# Business Analytics & Machine Learning

Convex Optimization and Neural Networks

Prof. Dr. Martin Bichler

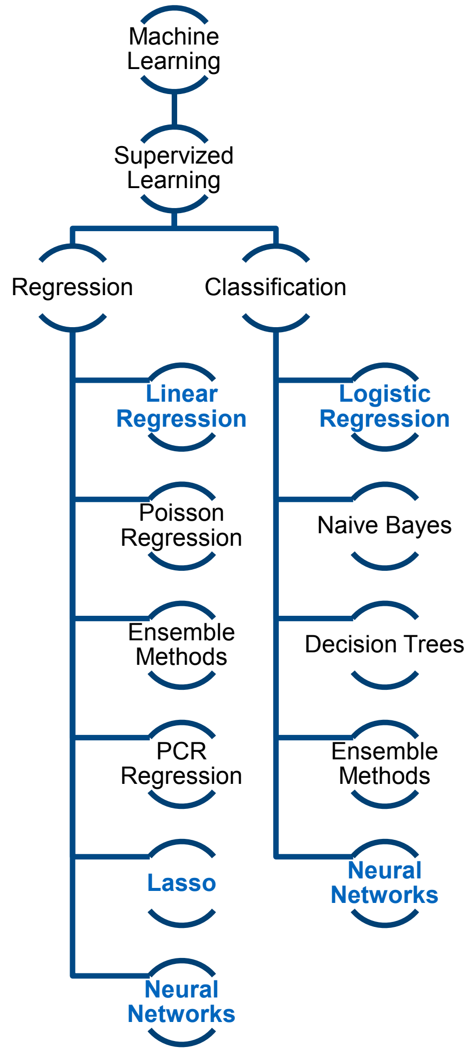
Department of Computer Science

School of Computation, Information, and Technology

Technical University of Munich

## Course Content

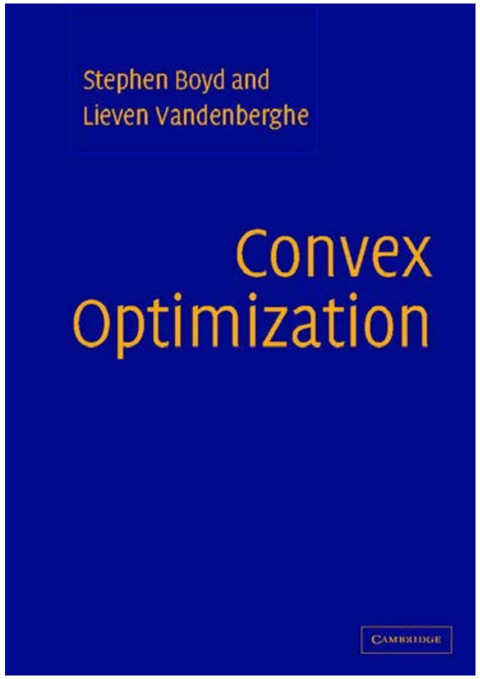
- Introduction
- Regression Analysis
- Regression Diagnostics
- Logistic and Poisson Regression
- Naive Bayes and Bayesian Networks
- Decision Tree Classifiers
- Data Preparation and Causal Inference
- Model Selection and Learning Theory
- Ensemble Methods and Clustering
- Dimensionality Reduction
- Association Rules and Recommenders
- **Convex Optimization**
- Neural Networks
- Reinforcement Learning



## Recommended Literature

- **Convex Optimization**
  - Stephen Boyd and Lieven Vandenberghe
  - discusses the field in great depth
  - https://web.stanford.edu/~boyd/cvxbook/

- **Gradient Descent**
  short description of this algorithm only
  https://en.wikipedia.org/wiki/Gradient_descent



## Agenda for Today

- Convex optimization basics
- Gradient descent
- Linear and logistic regression recap
- Introduction to neural networks
- Empirical risk minimization

# Convex Optimization in Machine Learning

Many machine learning problems can be cast as minimizing a loss function:

- the OLS estimator for the linear regression (already invented by Gauss!)
- the ML estimator of a logistic regression
- the use of regularization in lasso and the ridge regression (today)
- backpropagation in neural networks (next class)
- etc.

➔ **Convex optimization** plays a crucial role, because it is often easier to "convexify" a problem (make it convex optimization friendly), rather than to use non-convex optimization.

*In this class, we* ***revisit*** *material from prior classes in our course through the lense of convex optimization. This should aid your understanding of previous classes and machine learning as a combination of statistics and optimization.*

## Multidimensional Optimization Problems

**Goal**, given any function $f: \mathbb{R}^d \to \mathbb{R}$, find

$$x^* = \underset{x \in \mathbb{R}^d}{\operatorname{argmin}} f(x)$$

If $f: \mathbb{R}^d \to \mathbb{R}$ is differentiable, then its **gradient at position $x$** is defined as

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_d} \end{pmatrix}$$

The vector $\nabla f(x)$ of partial derivatives $\frac{\partial f(x)}{\partial x_i}$ at $x$ is the **gradient** of $f$, and points to the direction of steepest ascent.

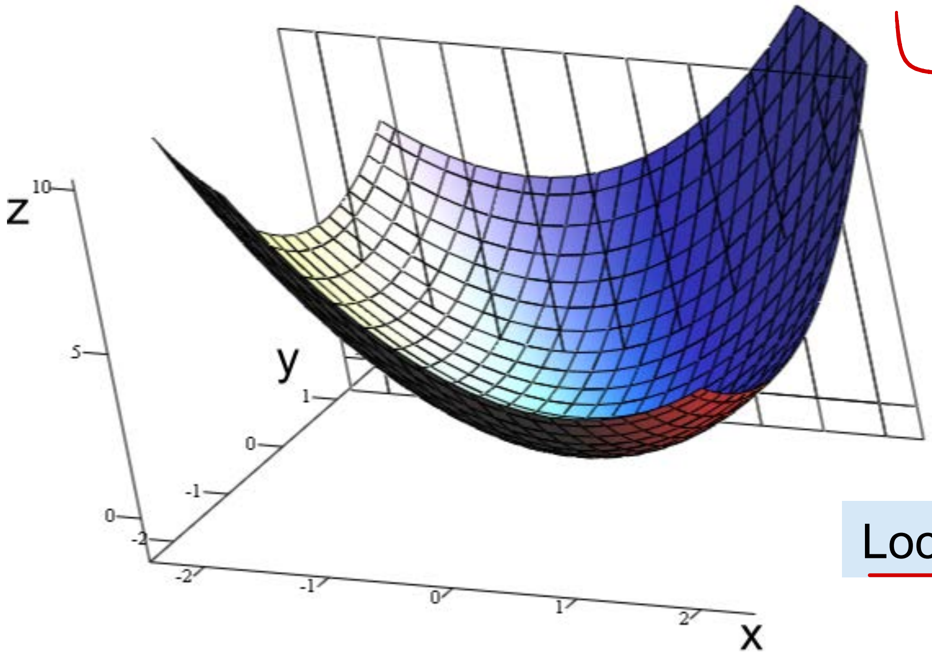
Thus $-\nabla f(x)$ is a **descent direction**, i.e. (at least) for small $\alpha > 0$: $f\big(x - \alpha \nabla f(x)\big) \leq f(x)$.

# Convexity in Multiple Dimensions

A function $f\colon \mathbb{R}^d \to \mathbb{R}$ is called **convex** iff

$$\forall \lambda \in (0,1), x, y \in \mathbb{R}^d:$$
$$\boxed{f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)}$$

$$\hookrightarrow f\left(\sum_{i=1}^n \lambda_i x_i\right) \leq \sum_{i=1}^n \lambda_i f(x_i)$$



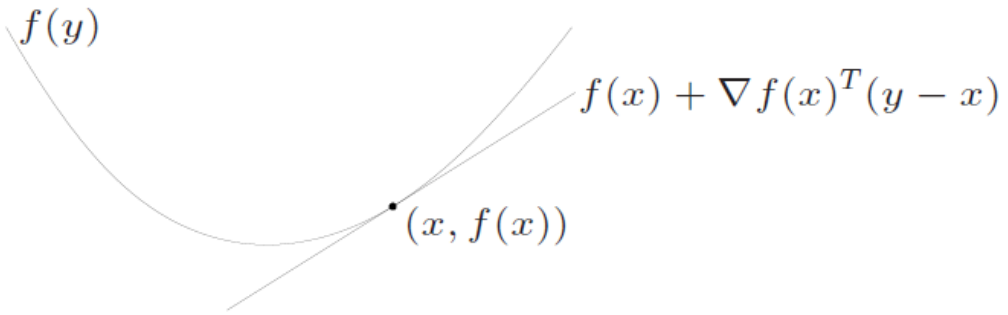
Local optima are globally optimal!

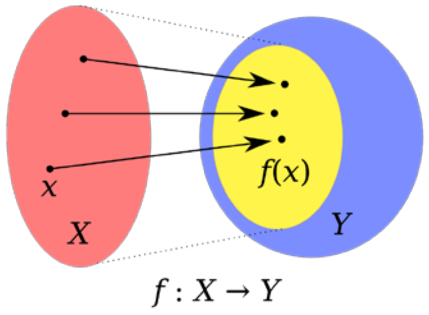
## Differentiable Convex Functions

**Definition of convexity via the gradient**

A differentiable function $f$ is convex, iff dom $f$ is convex and

$$f(y) - f(x) \geq \nabla f(x)^T(y-x) \; \forall x, y \in dom\ f$$
$$\equiv f(x) - f(y) \leq \nabla f(x)^T(x-y) \; \forall x, y \in dom\ f$$





The red oval is dom $f$

The affine function is a tangent of $f$ at $(x, f(x))$.

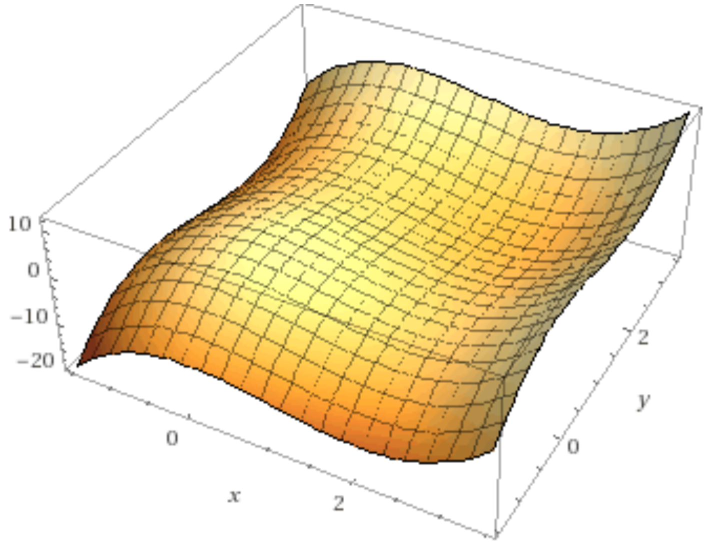
**Definition via the Hessian matrix:**

A twice differentiable function $f$ is convex, iff dom $f$ is convex and the Hessian matrix is positive semidefinite: $\nabla^2 f(x) \geq 0 \; \forall x \in dom\ f$

## Example Gradient

$f(x, y) = x^3 - 3x^2 + y^3 - 3y^2$

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} 3x^2 - 6x \\ 3y^2 - 6y \end{pmatrix}$$



(Source: Wolfram Alpha)

Stationary points are at $3x^2 - 6x = 0$ and $3y^2 - 6y = 0$

Four stationary points: $(0,0), (2,0), (0,2), (2,2)$

The function is neither (globally) convex nor concave. Optimization algorithms might get stuck in local optima!

## Extreme Values

If $X \subseteq \mathbb{R}^d$ is open and $f: X \to \mathbb{R}$ is a twice continuously differentiable function of multiple variables $x = (x_1, \dots, x_d)$, then

- $\nabla f(x') = 0$ is a necessary condition for $x'$ to be a local minimum or maximum.
- The symmetric matrix $H(x) = \left(\frac{\partial f^2(x)}{\partial x_i \partial x_j}\right)_{\mathrm{i,j}}$ **(Hessian)** exists.
- $\nabla f(x') = 0$ and - $H(x')$ is positive definite (or $H(x')$ negative definite) is a sufficient condition that $x'$ is a local maximum.

- $H(x')$ negative definite → Eigenvalues < 0 → maximum
- $H(x')$ positive definite → Eigenvalues > 0 → minimum
- $H(x')$ indefinite → Eigenvalues < 0 and > 0 → saddle point
- $H(x')$ positive <u>semi</u>definite → Eigenvalues $\geq 0$ → inconclusive

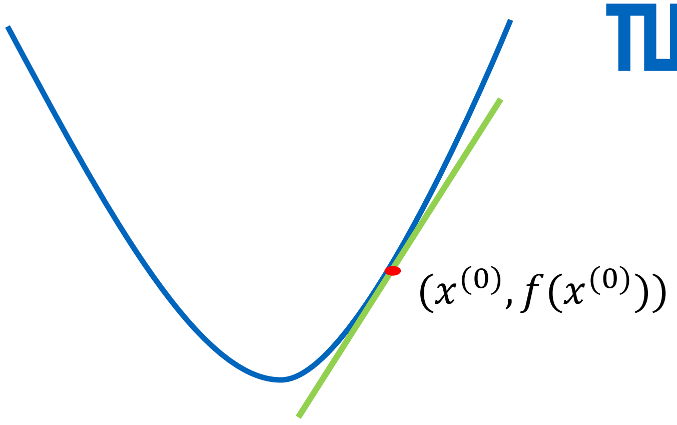
## Gradient Descent

Input:
Convex, continuously differentiable function $f\colon \mathbb{R}^d \to \mathbb{R}$,
feasible start point $x^{(1)} \in \mathbb{R}^d$, and parameter $\varepsilon > 0$

```
$k = 1$
While($\|\nabla f(x^{(k)})\| \geq \varepsilon$){
  • Choose step size $\alpha > 0$,
    e.g. compute $\alpha^* > 0$, to maximize $f\left(x^{(k)} + \alpha \nabla f\left(x^{(k)}\right)\right)$, (aka. “line search”)
  • Set $x^{(k+1)} = x^{(k)} - \alpha^* \nabla f\left(x^{(k)}\right)$
  • $k++$
}
```

Alternatively, use $x^{(k+1)} = x^{(k)} + \alpha^* \nabla f\left(x^{(k)}\right)$ for gradient **ascent** in maximization.

