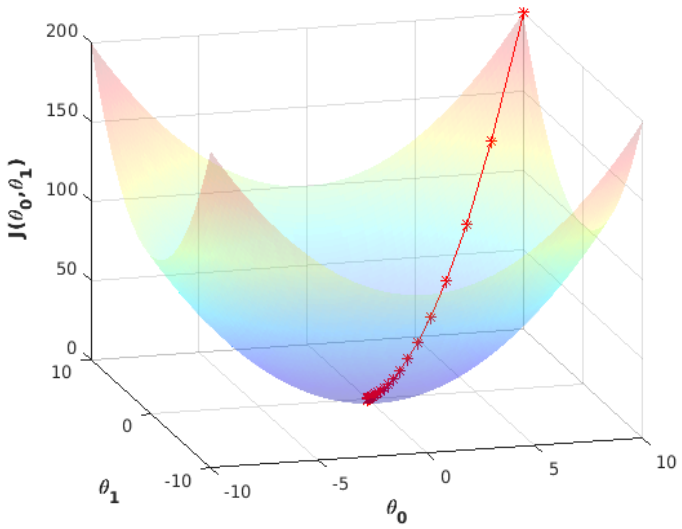
Lecture/Workshop: Introduction to Neural Networks

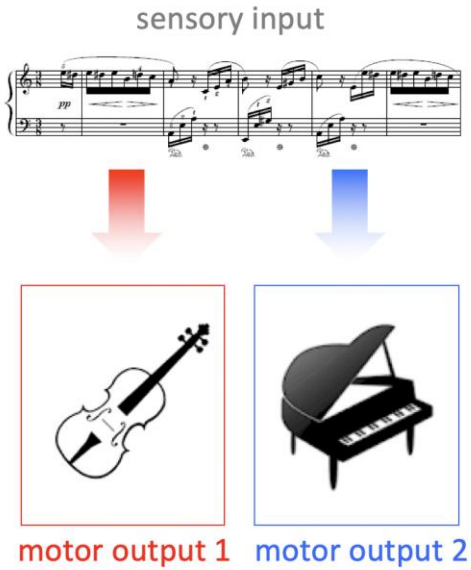


Timo Flesch
University of Oxford

My Background

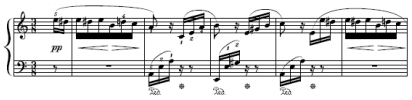
- **10/2012-9/2015: BSc Cognitive Science,**
  Universität Osnabrück
  - 2014/15: Erasmus Internship Human Information Processing Lab, University of Oxford
  - 2013,2014,2015: Teaching assistant in Mathematics, (Symbolic) Artificial Intelligence and Machine Learning
- **9/2015-9/2018: Graduate Research Assistant,**
  Human Information Processing Lab, University of Oxford
  - Projects on human perceptual decision making, category learning, curriculum learning.
  - EEG & Behaviour data , Computational Modeling
- **9/2016-9/2018: PgDip Computational Statistics and Machine Learning,**
  University College London
  - Spent 2-3 days a week in London
  - Learned a bit more about machine learning
- **Since 10/2018: DPhil (PhD) Experimental Psychology,**
  Human Information Processing Lab, University of Oxford
  - „Representation Learning for Continual Task Performance“. EEG, fMRI, Deep Learning

My Research: Continual Learning

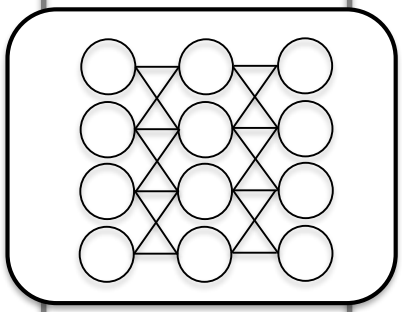


Humans and other animals continue to acquire new knowledge over their long lifespans

