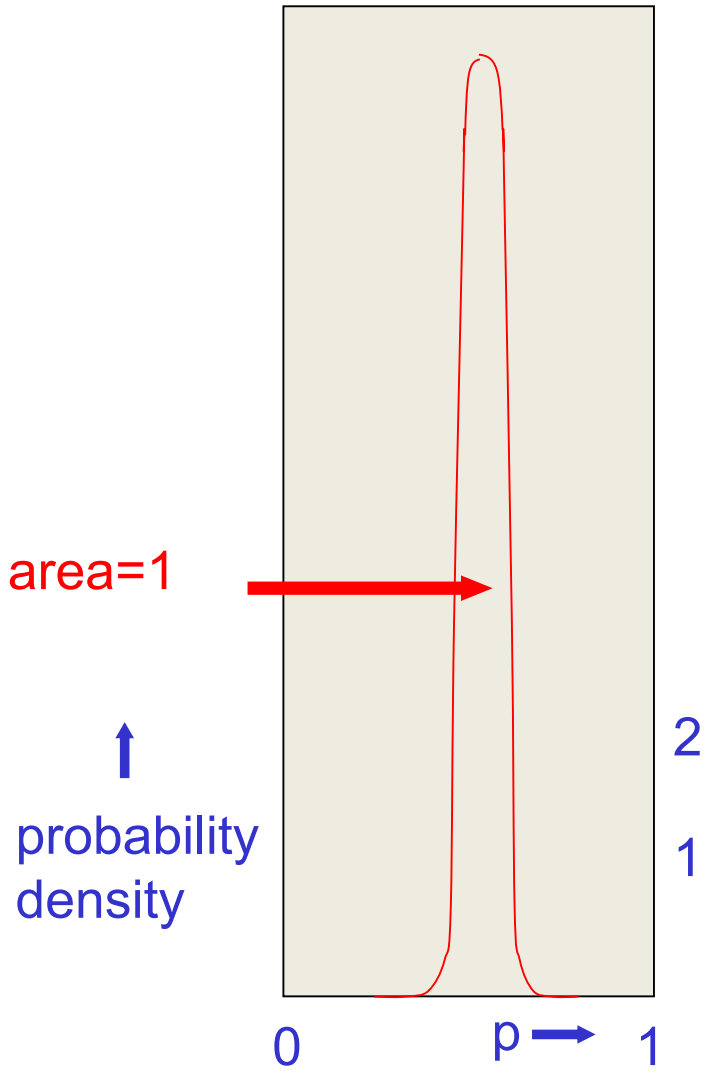


- Then renormalize to get the posterior distribution. Look how sensible it is!



# Lets do it another 98 times

- After 53 heads and 47 tails we get a very sensible posterior distribution that has its peak at 0.53 (assuming a uniform prior).



# Bayes Theorem

joint probability

conditional probability

$$p(D)p(W \mid D) = p(D,W) = p(W)p(D \mid W)$$

prior probability of weight vector W

probability of observed data given W

$$p(W \mid D) = \frac{p(W)\ \ p(D \mid W)}{p(D)}$$

posterior probability of weight vector W given training data D

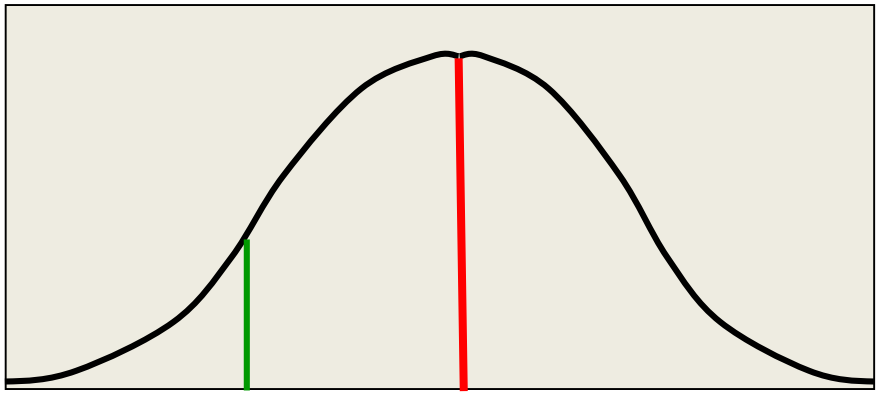
$$\int_W p(W)p(D \mid W)$$

# 4c+4e. Why weight decay is Bayesian regularization

# Supervised Maximum Likelihood Learning

- Finding a weight vector that minimizes the squared residuals is equivalent to finding a weight vector that maximizes the log probability density of the correct answer.

- We assume the answer is generated by adding Gaussian noise to the output of the neural network.



t
correct answer

y
model's estimate of most probable value

# Supervised Maximum Likelihood Learning

output of the net → $y_c = f(input_c, W)$

probability density of the target value given the net's output plus Gaussian noise →

$$p(t_c \mid y_c) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_c - y_c)^2}{2\sigma^2}}$$

← Gaussian distribution centered at the net's output

Cost → $$-\log p(t_c \mid y_c) = k + \frac{(t_c - y_c)^2}{2\sigma^2}$$

Minimizing squared error is the same as maximizing log prob under a Gaussian.