# Deriving Gradient Descent



Unconstrained minimization: $\min f(x), s.t.\ x \in X$

Derivation via first-order Taylor approximation (i.e., a linear function)

$$f(x) \approx f\left(x^{(0)}\right) + \left\langle \nabla f\left(x^{(0)}\right), x - x^{(0)} \right\rangle$$

This way, we would follow the gradient for very long.

The approximation is only close in the vicinity of $x^{(0)}$.

Therefore, we add a penalty term using the following formula starting with $k = 0$.

$$x^{(k+1)} = \text{argmin}_x \underbrace{f(x^{(k)})} + \underbrace{\left\langle \nabla f\left(x^{(k)}\right), x - x^{(k)} \right\rangle} + \frac{1}{2\alpha} \left\| x - x^{(k)} \right\|^2$$

The first-order condition w.r.t. $x$ is:

$$0 + \nabla f\left(x^{(k)}\right) + \frac{1}{\alpha}\left(x - x^{(k)}\right) = 0$$

$$\boxed{x^{(k+1)} = x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)}$$

$\rightarrow$ move in the direction against the gradient

## Gradient Descent

$$f(x_1, x_2) = -2(x_1 + x_2) + x_1^2 + 2x_2^2$$

$$\nabla f(x) = \begin{pmatrix} -2 + 2x_1 \\ -2 + 4x_2 \end{pmatrix}$$

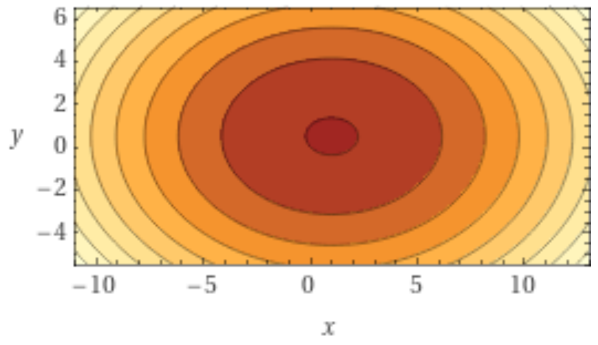
```
$k = 1$
While($\|\nabla f(x^{(k)})\| \geq \varepsilon$){
    • Compute $\alpha^* > 0$, to maximize $f\left(x^{(k)} - \alpha \nabla f(x^{(k)})\right)$.
    • Set $x^{(k+1)} = x^{(k)} - \alpha^* \nabla f(x^{(k)})$
    • $k++$
}
```

Start at $x^{(1)} = (0,0)$, $f(x^{(1)}) = 0$, $\nabla f(x^{(1)}) = \begin{pmatrix} -2 \\ -2 \end{pmatrix}$

Maximize $f(0 + 2\alpha, 0 + 2\alpha) = -8\alpha + 12\alpha^2$

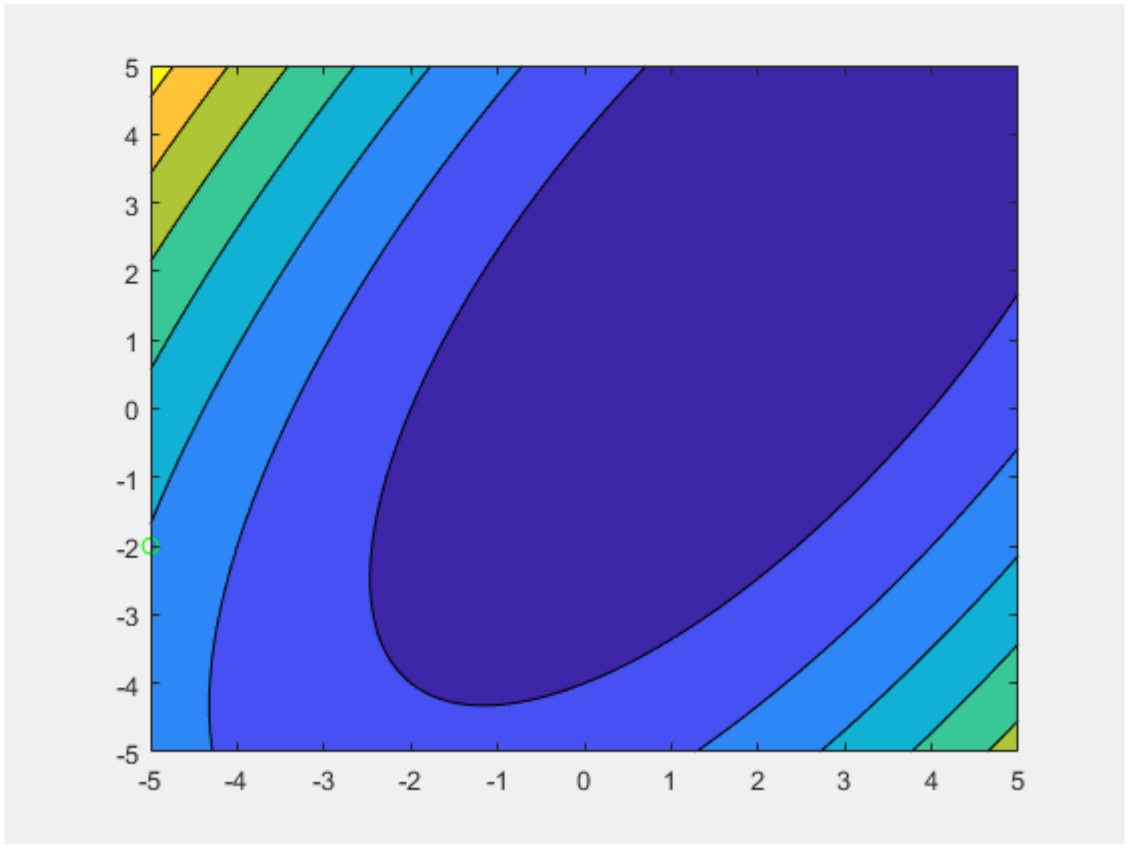
- Step size $\frac{1}{3}$

→ $x^{(2)} = \left(\frac{2}{3}, \frac{2}{3}\right)$, $f(x^{(2)}) = -\frac{4}{3}$, $\nabla f(x^{(2)}) = \begin{pmatrix} -\frac{2}{3} \\ \frac{2}{3} \end{pmatrix}$



This is the direction of steepest descent.
Line search is too expensive in many applications and replaced by fixed $\alpha$ or $\alpha_t$.

# Gradient Descent Animation



$$f(x_1, x_2) = 0.1x_1^4 + (x_1 - 2)^2 - (x_1 - 2)(x_2 - 2) + 0.5(x_2 - 2)^2$$

with diminishing step size rule

# Momentum Gradient Descent

Vanilla gradient descent:

1. Choose a starting point $x^{(0)}$
2. Choose the maximum number of iterations $T$
3. Choose a learning rate $\alpha \in [a, b]$
4. Repeat $x^{(k+1)} = x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)$

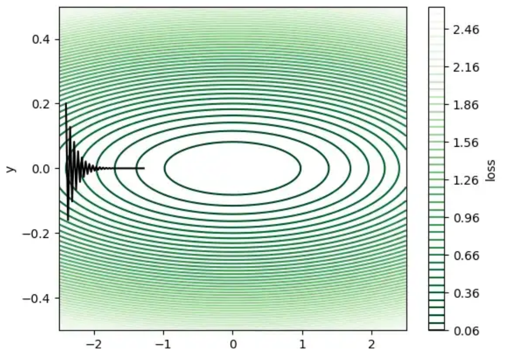
Gradient descent keeps no memory of the past.
This can lead to inefficiencies.

Use *momentum*:

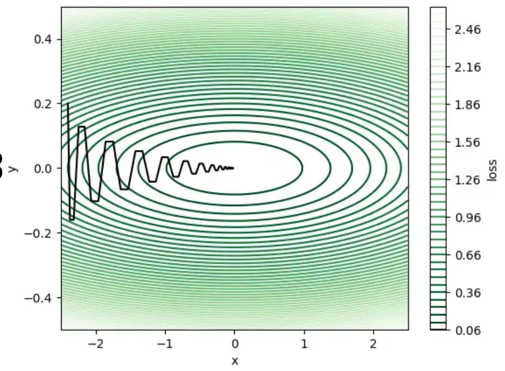
$$x^{(k+1)} = x^{(k)} - \alpha v^{(k)}$$

$$v^{(k)} = \underline{\beta}\underbrace{v^{(k-1)}}_{\text{memory of past gradients}} + (\underline{1-\beta})\nabla f\left(x^{(k)}\right)$$

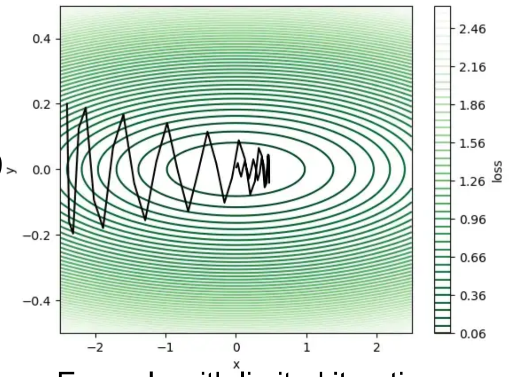
$\beta = 0$



$\beta = 0.8$



$\beta = 0.9$



Example with limited iterations.

# Python Code

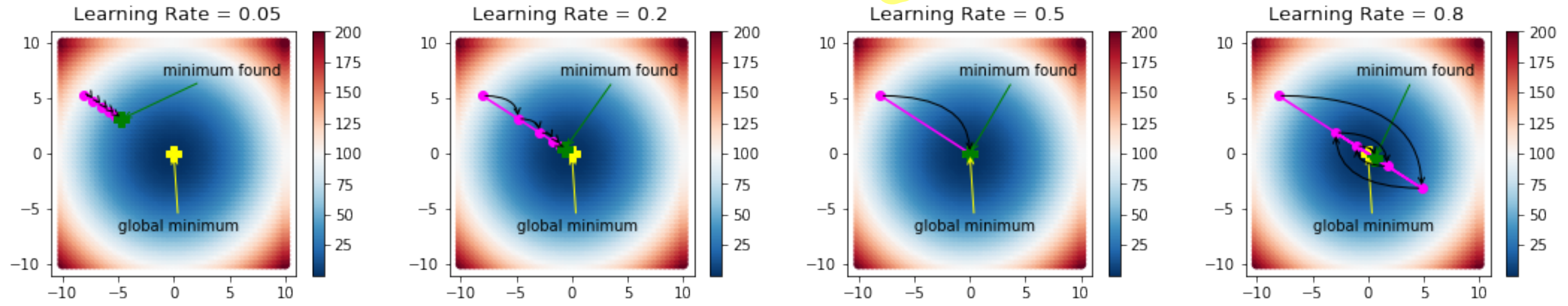
```
def gradient_descent(max_iterations,threshold,x_init,obj_func,grad_func,learning_rate=0.05, momentum=0.8):
    x = x_init
    x_history = x
    f_history = obj_func(x,extra_param)
    delta_x = np.zeros(x.shape)
    i = 0
    diff = 1.0e10
    while  i<max_iterations and diff>threshold:
        delta_x = -learning_rate*grad_func(x,extra_param) + momentum*delta_x
        x = x+delta_x
        # store history of x and f
        x_history = np.vstack((x_history,x))
        f_history = np.vstack((f_history,obj_func(x,extra_param)))
        # update iteration number and diff between successive values
        i+=1
        diff = np.absolute(f_history[-1]-f_history[-2])
     return x_history,f_history

def f(x):
    return np.sum(x*x)

def grad(x):
    return 2*x
...

gradient_descent(5,-1,x_init, f,grad,eta,alpha)
```