My Research: Continual Learning

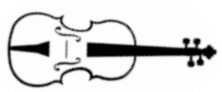


x



$y_1 = f(x, c_1)$

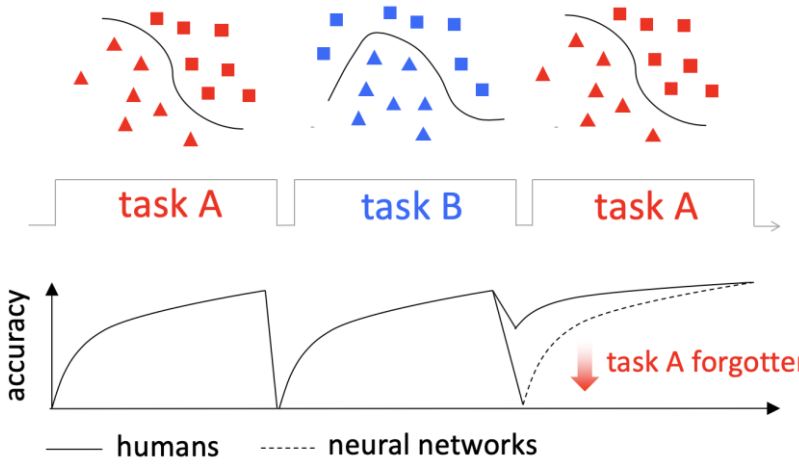
$y_2 = f(x, c_2)$



motor output 1

motor output 2

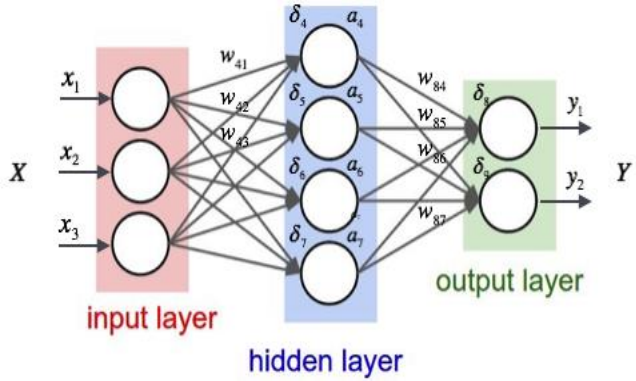
function approximator



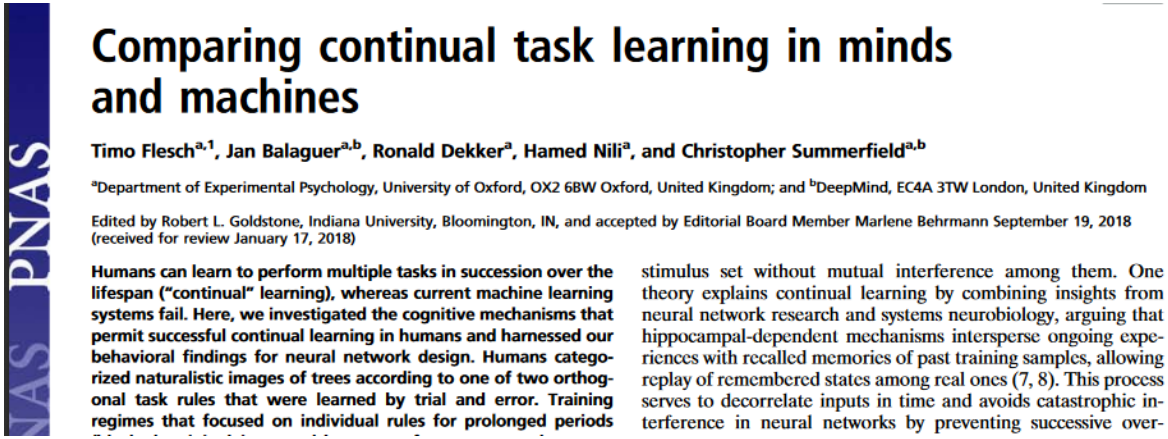
Supervised neural networks, but not humans, suffer from catastrophic interference during sequential task performance

Outline

- Today: Intro to Neural Networks



- Thursday: Case Study



PNAS

Comparing continual task learning in minds and machines

Timo Flesch[a,1], Jan Balaguer[a,b], Ronald Dekker[a], Hamed Nili[a], and Christopher Summerfield[a,b]

[a]Department of Experimental Psychology, University of Oxford, OX2 6BW Oxford, United Kingdom; and [b]DeepMind, EC4A 3TW London, United Kingdom