# MAP: Maximum a Posteriori

- The proper Bayesian approach is to find the full posterior distribution over all possible weight vectors.
  - If we have more than a handful of weights this is hopelessly difficult for a non-linear net.
  - Bayesians have all sort of clever tricks for approximating this horrendous distribution.
- Suppose we just try to find the most probable weight vector.
  - We can find an optimum by starting with a random weight vector and then adjusting it in the direction that improves $p(W \mid D)$.
  - But it's only a local optimum.
- It is easier to work in the log domain. If we want to minimize a cost we use negative log probs

# Why we maximize sums of log probabilities

- We want to maximize the product of the probabilities of the producing the target values on all the different training cases.
  - Assume the output errors on different cases, c, are independent.

$$p(D \mid W) = \prod_c p(t_c \mid W) = \prod_c p\left(t_c \mid f(input_c, W)\right)$$

- Because the log function is monotonic, it does not change where the maxima are. So we can maximize sums of log probabilities

$$\log p(D \mid W) = \sum_c \log p(t_c \mid W)$$

# MAP: Maximum a Posteriori

$$p(W \mid D) = p(W)\, p(D \mid W) \,/\, p(D)$$

$$Cost = -\log p(W \mid D) = -\log p(W) - \log p(D \mid W) + \log p(D)$$

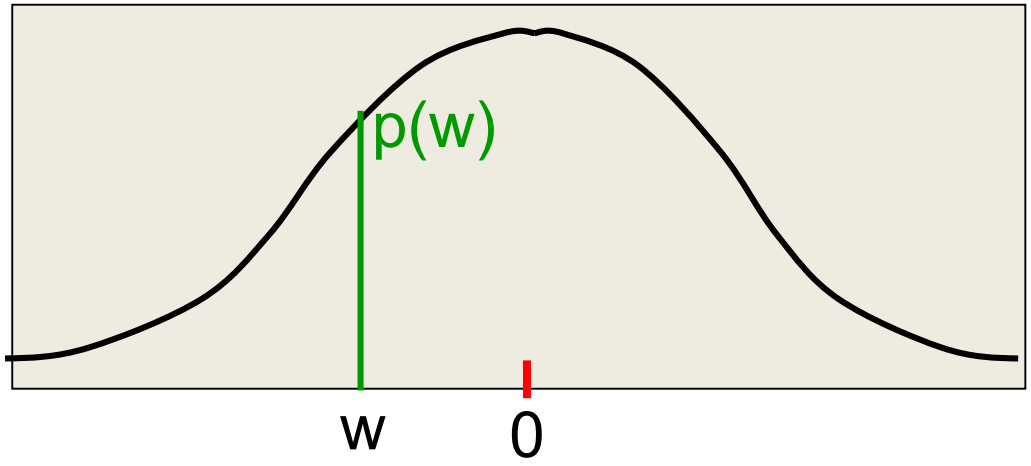
log prob of W under the prior

log prob of target values given W

This is an integral over all possible weight vectors so it does not depend on W

# The log probability of a weight under its prior

- Minimizing the squared weights is equivalent to maximizing the log probability of the weights under a zero-mean Gaussian prior.




$$p(w) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{w^2}{2\sigma_W^2}}$$

$$-\log p(w) = \frac{w^2}{2\sigma_W^2} + k$$

# The Bayesian interpretation of weight decay

$$-\log p(W \mid D) = -\log p(D \mid W) - \log p(W) + \log p(D)$$

constant

$$C^* = \frac{1}{2\sigma_D^2}\sum_c (y_c - t_c)^2 + \frac{1}{2\sigma_W^2}\sum_i w_i^2$$

assuming that the model makes a Gaussian prediction

assuming a Gaussian prior for the weights

$$C = E + \frac{\sigma_D^2}{\sigma_W^2}\sum_i w_i^2$$

So the correct value of the weight decay parameter is the ratio of two variances. It's not just an arbitrary hack.

# 4f. Variance of residual errors (MacKay's quick and dirty method)

# Estimating the variance of the output noise

- After we have learned a model that minimizes the squared error, we can find the best value for the output noise.
  - The best value is the one that maximizes the probability of producing exactly the correct answers after adding Gaussian noise to the output produced by the neural net.
  - The best value is found by simply using the variance of the residual errors.

# Estimating the variance of the Gaussian prior on the weights

- After learning a model with some initial choice of variance for the weight prior, we could do a dirty trick called “empirical Bayes”.
  - Set the variance of the Gaussian prior to be whatever makes the weights that the model learned most likely.
    - i.e. use the data itself to decide what your prior is!
  - This is done by simply fitting a zero-mean Gaussian to the one-dimensional distribution of the learned weight values.
    - We could easily learn different variances for different sets of weights.
- We don’t need a validation set!

# MacKay's quick and dirty method of choosing the ratio of the noise variance to the weight prior variance.

- Start with guesses for both the noise variance and the weight prior variance.
- While not yet bored
  - Do some learning using the ratio of the variances as the weight penalty coefficient.
  - Reset the noise variance to be the variance of the residual errors.
  - Reset the weight prior variance to be the variance of the distribution of the actual learned weights.
- Go back to the start of this loop.