# Different Levels of Momentum

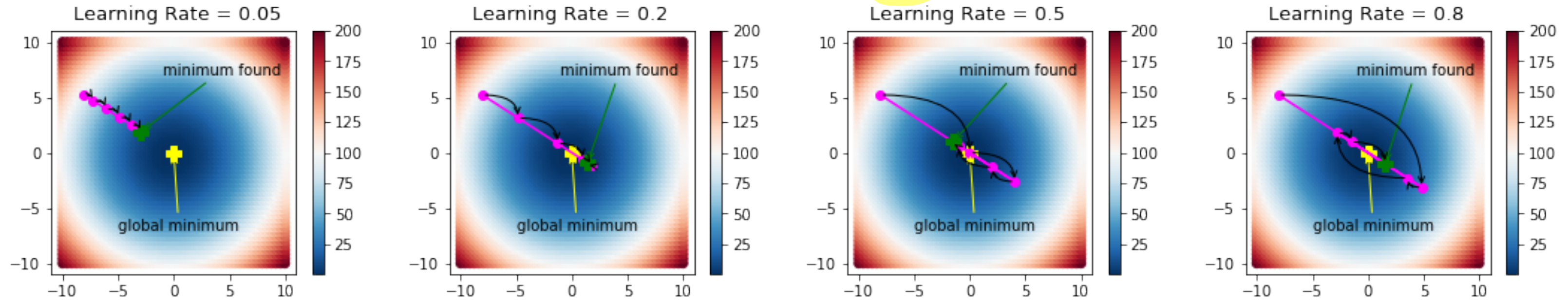
$$f(x_1, x_2) = x_1^2 + x_2^2$$
$$\nabla f(x) = \begin{pmatrix} 2x_1 \\ 2x_2 \end{pmatrix}$$

small steps

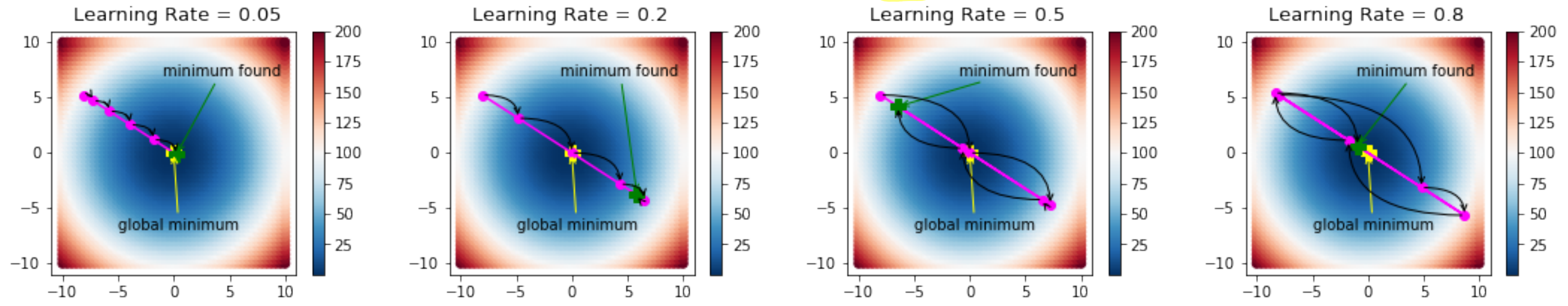
momentum = 0



momentum = 0.5



momentum = 0.9



## Versions of Gradient Descent

- **Vanilla** Gradient Descent

$$\boxed{x^{(k+1)} = x^{(k)} - \alpha \nabla f\left(x^{(k)}\right)}$$

- **Adaptive** Gradient descent uses different step sizes for different components of the gradient

- **Momentum** Gradient Descent

$$x^{(k+1)} = x^{(k)} - \alpha v^{(k)}$$

$$v^{(k)} = \beta v^{(k-1)} + (1-\beta) \nabla f\left(x^{(k)}\right)$$

$$= (1-\beta)\nabla f\left(x^{(k)}\right) + \beta(1-\beta)\nabla f\left(x^{(k-1)}\right) + \beta^2(1-\beta)v^{(k-2)} + \cdots$$

keeps an exponential average of the gradient

if previous gradients are in the same direction, it accelerates. If not, it delays

- **Nesterov's Accelerated** Gradient Descent converges faster for strongly convex functions

$$x^{(k+1)} = x^{(k)} - \alpha v^{(k)}$$

$$v^{(k)} = \beta v^{(k-1)} + (1-\beta) \nabla f\left(x^{(k)} - \beta v^{(k-1)}\right)$$

the algorithm looks ahead, $x^{(k)} - \beta v^{(k-1)}$, before computing the gradient

## Gradient Descent on Data Sets

In **batch gradient descent**, the gradient is determined based on the average of the gradients for all data points in a data set. An iteration or gradient update considers the entire data set.

**Mini-batch Gradient Descent:**

Instead of using just one data point, a small random subset (mini-batch) of the data is used to compute the gradient. Useful when computing the gradient over the entire dataset in each iteration is too expensive.

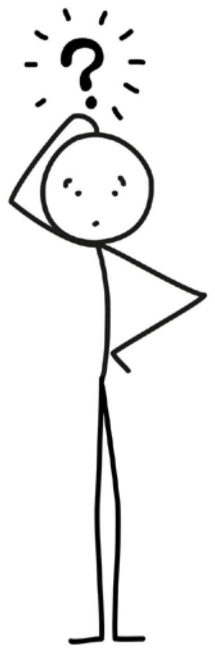
$$x^{(k+1)} = x^{(k)} - \alpha \frac{1}{|Mini - batch|} \sum_{i \in Mini\ batch} \nabla f_i\left(x^{(k)}\right)$$

**Stochastic Gradient Descent (SGD)** is a variant of gradient descent that is commonly used in the context of large-scale optimization problems, particularly in machine learning and deep learning.

$$x^{(k+1)} = x^{(k)} - \alpha \nabla f_i\left(x^{(k)}\right)$$

The difference to vanilla gradient descent is that the gradient is taken for a randomly chosen data point $i$. The parameter vector $x^{(k+1)}$ is then chosen for that specific tata point‘s gradient.

## How did we compute the parameters of a linear regression?



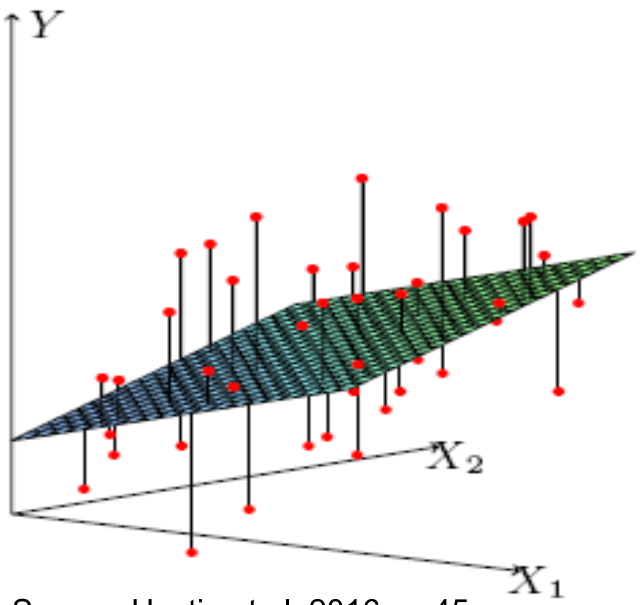
# The Multiple Linear Regression Model

A $p$-variable regression model can be expressed as a series of equations.

Equations condensed into a matrix form, give the general linear model.

β coefficients are known as partial regression coefficients:
$X_1, X_2$, for example,
$X_1$='years of experience'
$X_2$='age'
Y='salary'

Estimated equation:
$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2 = \mathbf{X}\hat{\beta}$



Source: Hastie et al. 2016, p. 45

## Finding the Parameters of a Linear Regression

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & x_{12} & \dots & x_{1p} \\ 1 & x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}$$

| $y$ | $\boldsymbol{X}$ | β | + ε |
|---|---|---|---|
| $(n \times 1)$ | $(n \times (p+1))$ | $((p+1) \times 1)$ | $(n \times 1)$ |

# OLS Estimation

Sample-based counter part to population regression model:

$$y = \mathbf{X}\beta + \varepsilon$$
$$y = \mathbf{X}\hat{\beta} + e$$

OLS requires choosing values of the estimated coefficients, such that Residual Sum of Squares (RSS) is as small as possible for the sample

$$RSS = \underline{e^T e = (y - \mathbf{X}\hat{\beta})^T (y - \mathbf{X}\hat{\beta})}$$

This can also be called a **loss function**.
Need to differentiate with respect to the unknown coefficients.

# Least Squares Estimation

$\mathbf{X}$ is $n \times (p+1)$, $y$ is the vector of outputs
RSS(β) = $(y - \mathbf{X}\beta)^T (y - \mathbf{X}\beta)$

If $\mathbf{X}$ is full rank, then $\mathbf{X}^T\mathbf{X}$ is positive definite:

$$\Rightarrow RSS = (y^T y - 2\beta^T \mathbf{X}^T \mathrm{y} + \beta^T \mathbf{X}^T \mathbf{X}\beta)$$

$$\frac{\partial RSS}{\partial \beta} = -2\mathbf{X}^T \mathrm{y} + 2\mathbf{X}^T\mathbf{X}\beta = 0$$ First-order condition

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

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

$$\frac{\partial^2 RSS}{\partial \beta^T \partial \beta} = 2\mathbf{X}^T\mathbf{X}$$

Positive Semi-Definiteness:
$v^T H v \geq 0$, *with* $v \geq 0$

$2v^T X^T X v$
$= 2(Xv)^T(Xv)$
$= 2||Xv||^2 >= 0$

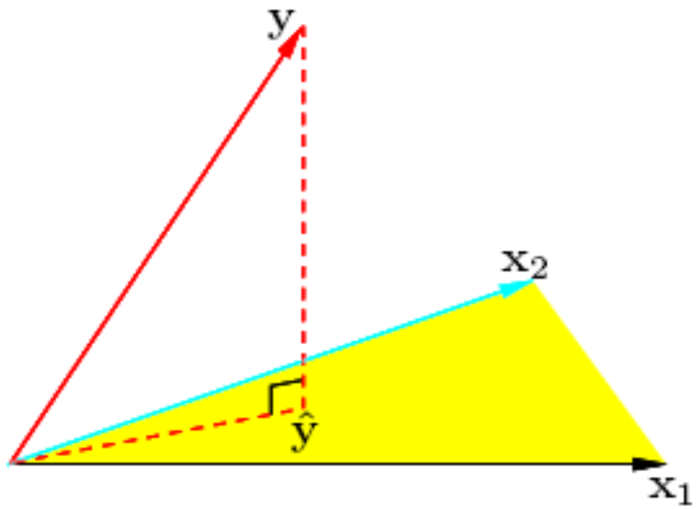
For a matrix $\boldsymbol{A}$, the matrix $\boldsymbol{A}^T\boldsymbol{A}$ is always positive semidefinite. This means that the covariance matrix $\mathbf{X}^T\mathbf{X}$ is positive semi-definite and invertible.

# Quadratic Optimization in OLS Estimation

- least square estimates in $\mathbb{R}^n$
- We minimize the squared distance between observed and estimated values: RSS(β)=$||y - \mathbf{X}\beta||^2$, s.t. residual vector $y - \hat{y}$ is orthogonal to this subspace $\mathbf{X}$.
- We found an analytical solution: $\hat{y} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y$

**Definition (Projection):**
The set $X \subset \mathbb{R}^n$ is non-empty, closed and convex. For a fixed $y \in \mathbb{R}^n$ we search a point $\hat{y} \in X$, with the smallest distance to $y$ (w.r.t. the Euclidean norm), i.e. we solve the minimization problem

$$P_X(y) = \min_{\hat{y} \in X} ||y - \hat{y}||^2$$



Source: Hastie et al. 2016, p. 46

# OLS Estimation via Gradient Descent

Instead of computing the analytical solution, we can use gradient descent to minimize the sum of squared residuals.

$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$

We use the mean squared error as the **loss function**.

$$L(\beta) = \frac{1}{n}\sum_i e_i^2 = \frac{1}{n}\sum_i (y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_i))^2$$

Partial derivatives of the loss function w.r.t. $\hat{\beta}_1$ and $\hat{\beta}_0$ using the chain rule:

$$\frac{\partial L}{\partial \hat{\beta}_1} = \frac{1}{n}\sum_i 2(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)(-x_i) = \frac{-2}{n}\sum_i x_i(y_i - \hat{y}_i)$$

$$\frac{\partial L}{\partial \hat{\beta}_0} = \frac{-2}{n}\sum_i (y_i - \hat{y}_i)$$

# OLS Estimation via Gradient Descent

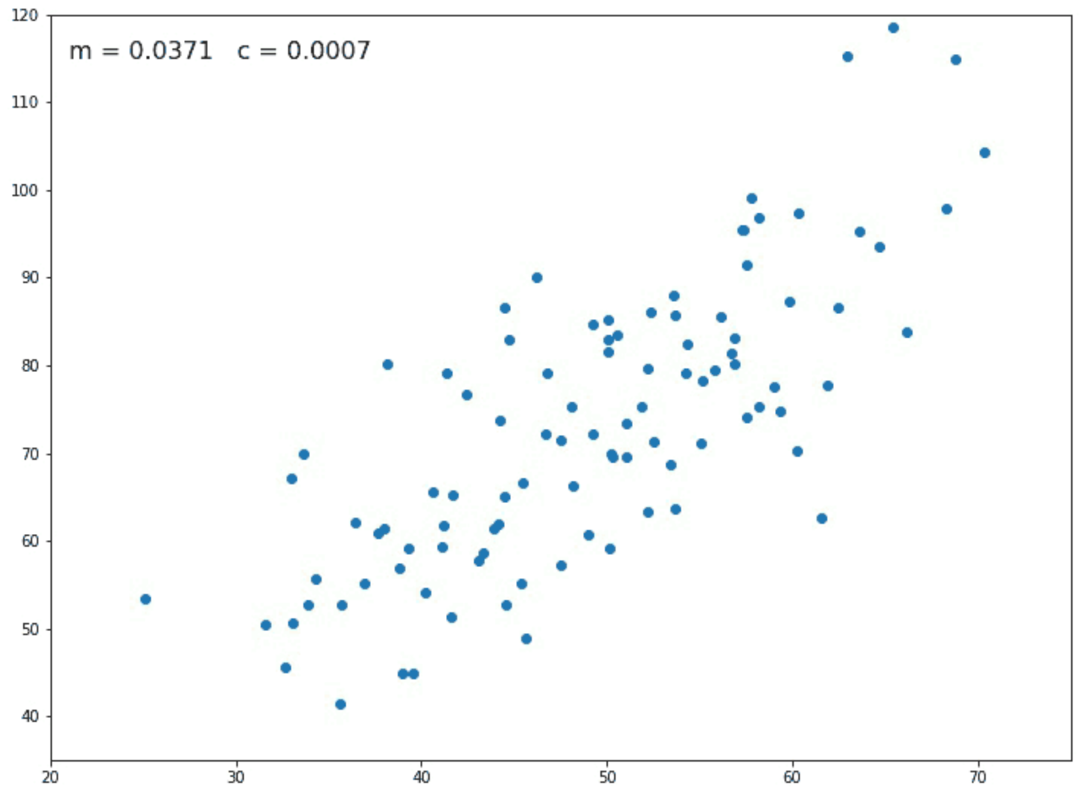
Pseudo code:

```
Start with $\hat{\beta}_0 = \hat{\beta}_1 = 0$
Repeat until the objective of the loss function is very small
        $\boxed{\hat{\beta}_1 := \hat{\beta}_1 - \alpha \frac{\partial L}{\partial \hat{\beta}_1}; \quad \hat{\beta}_0 := \hat{\beta}_0 - \alpha \frac{\partial L}{\partial \hat{\beta}_0}}$
```

Python code snippet:

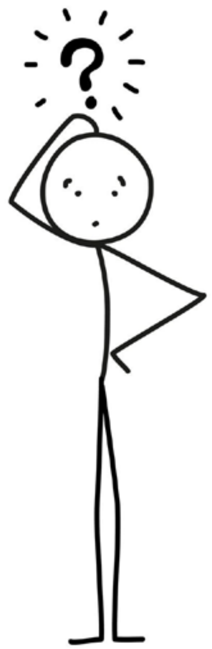
```
# Initialization of variables
b = 0
a = 0
L = 0.0001 # learning rate
epochs = 1000 # number of iterations
n = float(len(X)) # number of elements in X
# gradient descent
for i in range(epochs):
          Y_pred = b*X + a # The current predicted value of Y
          D_b = (-2/n) * sum(X * (Y - Y_pred)) # Derivative wrt b
          D_a = (-2/n) * sum(Y - Y_pred) # Derivative wrt a
          b = b - L * D_b # Update b
          a = a - L * D_a # Update a
print (b, a)
```

# Progress of Gradient Descent



Note that there is no need to compute the minimum via gradient descent for the linear regression, because we have a closed-form solution.

## How did we deal with high-dimensional regression problems and many (possibly irrelevant) variables?



## Remember: Subset Selection Methods

Last class: Principal Component Regression

Find the global optimal model, i.e. the best subset of independent variables:
Best subset regression (too computationally expensive)

Greedy search for the optimal model (practical):

- Forward stepwise selection
  - Begin with empty set, and sequentially adds predictors
- Backward stepwise selection
  - Begin with full model, and sequentially deletes predictors
- Stepwise selection: combination of forward and backward move

## Shrinkage (Regularization) Methods

The subset selection methods use OLS to fit a linear model that contains a subset of the predictors.

As an alternative, we can fit a model containing all *p* predictors using a technique that constrains or **regularizes** the coefficient estimates (i.e. **shrinks** the coefficient estimates towards zero).

It may not be immediately obvious why such a constraint should improve the fit, but it turns out that **shrinking** the coefficient estimates can significantly reduce their variance.

Gradient descent can be used to find the parameters of regularized OLS estimators.

# Gauss-Markov Theorem

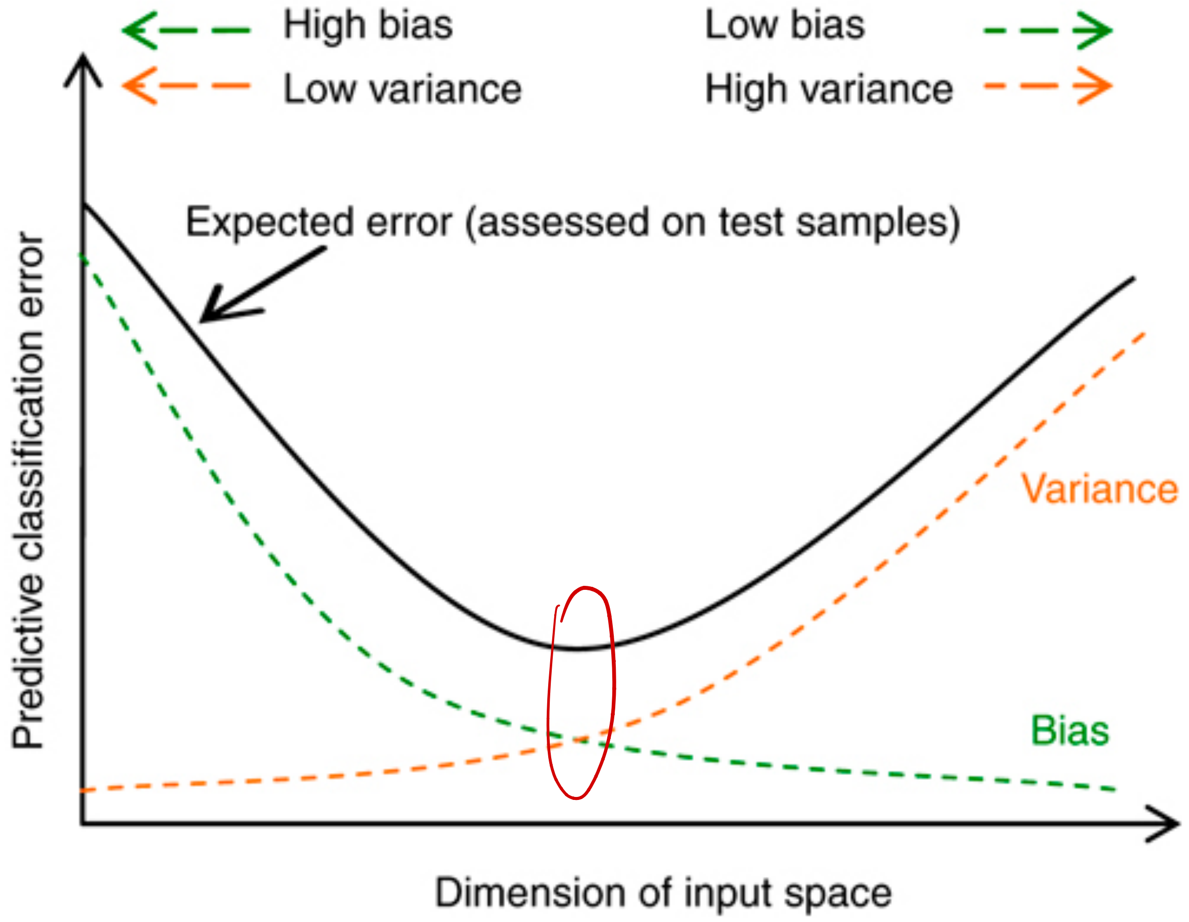
The Gauss-Markov theorem states that in a linear regression model in which the errors

- have expectation zero and
- are uncorrelated and
- have equal variances,

the **best linear unbiased estimator** (BLUE) of the coefficients is given by the ordinary least squares (OLS) estimator.

- „Unbiased“ means $E\left(\hat{\beta}_j\right) = \beta_j$
- „Best“ means giving the **lowest variance** of the estimate as compared to other linear unbiased estimators.
  – Restriction to unbiased (linear) estimation is **not always what you want**.

## Bias-Variance Tradeoff



## Regularization

If the linear model is correct for a given problem, then the OLS prediction is unbiased, and has the lowest variance among all linear unbiased estimators.

But there can be (and often exist) biased estimators with smaller MSE.

Generally, by **regularizing** the estimator in some way, its variance will be reduced; if the corresponding increase in bias is small, this trade-off will be worthwhile.

Examples of regularization:

- subset selection (forward, backward, all subsets)
- **ridge regression**
- **the lasso**

## Ridge Regression

Ridge coefficient minimize a penalized RSS. The parameter $\lambda > 0$ penalizes $\beta_j$ proportional to its size $\beta_j^2$.

$$\hat{\beta}^{ridge} = \arg\min_{\beta}\{\sum_i (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}\beta_j^2\} \rightarrow l_2 \text{ penalty}$$

Or

$$\left[\begin{array}{l} \underset{\beta}{Minimize}\{\sum_i (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2\} \\ subject \text{ to} \sum_{j=1}^{p}\beta_j^2 \le s \end{array}\right.$$

$\rightarrow$ avoid $\beta_j$ from going too large

This is a biased estimator that for some value of $\lambda$>0 may have smaller mean squared error than the least squares estimator.

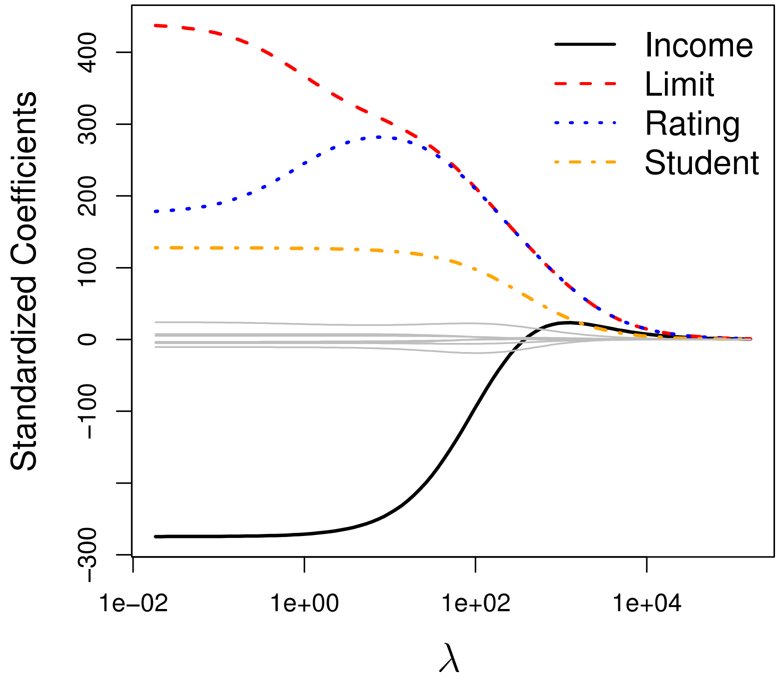
The sum of squared residuals and the lasso penalty are both convex, i.e. the lasso penalty function is also **convex**.

# Ridge Regression (cont.)

As $\lambda$ increases, the standardized ridge regression coefficients shrink to zero.

Thus, when $\lambda$ is extremely large, then all of the ridge coefficient estimates are basically zero; this corresponds to the **null model** that contains no predictors.

It is best to apply ridge regression after **standardizing the predictors**.



## Selecting the Tuning Parameter $\lambda$

As for subset selection, for ridge regression and lasso we require a method to determine which of the models under consideration is best; thus, we required a method selecting a value for the tuning parameter $\lambda$.

Select a grid of potential values; use **cross-validation** to estimate the error rate on test data (for each value of $\lambda$) and select the value that gives **the smallest error rate**.

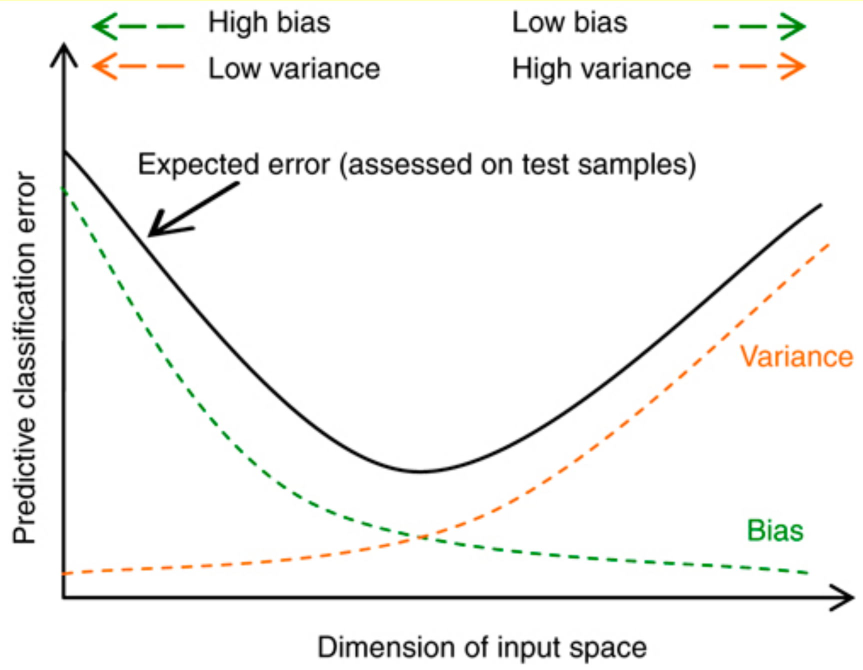
Finally, the model is re-fit using all of the variable observations and the selected value of the parameter $\lambda$.