# Recurrent Neural Networks (RNNs) + Generalization

1. **How do you read/listen/understand/write? Can machines do that?**
   - Context matters: characters, words, letters, sounds, completion, multi-modal
   - Predicting next word/image: from unsupervised learning to supervised learning
2. **Encoding temporal context: Hidden Markov Models (HMMs), RNNs**
   - Primitives: hidden state, memory of previous experiences, limitations of HMMs
   - RNN architectures,unrolling,back-propagation-through-time(BPTT),param reuse
3. **Vanishing gradients, Long-Short-Term Memory (LSTM)**
   - Key idea: gated input/output/memory nodes, model choose to forget/remember
   - Example: online character recognition with LSTM recurrent neural network
4. **Improving generalization**
   - **More training data**
   - **Tuning model capacity**
     - Architecture: # layers, # units
     - Early stopping: (validation set)
     - Weight-decay: L1/L2 regularization
     - Noise: Add noise as a regularizer
   - **Bayesian prior on parameter distribution**
   - **Why weight decay ⇔ Bayesian prior**
   - **Variance of residual errors**

# ML foundations

- neural networks (NNs)
  - convolutional neural networks (CNNs)
  - recurrent neural networks (RNNs)
  - residual neural networks
  - (variational) autoencoders (VAEs)
  - generative adversarial networks (GANs)
- regularization
  - $L_1$ regularization
  - $L_2$ regularization
  - dropout
  - early stopping
- model selection
  - cross-validation (CV)
  - Akaike information criterion (AIC)
  - Bayesian information criterion (BIC)
- model interpretation methods
  - sufficient input subsets (SIS)
  - saliency maps
- model uncertainty
  - identifying out of distribution inputs
  - ensembles and calibrated uncertainty
- dimensionality reduction methods
  - principal component analysis (PCA)
  - t-SNE
  - autoencoders
  - non-negative matrix factorization (NMF)
- hyperparameter optimization and AutoML

# Additional slides

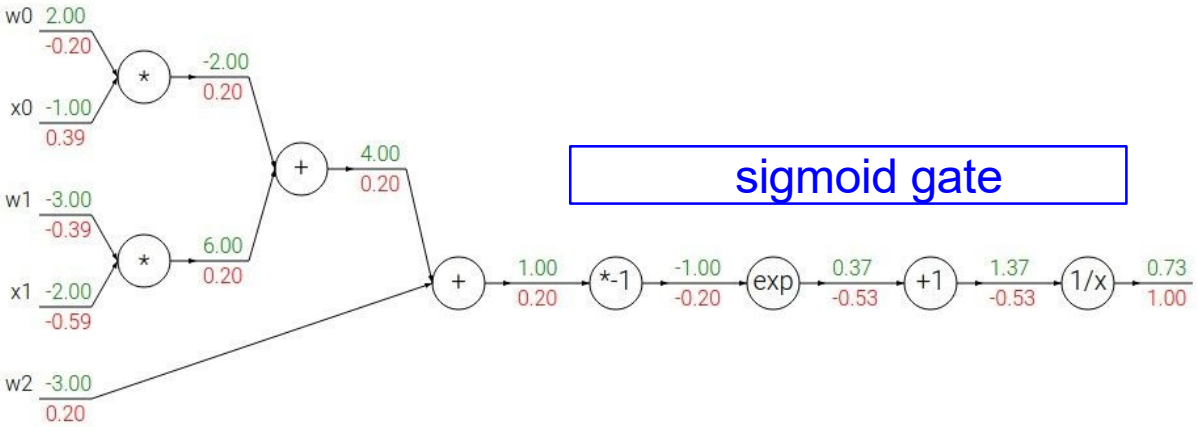
# Symbolic differentiation can save work and improve accuracy

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

sigmoid function

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$



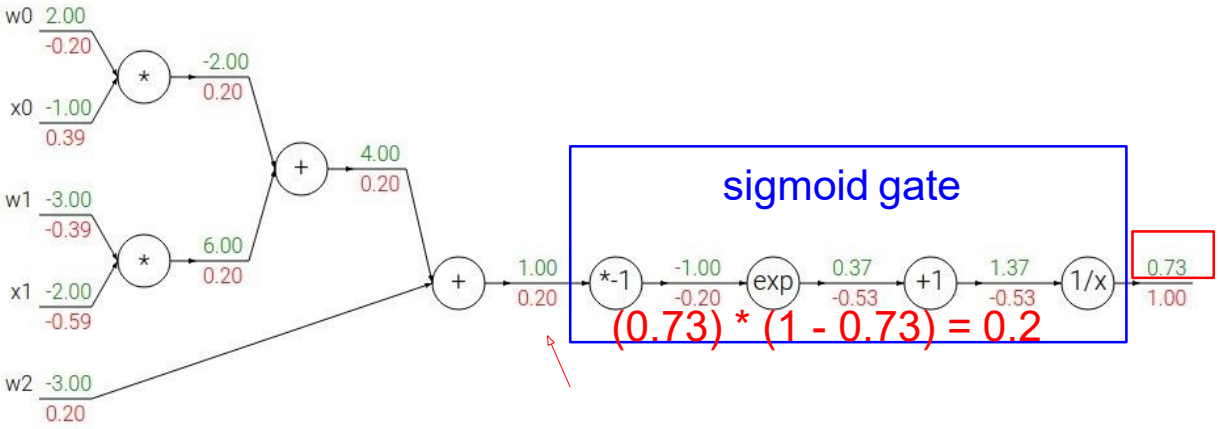

sigmoid gate

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

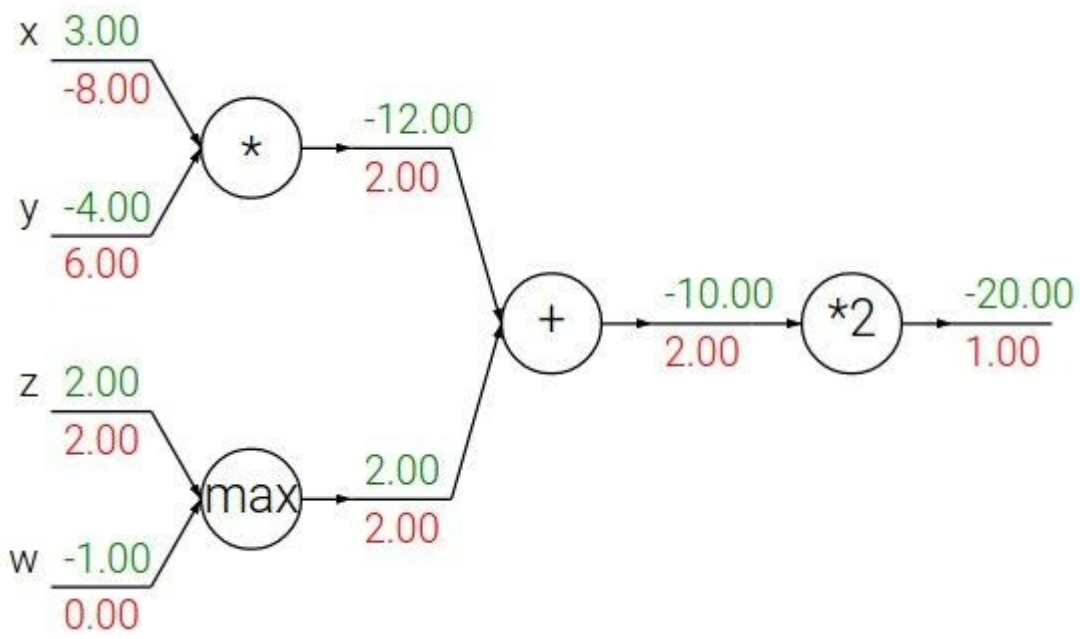
sigmoid function

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$

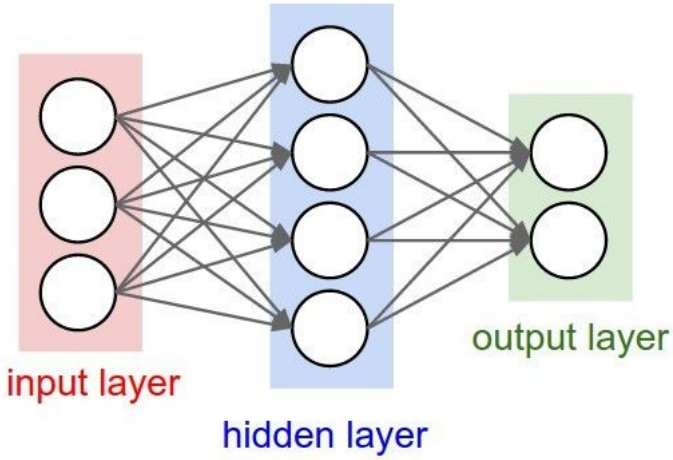


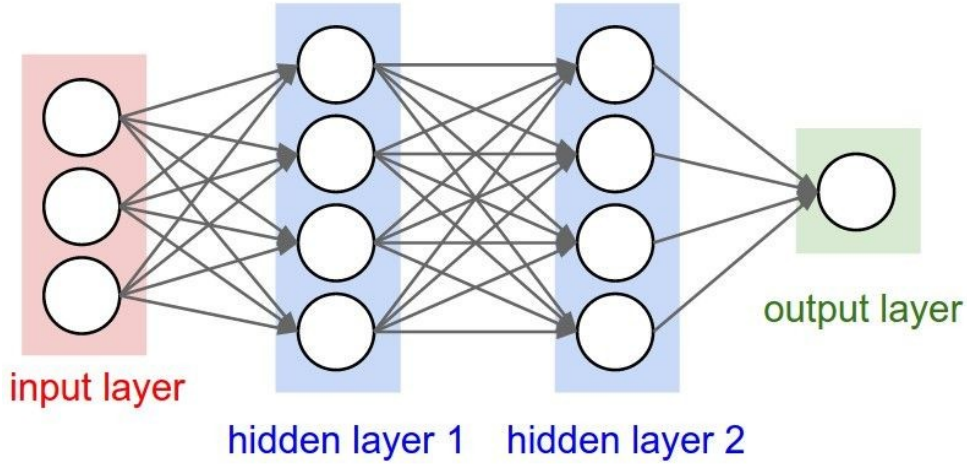
# Patterns in backward flow

**add** gate: gradient distributor
**max** gate: gradient router
**mul** gate: gradient… “switcher”?

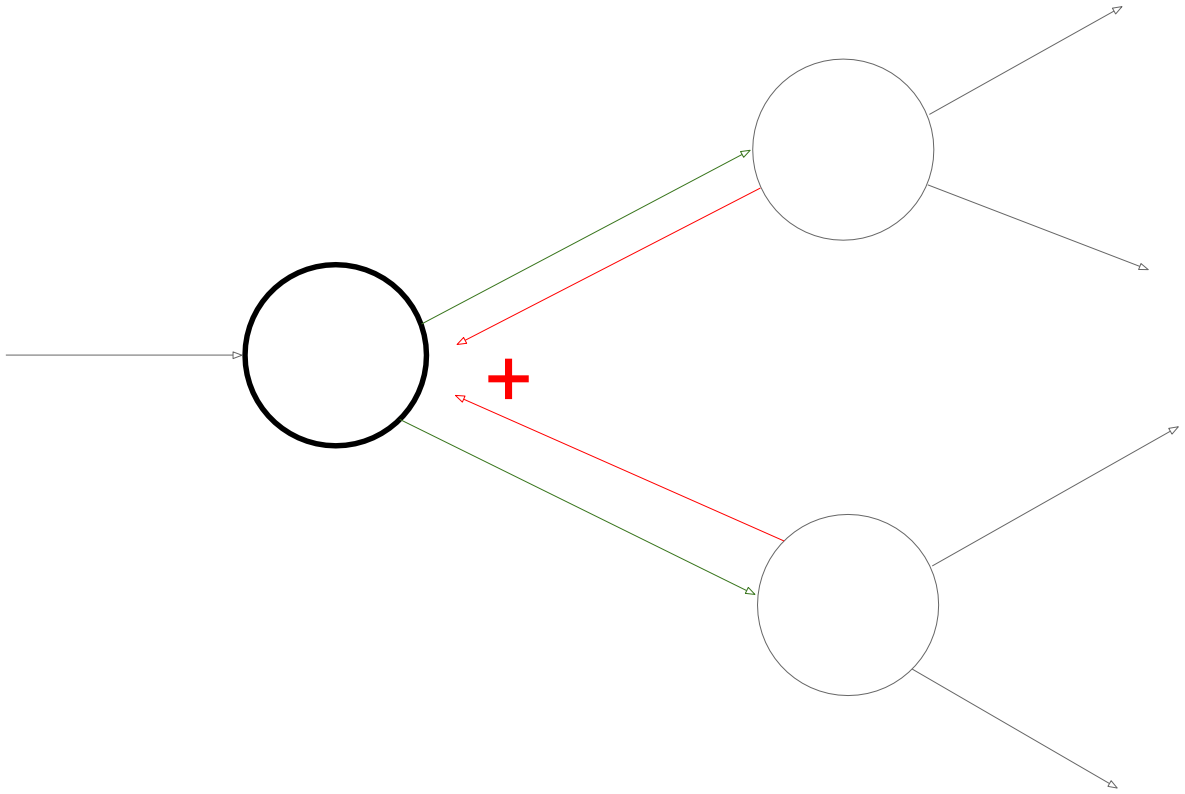