## Ridge Regression

In general, the ridge regression estimates will be **more biased** than the OLS ones but have **lower variance**. This means, it does not fit the training data as well as the OLS estimator, but might do better on unseen test data.

Ridge regression will work best in situations where the OLS estimates have high variance.

Similar ideas can be applied to logistic regression.



## The Lasso

The lasso (least absolute shrinkage & selection operator) coefficients minimize the quantity:

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j| = \text{RSS} + \lambda\sum_{j=1}^{p}|\beta_j|$$

The key difference from ridge regression is that the lasso uses an $\ell_1$ penalty (regularizer) instead of an $\ell_2$, which has the effect of forcing some of the coefficients to be exactly equal to zero when the tuning parameter λ is sufficiently large.

Thus, the lasso performs variable/feature selection.

## Lasso vs. Ridge Regression

The lasso has a major advantage over ridge regression, in that it produces simpler and more interpretable models that involve only a subset of predictors.

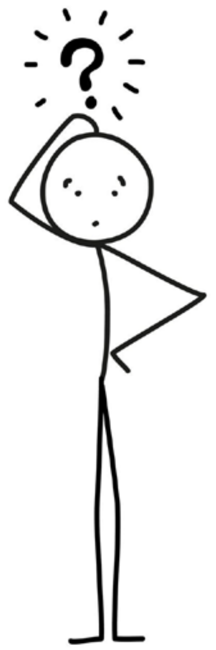
The lasso leads to qualitatively similar behavior to ridge regression, in that as $\lambda$ increases, the variance decreases and the bias increases.

The lasso can generate more accurate predictions compared to ridge regression.

Cross-validation can be used in order to determine which approach is better on a particular data set.

Subgradient methods can be used to compute regression coefficients with an $\ell_1$ regularizer in lasso. Proximal gradient methods are even more effective.

## How did we compute the parameters of a logistic regression?



# Logistic Regression (Recap)

The logistic function is an example of a sigmoid function often used in feed-forward neural networks as activation function.

$$\Pr[Y_i = 1|X] = p(X) = \frac{e^{\beta_0+\beta_1 X}}{1+e^{\beta_0+\beta_1 X}} = \sigma(\beta_0 + \beta_1 X_{1i})$$

- $\Pr(Y_i = 1) \rightarrow$ 0 as $\beta_0 + \beta_1 X_{1i} \rightarrow -\infty$
- $\Pr(Y_i = 1) \rightarrow$ 1 as $\beta_0 + \beta_1 X_{1i} \rightarrow \infty$

Likelihood function models a sequence of Bernoulli trials

$$L = \prod_{i=1} p^{y_i}(1-p)^{1-y_i} = \prod_{i=1} \sigma(\beta_0 + \beta_1 X_{1i})^{y_i} * [1 - \sigma(\beta_0 + \beta_1 X_{1i})]^{1-y_i}$$

## The Likelihood Function for the Logit Model

Use $L = \prod_{i=1} p^{y_i}(1-p)^{1-y_i}$ and take the log to get the log likelihood

$$LL = \ln(L) = \sum_{i=1} y_i \ln p_i + (1-y_i)\ln(1-p_i)$$

We look for the vector $\beta$ that maximizes $LL$ (or minimizes $-LL$)

$$\beta = \text{argmax}_\beta LL(\beta) = \text{argmax}_\beta \left[ \sum_{i=1} y_i \ln \sigma(\beta_0 + \beta_1 X_{1i}) + (1-y_i)\ln(1-\sigma(\beta_0 + \beta_1 X_{1i})) \right]$$

The $LL$ function is twice differentiable and concave!

- For the OLS estimator, we set the FOC=0 and get an analytical solution.
- For the logistic regression, this does **not get us a closed-form solution** due to the nonlinearity of the logistic sigmoid function.
- We can use a numerical algorithm to find the maximum such as **gradient ascent**!

# The Negative LL Function is Convex

Is the *negative* $LL$ function convex?

$$\sigma(\beta_0 + \beta_1 X_{1i}) = \frac{e^{\beta_0+\beta_1 X}}{1+e^{\beta_0+\beta_1 X}} = \frac{1}{1+e^{-(\beta_0+\beta_1 X_1)}} = \frac{1}{1+e^{-z}} = \sigma(z)$$

$$-LL(\beta) = \sum_{i=1} -y_i \ln \sigma(z) - (1-y_i)\ln(1-\sigma(z))$$

It is known that

- a convex function $f$ of an affine function (e.g. $\beta_0 + \beta_1 X_1$) is convex, i.e., $f(\beta_0 + \beta_1 X_1)$ is convex.
- the sum of convex functions is convex

  i.e., $\sum_{i=1} -y_i \ln \sigma(z) - (1-y_i)\ln(1-\sigma(z))$ is convex, if $-\ln(\sigma(z))$ and $-\ln\big(1 -$

Lets define $\underline{f_1(z) = -\ln(\sigma(z))}$ and $\underline{f_2(z) = -\ln(1-\sigma(z))}$

and $\underbrace{f(z) = y_i f_1(z) + (1-y_i) f_2(z)}$

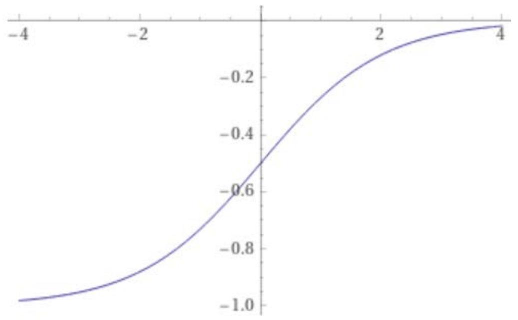
# The Negative LL Function is Convex

1. $f_1(z)$ is convex:

$$f_1(z) = -\ln\left(\frac{1}{1+\exp(-z)}\right) = \ln\big(1+\exp(-\mathrm{z})\big)$$

The derivative is monotonically increasing.

$$\frac{d}{dz} f_1(z) = -\frac{1}{1+\exp(z)}$$



2. $f_2(z)$ is convex:

$$f_2(z) = -\ln\left(1 - \frac{1}{1+\exp(-z)}\right) = -(\ln(\exp(-z) - \ln(1+\exp(-z))$$

The derivative of $f_2(z)$ is also monotonically increasing.

$$\frac{\mathrm{d}}{dz} f_2(z) = \frac{\mathrm{d}}{dz}(z + \ln(1+\exp(-z))) = \frac{\exp(z)}{1+\exp(z)} = \frac{1}{1+\exp(-z)}$$

## Gradient for the LL Function

$$\beta = \mathrm{argmax}_{\beta} LL(\beta) = \mathrm{argmax}_{\beta}\left[\sum_{i=1} y_i \ln \sigma(\mathbf{X}\beta) + (1-y_i)\ln(1-\sigma(\mathbf{X}\beta))\right]$$

$LL(\beta) = y \ln p + (1-y)\ln(1-p)$ for one sample

$p = \sigma(z), z = \mathbf{X}\beta$ short hands

$\frac{\partial LL(\beta)}{\partial \beta_j} = \frac{\partial LL(\beta)}{\partial p} * \frac{\partial p}{\partial z} * \frac{\partial z}{\partial \beta_j}$ chain rule

$$\frac{\partial p}{\partial z} = \frac{\partial \sigma(z)}{\partial z} = \frac{\partial}{\partial z}\frac{\exp(z)}{1+\exp(z)} = \frac{\exp(z)}{(1+\exp(z))^2} = \frac{\exp(z)(1+\exp(z)) - \exp(2z)}{(1+\exp(z))^2} =$$

$$\frac{\exp(z)}{1+\exp(z)} - \left(\frac{\exp(z)}{1+\exp(z)}\right)^2 = \sigma(z) - \left(\sigma(z)\right)^2 = \sigma(z)(1-\sigma(z))$$

$$\equiv \frac{\partial p}{\partial z} = \sigma(z)[1-\sigma(z)]$$

## Gradient for the LL Function

$$\beta = \text{argmax}_\beta LL(\beta) = \text{argmax}_\beta \left[\sum_{i=1} y_i \ln \sigma(\mathbf{X}\beta) + (1-y_i)\ln(1-\sigma(\mathbf{X}\beta))\right]$$

We can use the chain rule:

$$LL(\beta) = y \ln p + (1-y)\ln(1-p)$$

$$\frac{\partial LL(\beta)}{\partial p} = \frac{y}{p} - \frac{1-y}{1-p}$$

$$p = \sigma(z)$$

$$\frac{\partial p}{\partial z} = \sigma(z)[1-\sigma(z)]$$

$$z = \mathbf{X}\beta$$

$$\frac{\partial z}{\partial \beta_j} = x_j$$

$$\frac{\partial LL(\beta)}{\partial \beta_j} = \frac{\partial LL(\beta)}{\partial p} * \frac{\partial p}{\partial z} * \frac{\partial z}{\partial \beta_j} =$$

$$\left[\frac{y}{p} - \frac{1-y}{1-p}\right]\sigma(z)[1-\sigma(z)]x_j =$$

since $p = \sigma(z)$

$$\left[\frac{y}{p} - \frac{1-y}{1-p}\right]p[1-p]x_j =$$

$$[y(1-p) - p(1-y)]x_j =$$

$$[y-p]x_j =$$

$$[y-\sigma(\mathbf{X}\beta)]x_j$$ → gradient

# Gradient Ascent for the Likelihood Function

We want to choose parameters ($\beta$) that maximize the likelihood, and we know the partial derivative of the log likelihood (LL) with respect to each parameter.
The LL function is concave, but no closed-form solution exists for the derivative. So, we can use batch gradient ascent to maximize the log likelihood.

```
Repeat many times:
      For each parameter $0 \leq j \leq p$ do:
          Compute the gradient for each training example $(\boldsymbol{x}_i, y_i)$
```

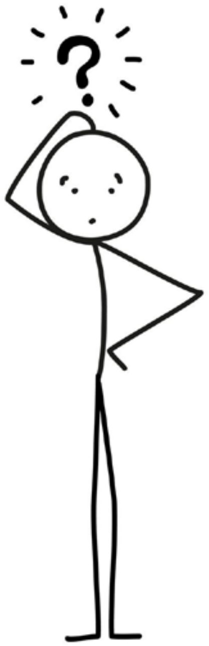
$$\boxed{\beta_j^{new} = \beta_j^{old} + \alpha * \frac{\partial LL(\beta^{old})}{\partial \beta_j^{old}}}$$

$$= \beta_j^{old} + \alpha * \frac{1}{n} \sum_i [y_i - \sigma(\beta^T \boldsymbol{x}_i)] x_{ij}$$

$$= \boxed{\beta_j^{old} + \alpha * \frac{1}{n} \sum_i \left[ y_i - \frac{\exp(\beta^T \boldsymbol{x}_i)}{1 + \exp(\beta^T \boldsymbol{x}_i)} \right] x_{ij}}$$

Is this relevant to neural networks?

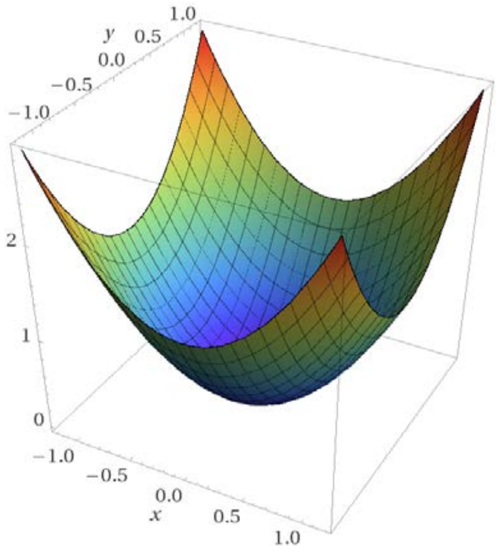
What exactly are neural networks after all?