# How do we handle gradients for nodes with multiple output connections?



# Gradients add when there are multiple output connections



# How do we backdrop through a RELU?

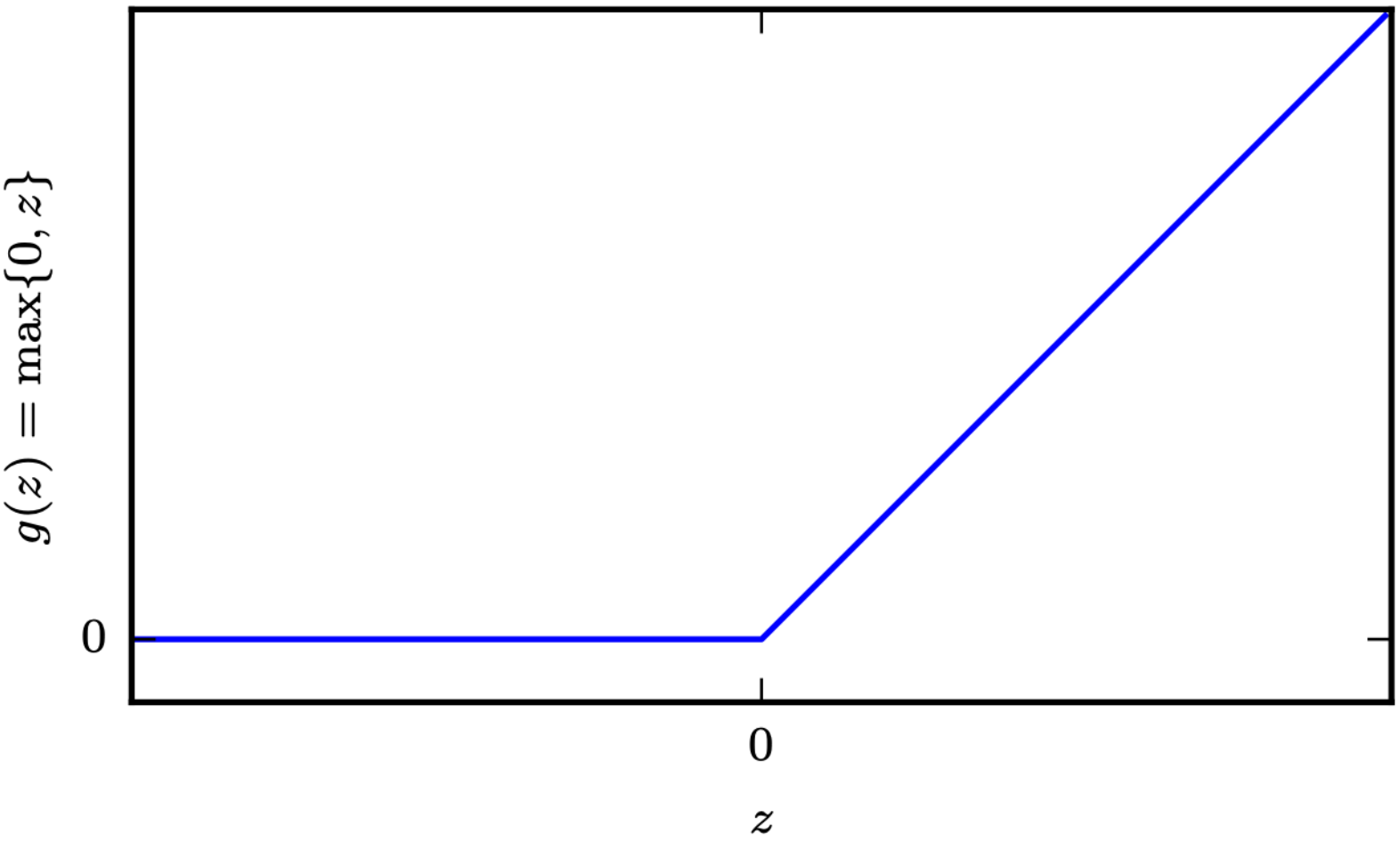



Figure 6.3

# Only propagate gradient if input is non-zero

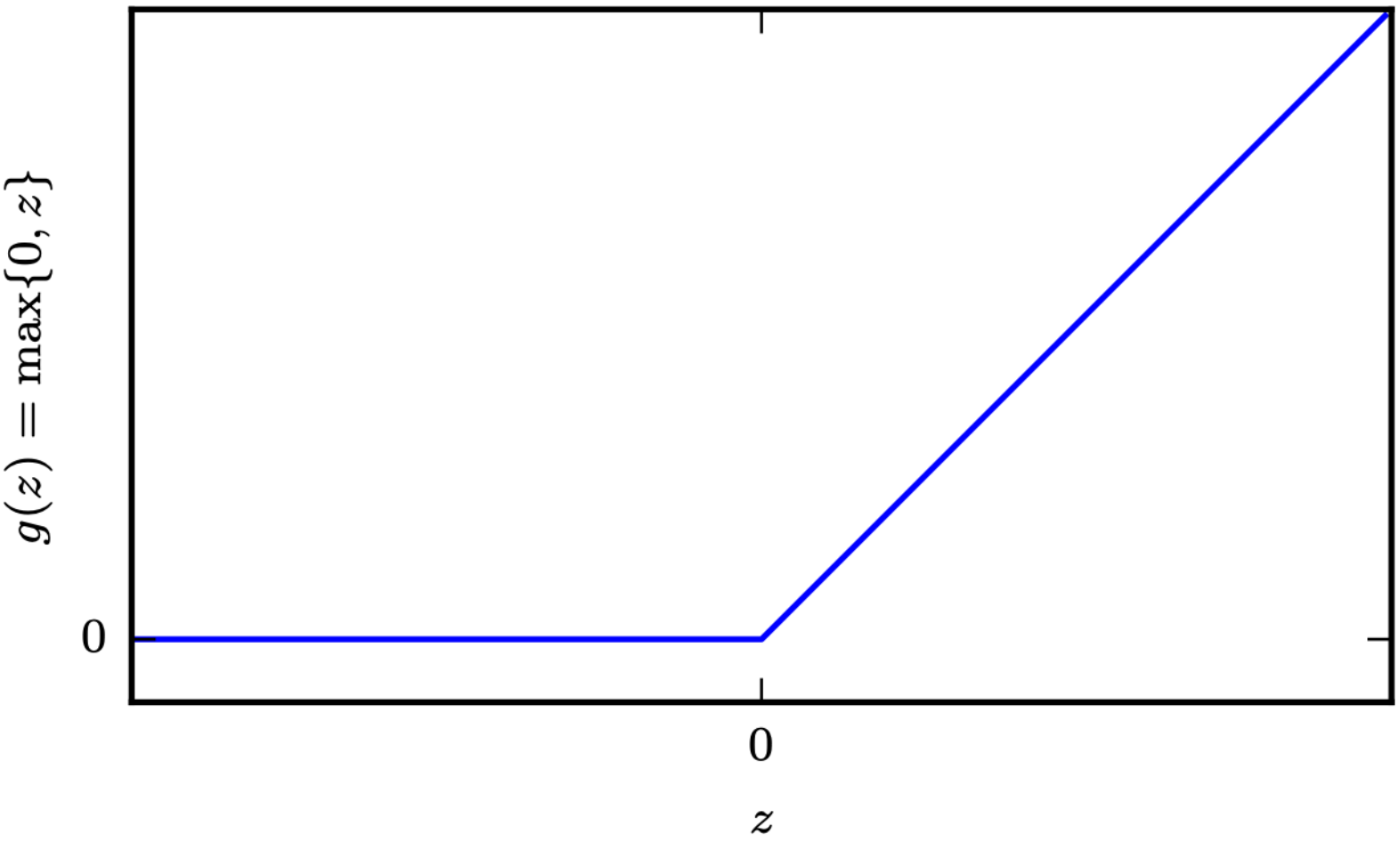



Figure 6.3

# Evaluation metrics

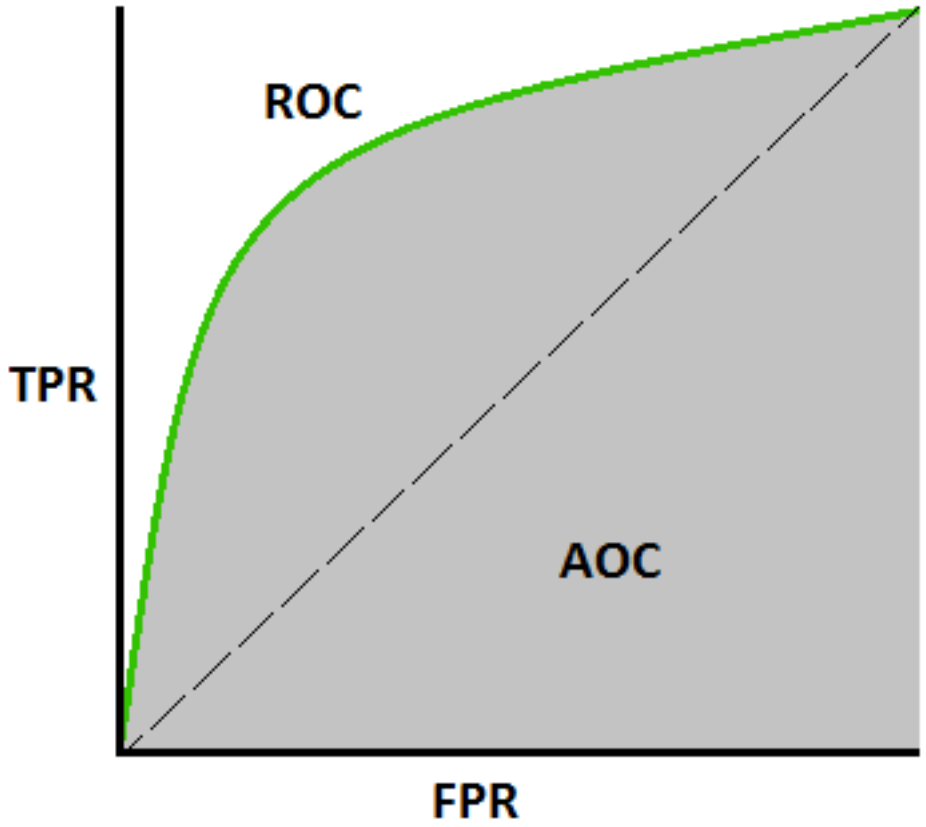
| | | True condition | | | | |
|---|---|---|---|---|---|---|
| | Total population | Condition positive | Condition negative | Prevalence = $\frac{\Sigma \text{ Condition positive}}{\Sigma \text{ Total population}}$ | Accuracy (ACC) = $\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$ | |