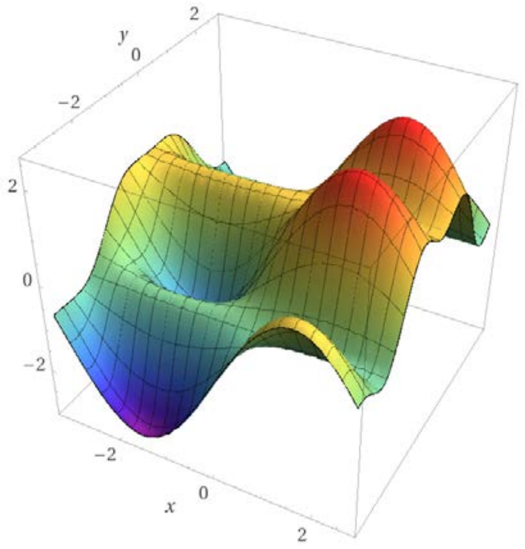


# Linear Models vs. Non-Linear Neural Networks

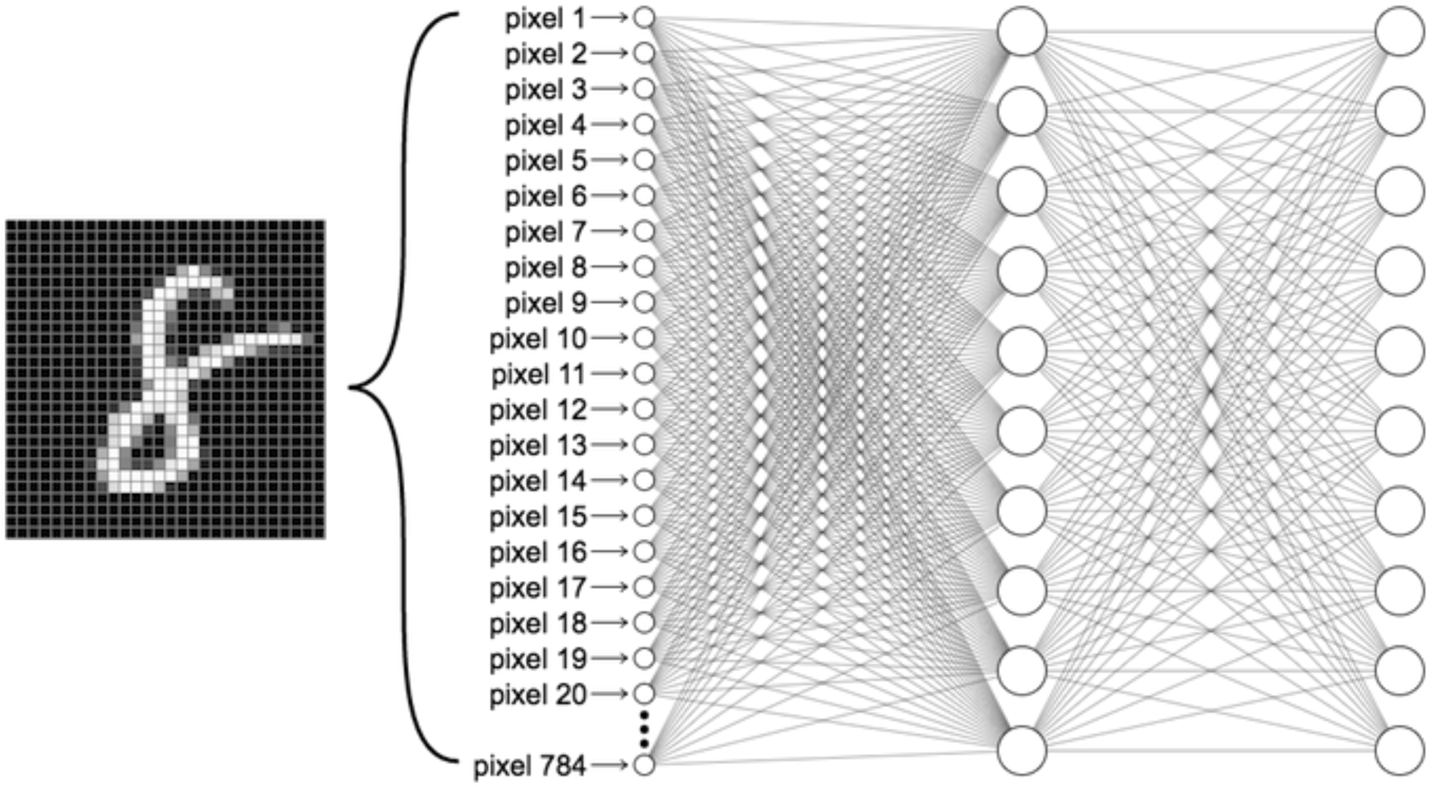
- Convex optimization
- Good for CPUs
- Reliable convergence
- Few hyperparameters
- Simple to interpret
- Requires feature engineering

- Non-convex optimization of the loss fctn.
- Often difficult to train
- Many hyperparameters
- Hard to interpret
- Automatically explores feature-engineering for the user



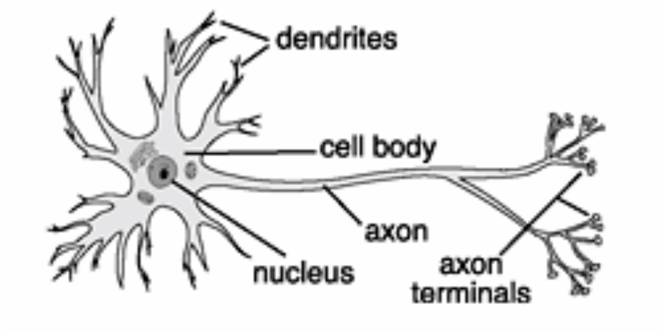


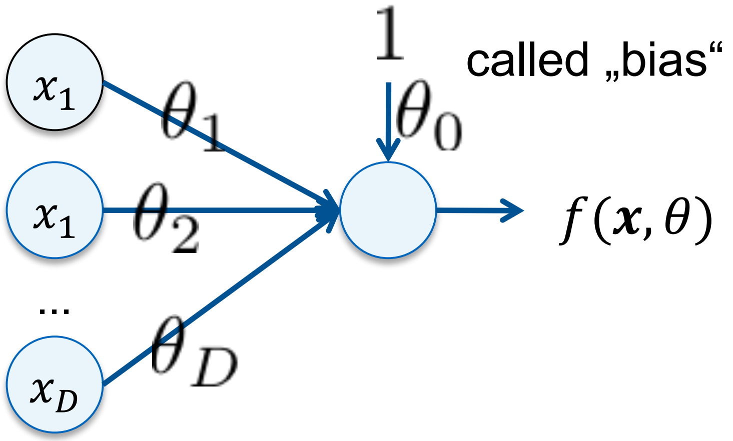
## Classification via Neural Networks



## Neuron vs. Regression

- Edges multiply the signal ($x_i$) by some weight ($\theta_i$) as a simple model
- Nodes sum inputs
- Equivalent to **linear regression** $f(x,\theta) = \sum_{d=1}^{D} \theta_d x^d + \theta_0$
- The bias determines when the neuron fires

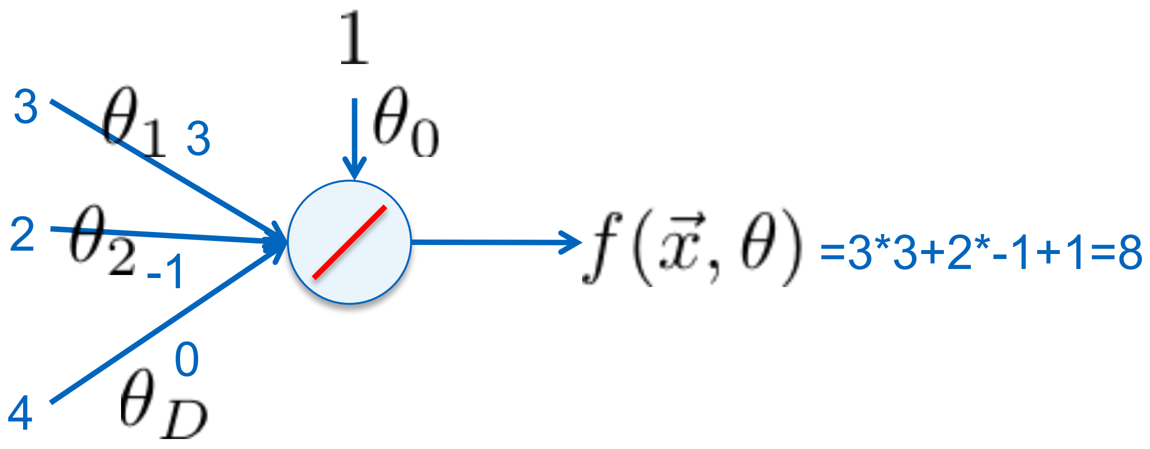




## Linear Activation Function

Each neuron has its own bias and weights:

$$f(\vec{x},\theta) = \sum_{n=0}^{N-1}\sum_{d=1}^{D}\theta_d\phi_d(x_n) + \theta_0 \qquad f(\vec{x},\theta) = \theta^T\vec{x}$$

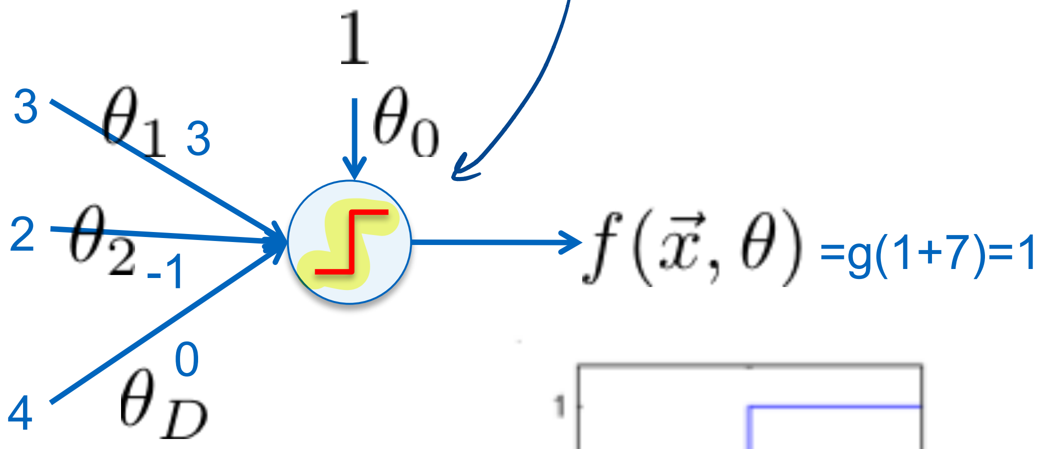


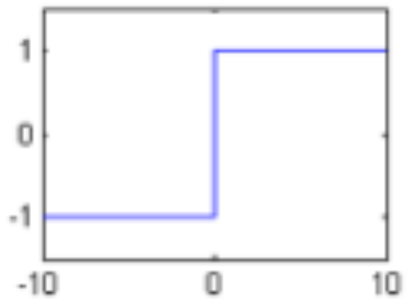
Linear neuron

## Perceptron

Binary classification via single-layer neural network.

$$g(z) = \begin{cases} -1 \text{ when } z < 0 \\ +1 \text{ when } z \geq 0 \end{cases}$$



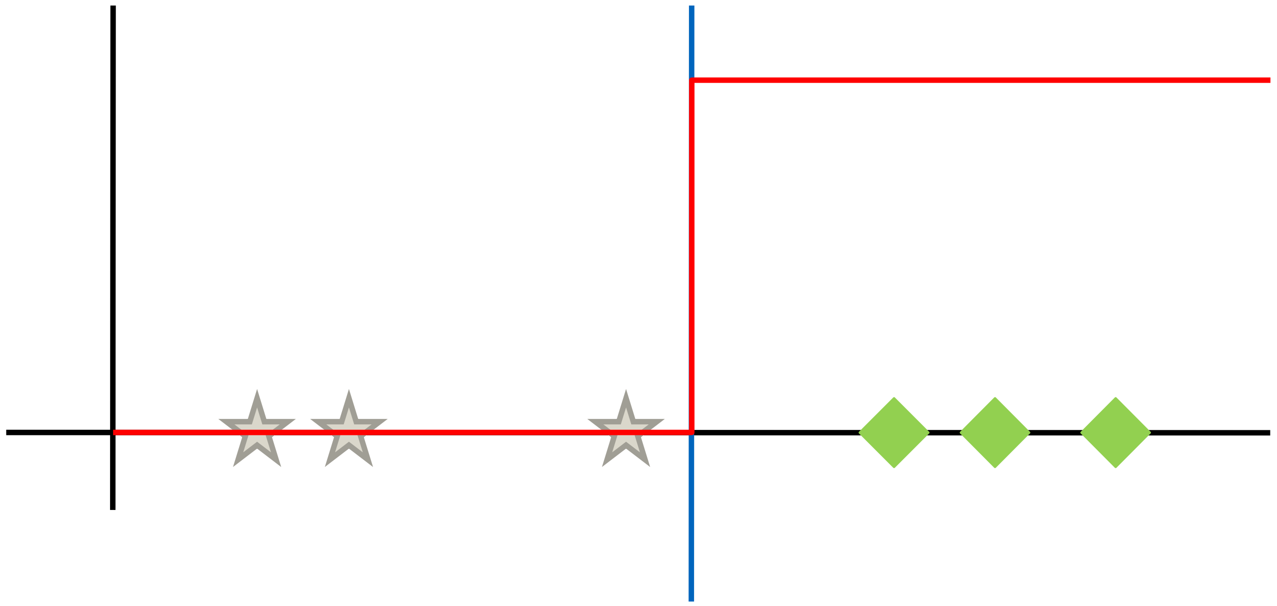


Classification error

## Perceptron Error

We can't do gradient descent (see next class) with step functions, as it is non-differentiable at one point and otherwise the gradient is zero.

.