| **Predicted condition** | Predicted condition positive | **True positive**, Power | **False positive**, Type I error | Positive predictive value (PPV), Precision = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition positive}}$ | False discovery rate (FDR) = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Predicted condition positive}}$ | |
| | Predicted condition negative | **False negative**, Type II error | **True negative** | False omission rate (FOR) = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Predicted condition negative}}$ | Negative predictive value (NPV) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Predicted condition negative}}$ | |
| | | True positive rate (TPR), Recall, Sensitivity, probability of detection = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$ | False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Condition negative}}$ | Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$ | Diagnostic odds ratio (DOR) = $\frac{\text{LR+}}{\text{LR−}}$ | $F_1$ score = $\frac{1}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}{2}$ |
| | | False negative rate (FNR), Miss rate = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Condition positive}}$ | Specificity (SPC), Selectivity, True negative rate (TNR) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$ | Negative likelihood ratio (LR−) = $\frac{\text{FNR}}{\text{TNR}}$ | | |

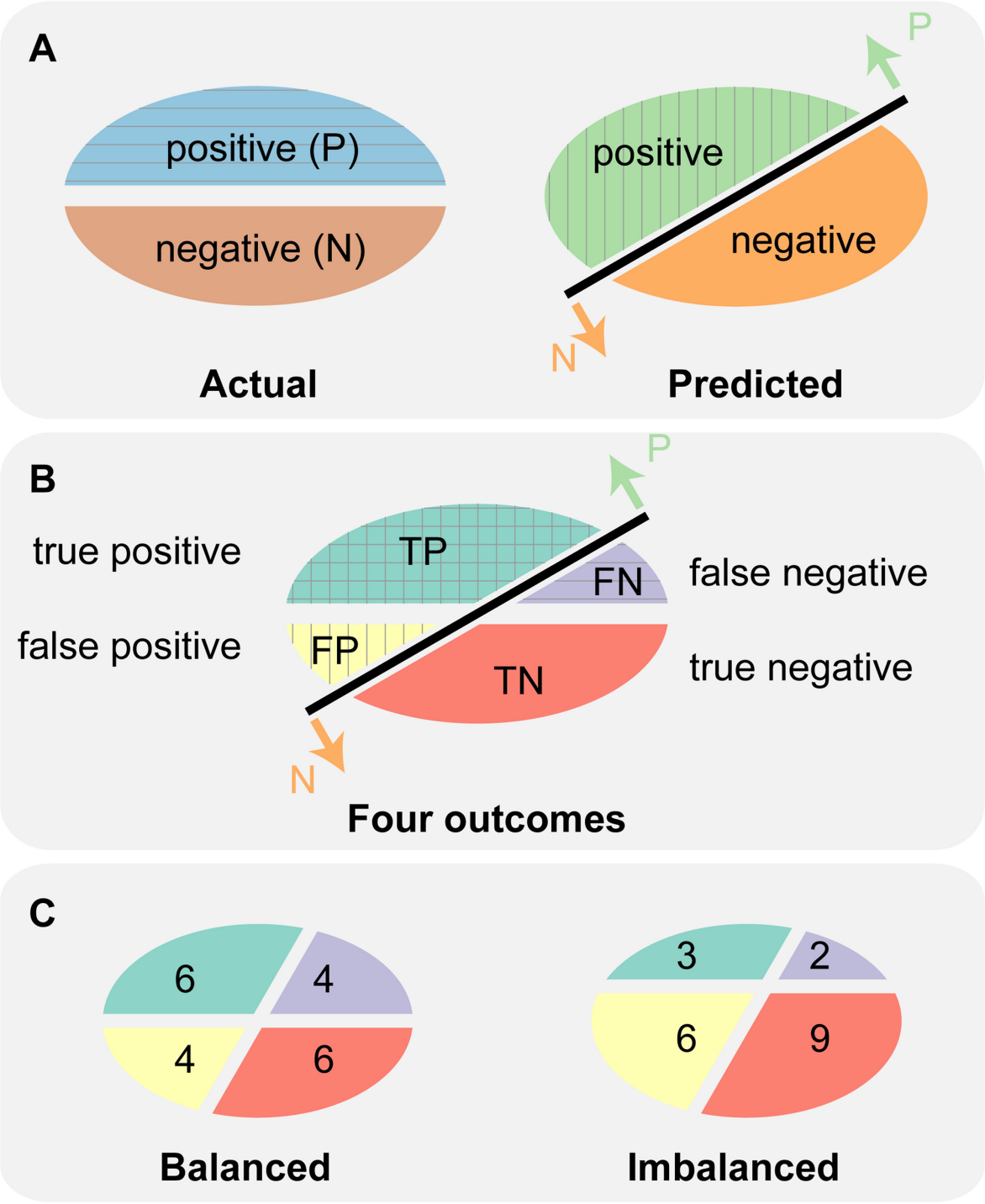
https://en.wikipedia.org/wiki/Sensitivity_and_specificity