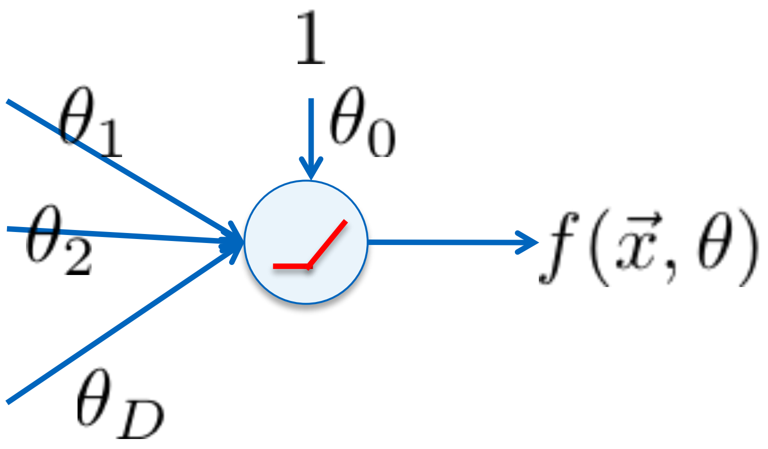


## ReLU Activation Function

ReLU activation (is now often used in DNNs)

(see https://en.wikipedia.org/wiki/Activation_function)

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

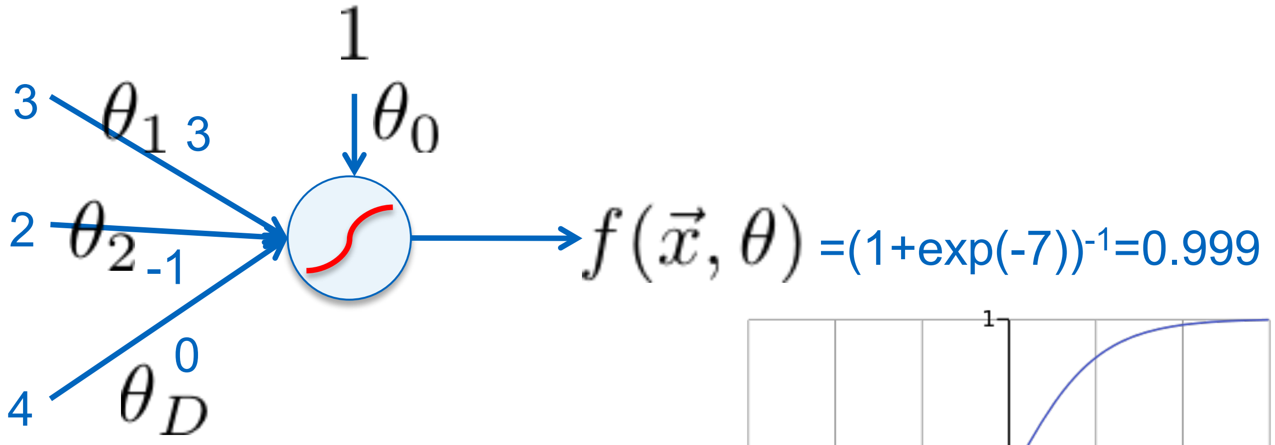


ReLU Neuron

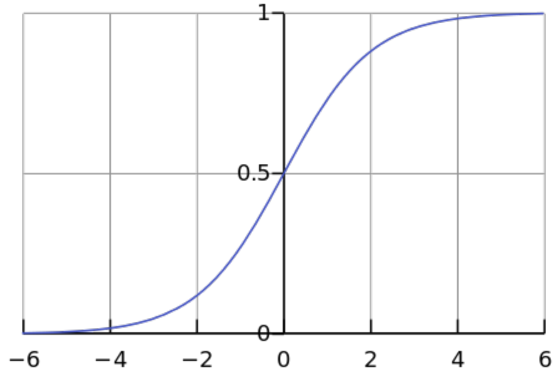
# Sigmoid Activation Function

Sigmoid function (special case logistic function)

$$f(\vec{x}, \theta) = g(\theta^T \vec{x}) \qquad g(z) = (1 + exp(-z))^{-1}$$



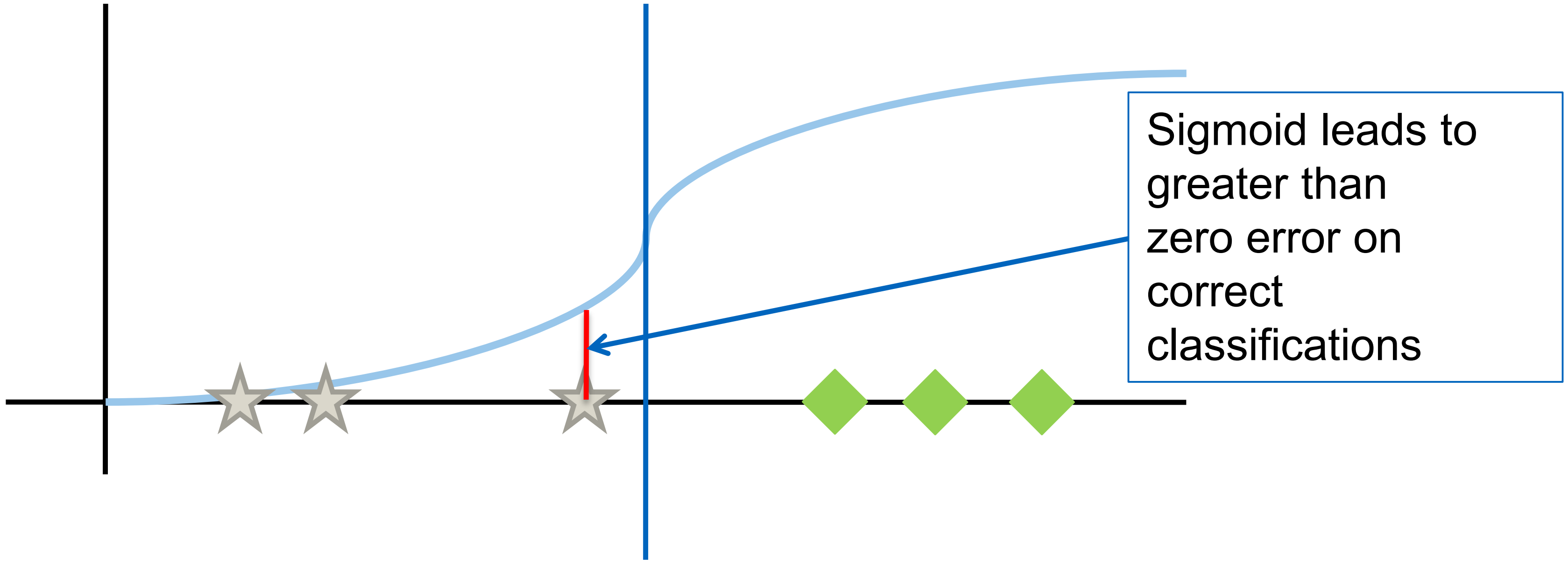
Logistic Neuron



# Classification Error

Only count errors when a classification is incorrect.

$$R(\theta) = \frac{1}{N}\sum_n L(y_n, f(x_n)) = \frac{1}{2N}\sum_n \left(y_n - g(\theta^T x_n)\right)^2$$


Sigmoid leads to greater than zero error on correct classifications

## Example

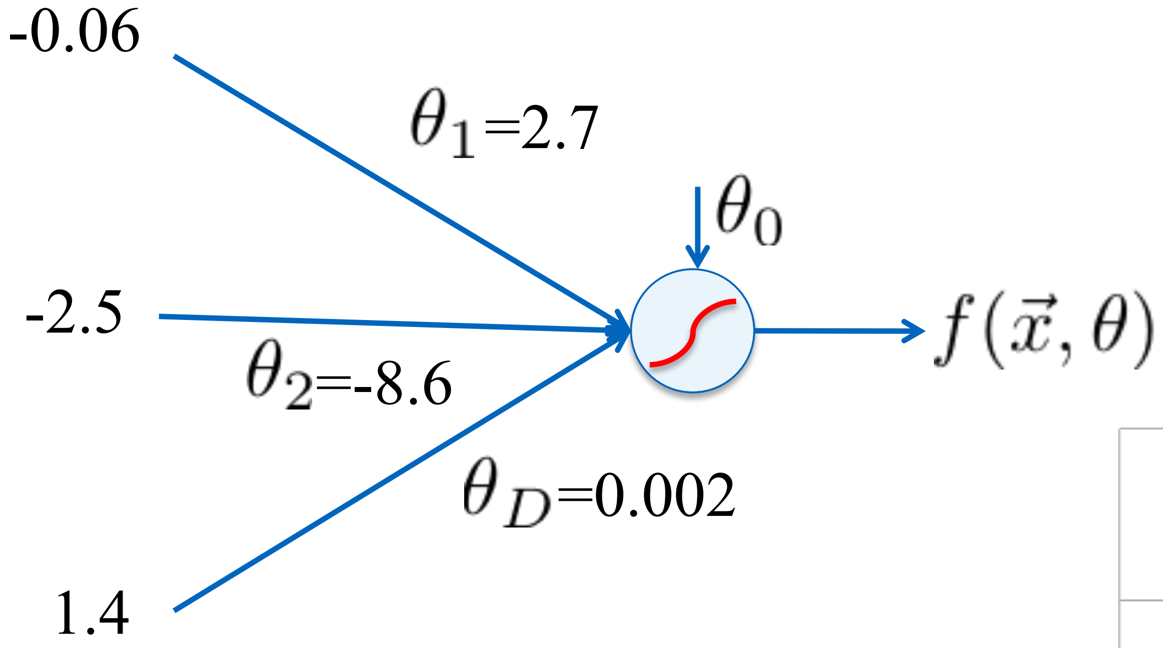
Activation function:

$f(\vec{x}, \theta) = \mathrm{g}(g^T \vec{x})$

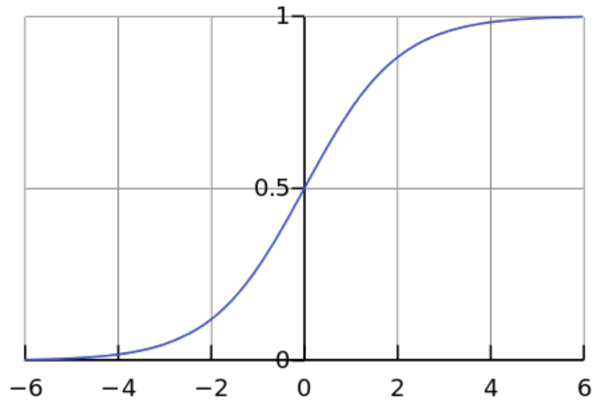
bias

$\theta^T \vec{x}$ = -0.06×2.7 + 2.5×8.6 + 1.4×0.002 + 4 = 25.34

The logistic function translates into values 0 to 1.



$$g(z) = (1 + exp(-z))^{-1}$$

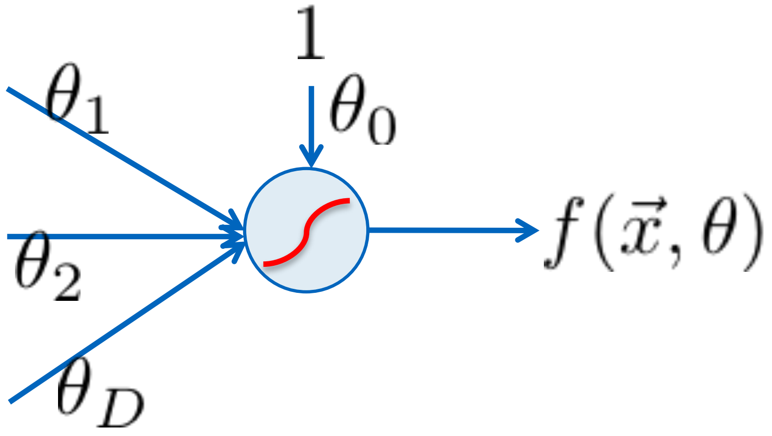


# Example

Training data set

| Fields | | | Class |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc. | … | | |

**Initialize with random weights.**

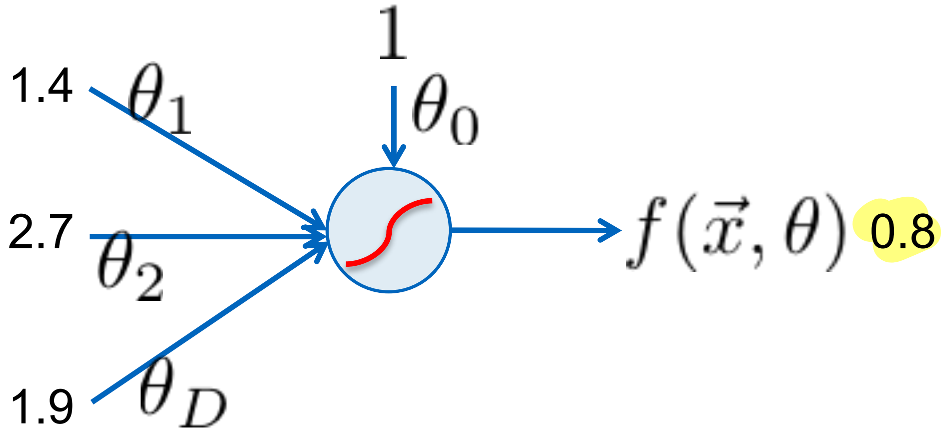


# Example

Training data set

| Fields | | | Class |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc. | … | | |

Initialize with random weights.
Present a training pattern.
Feed it through to get output.

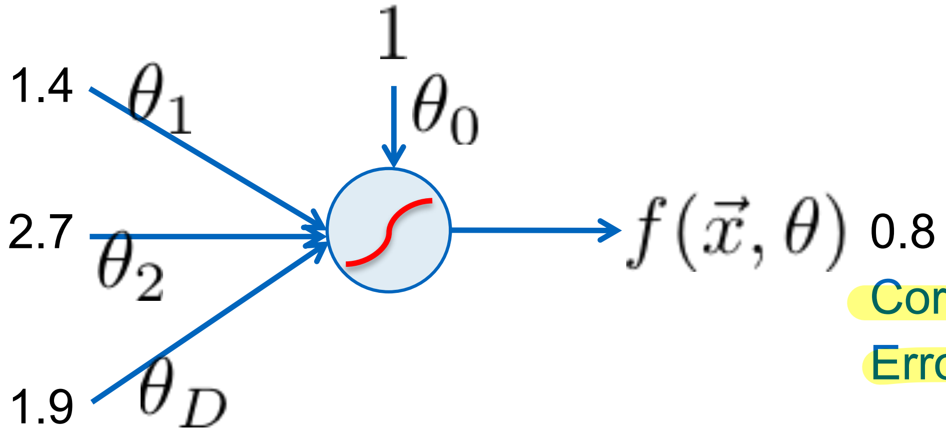


## Example

Training data set

| Fields | | | Class |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc. | … | | |

Initialize with random weights.
Present a training pattern.
Feed it through to get output.



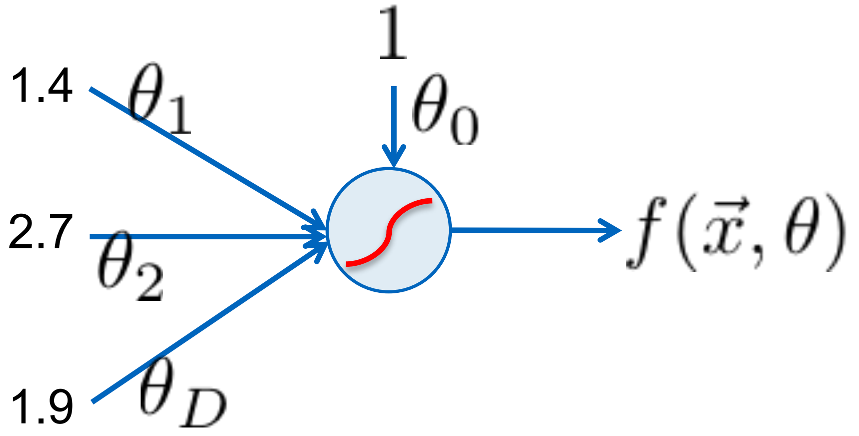
Correct 0
Error $0.8^2$

# Example

Training data set

| Fields | | | Class |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc. | … | | |

Initialize with random weights.
Present a training pattern.
Feed it through to get output.
**Adjust weights θ *(see next class)*.**



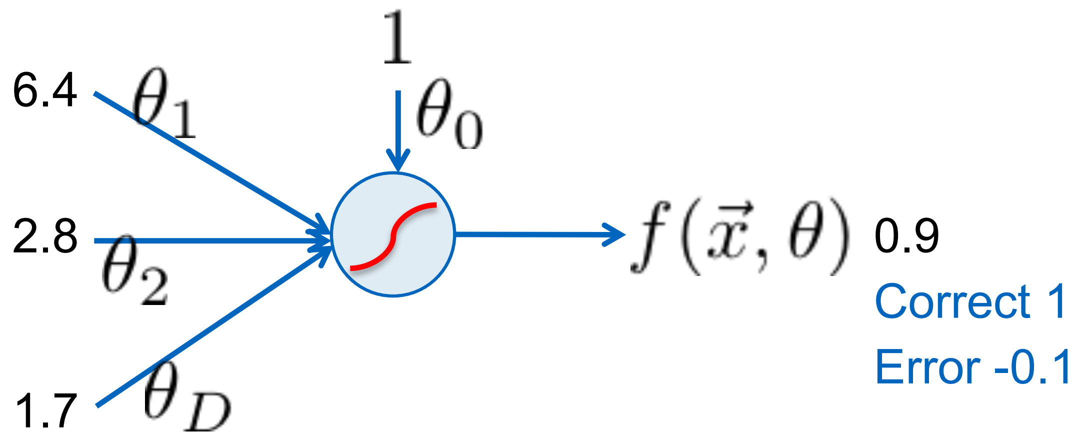
0.8
Correct 0
Error $0.8^2$

# Example

Training data set

| Fields | | | Class |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc. | … | | |

Present another training pattern.
Feed it through to get output.
Adjust weights.

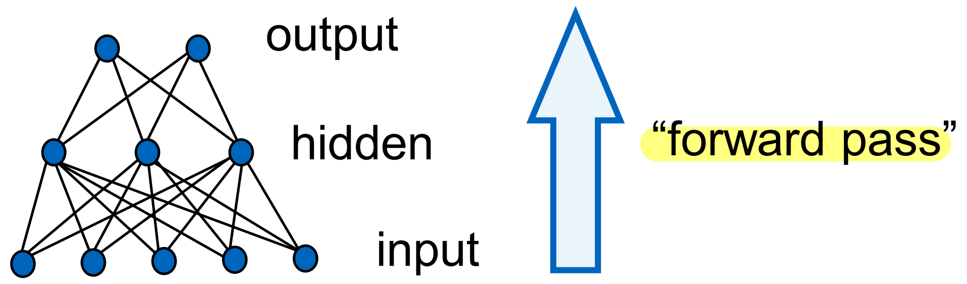


Repeat this thousands, maybe millions of times – each time taking a random training instance, and making slight weight adjustments.

# Multi-Layer Feed-Forward Networks

Multi-layer networks can represent arbitrary non-linear functions.

A typical multi-layer network consists of an input, hidden and output layer, each fully connected to the next, with activation feeding forward.

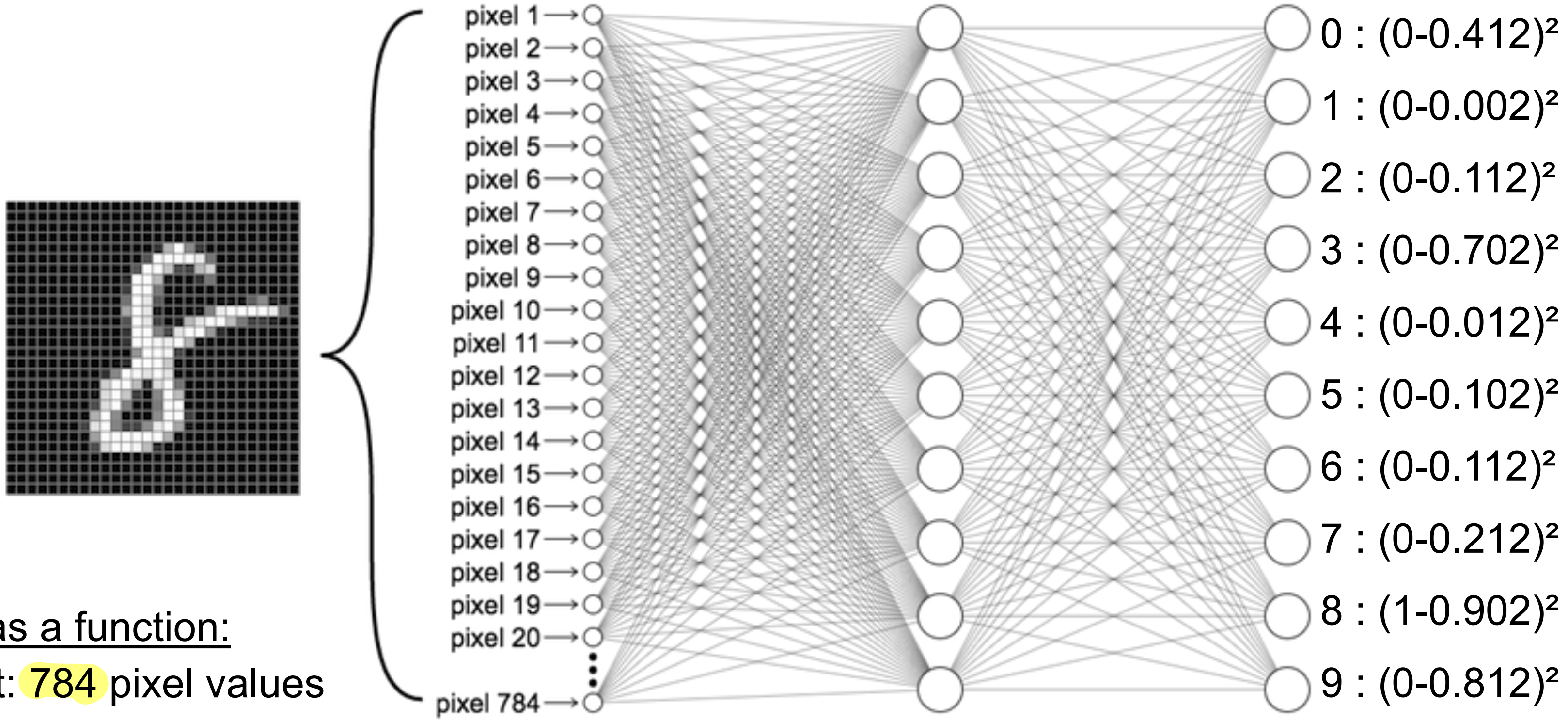


Multi-layer networks have thousands of weights and biases that need to be adapted. The weights determine the overall function computed.

Weights are updated via backpropagation (i.e. gradient decent).

## Loss Function

**Use a loss function** (aka. cost function) to compute the average (misclassification) cost in the example below adds up the squared error for a single training example. If we average over all training examples, this is referred to as **the empirical risk function (total cost function)**.



<u>NN as a function:</u>

Input: 784 pixel values

Output: 10 numbers

Parameters: 784*10+10*10=7940 weights plus 20 biases

# Empirical Risk Minimization

Minimizing the **empirical risk function** $R(\theta)$, which is modeling **expected loss** $\mathbb{E}[L(y, f(x))]$ (as we don't know the true distribution of data).
This means, the empirical risk $\boldsymbol{R(\theta)}$ is the average loss over the training data.

$$R(\theta) = \frac{1}{N}\sum_n L(y_n, f(x_n)) = \frac{1}{2N}\sum_n \left(y_n - g(\theta^T x_n)\right)^2$$

↳ sum all the pictures

derivative of $f(z)^2 \Rightarrow 2f(z)f'(z)$ (chain rule)

$$\nabla_\theta R = \frac{1}{2N}\sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$

$$g(z) = (1 + exp(-z))^{-1}$$

Unfortunately, there is no "closed-form" solution. We can backpropagate the error for after the gradients of the data instances are accumulated (batch GD) or after individual training samples (SGD).
To backpropagate a single training example, we use **gradient descent**.

# Minimizing a Loss Function in a Nutshell

We use the gradient descent algorithm to adapt the weights such that the loss function is minimized. For this we use the steepest descent of a function.

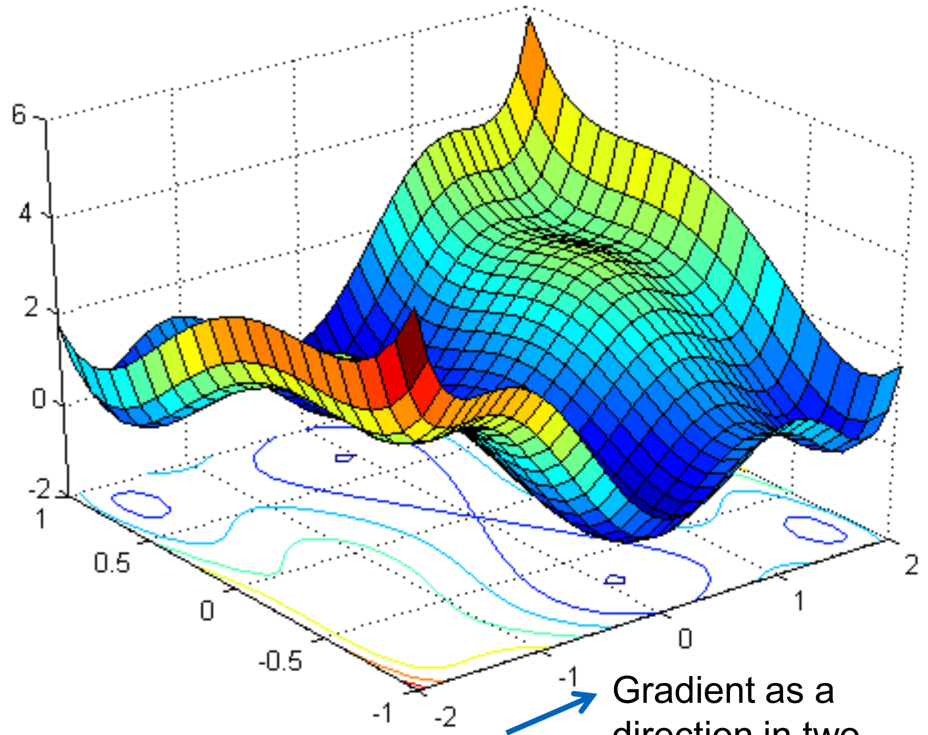
The negative gradient shows the local shift in weights of the NN that helps most in minimizing the loss function.

**Loss function:**

Input: 7960 parameters (weights + biases)

Output: scalar (cost)

Adapting the weights and biases for all training examples via gradient descent results in the negative gradient of the empirical risk function.



Gradient as a direction in two input dimensions

→ **Learning = optimization = minimizing the loss function!**