# Example confusion matrix

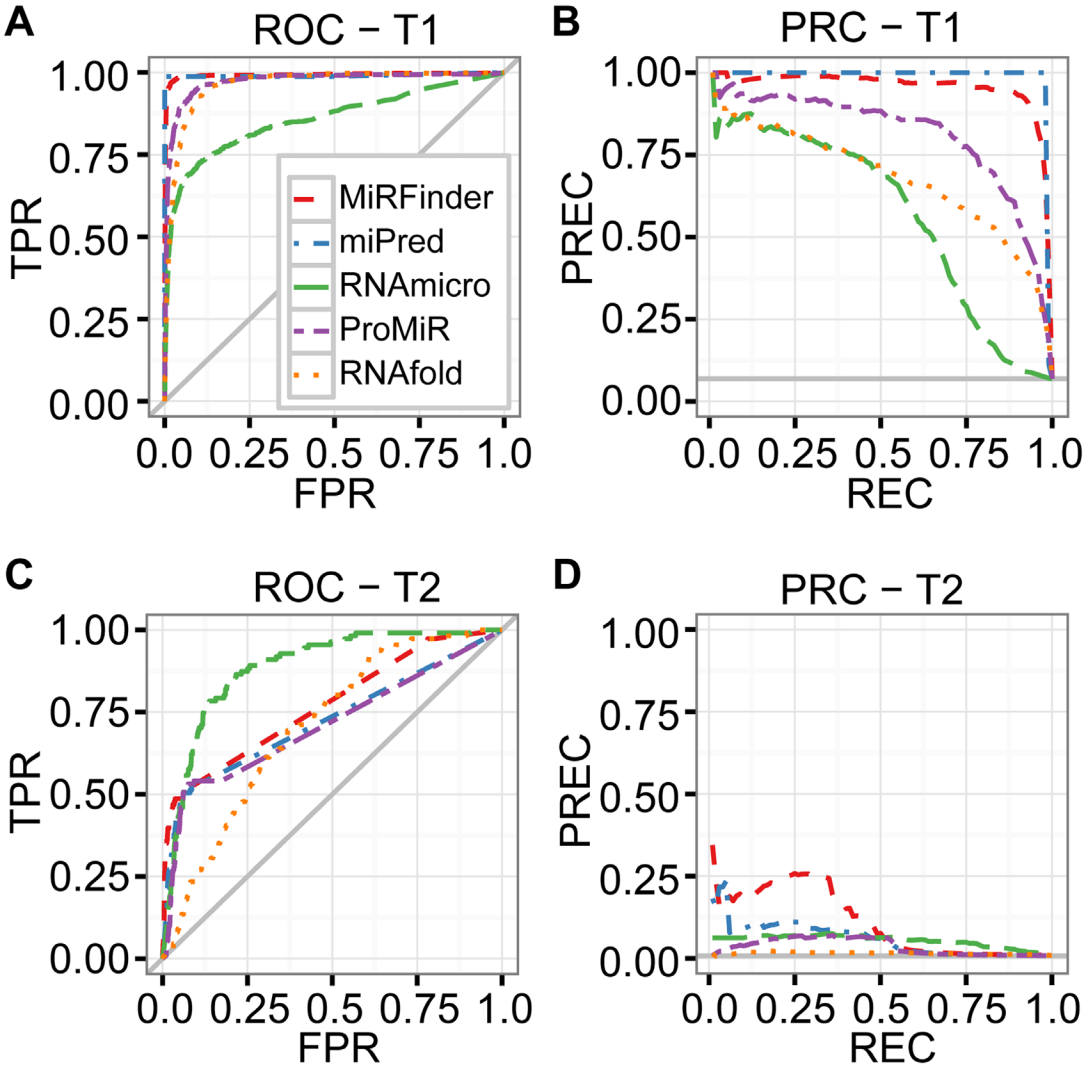
| n=165 | **Predicted: NO** | **Predicted: YES** | |
|---|---|---|---|
| **Actual: NO** | TN = 50 | FP = 10 | 60 |
| **Actual: YES** | FN = 5 | TP = 100 | 105 |
| | 55 | 110 | |

# A receiver operating characteristic (ROC) curve is made by changing the decision boundary




A
positive (P)
negative (N)
Actual
P
positive
negative
N
Predicted
B
true positive
false positive
TP
FN
FP
TN
P
N
false negative
true negative
Four outcomes
C
6
4
4
6
Balanced
3
2
6
9
Imbalanced

# ROC and Precision-Recall curves are both useful



# Training sets are best balanced

# Substantially unbalanced training sets have undesirable properties

- For example, 1000 Class 1 examples, 100 Class 2 examples
- Result can be overfit models that do not generalize well to new examples
- Performance metrics can deceive as they represent underlying class distribution

# How to deal with unbalanced training data

- Aquire more data
- Use different performance metrics (confusion matrix, precision-recall curves)
- Resample the data to be more balanced
- Use synthetic examples
- Try different methods (e.g. decision trees)
- Use methods specialized to anomaly or change detection