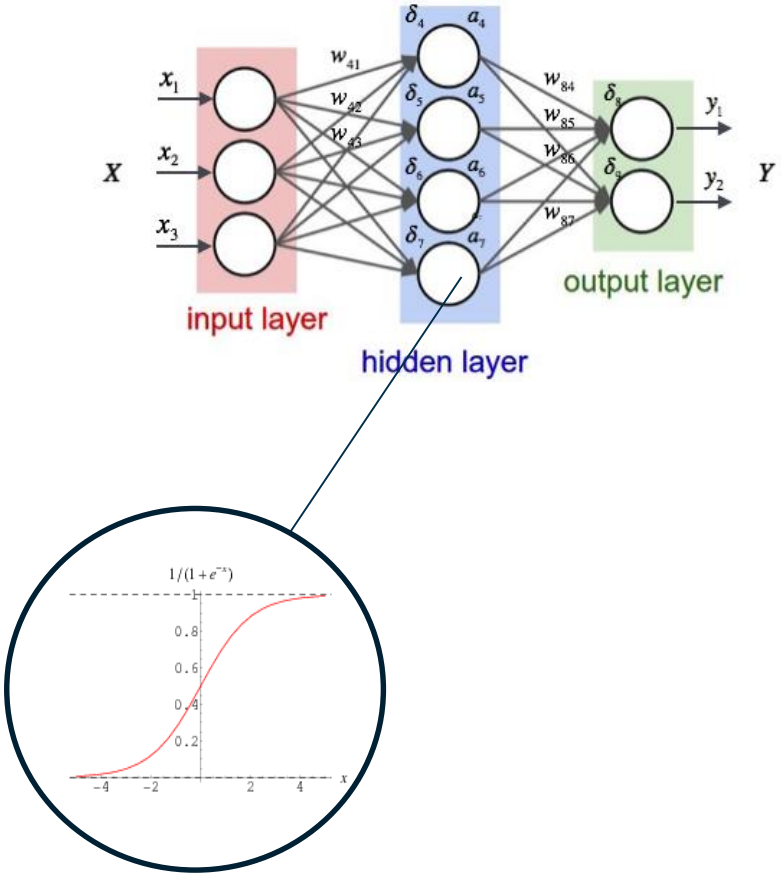
Edited by Robert L. Goldstone, Indiana University, Bloomington, IN, and accepted by Editorial Board Member Marlene Behrmann September 19, 2018 (received for review January 17, 2018)

Humans can learn to perform multiple tasks in succession over the lifespan ("continual" learning), whereas current machine learning systems fail. Here, we investigated the cognitive mechanisms that permit successful continual learning in humans and harnessed our behavioral findings for neural network design. Humans categorized naturalistic images of trees according to one of two orthogonal task rules that were learned by trial and error. Training regimes that focused on individual rules for prolonged periods

stimulus set without mutual interference among them. One theory explains continual learning by combining insights from neural network research and systems neurobiology, arguing that hippocampal-dependent mechanisms intersperse ongoing experiences with recalled memories of past training samples, allowing replay of remembered states among real ones (7, 8). This process serves to decorrelate inputs in time and avoids catastrophic interference in neural networks by preventing successive over-

What is a Neural Network?

- It‘s a composition of (non)linear functions!
- Compositions of simple functions can approximate very complex functions!



What is it Good for?

„apple“

„banana“

What is it Good for?

- Is red
- ~~Is yellow~~

„apple“

- ~~Is red~~
- Is yellow

„banana“

What is it Good for?

- Is red
- ~~Is yellow~~

„apple“

- ~~Is red~~
- Is yellow

„banana“

- ~~Is red~~
- Is yellow

„banana“

What is it Good for?

- ~~Is round~~
- Is elongated

„banana“

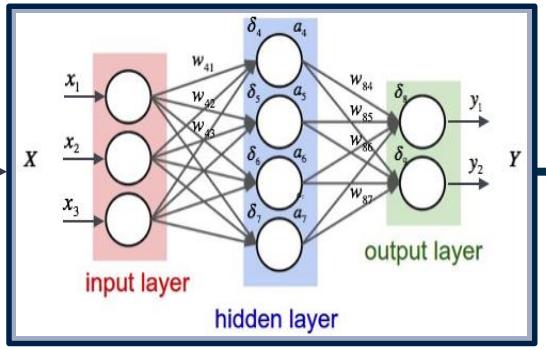
- Is round
- ~~Is elongated~~

„apple“

Defining a rule-based system quickly becomes infeasible, given how complex the real world is!
Furthermore, symbolic operations may reveal little about information processing in the brain!

What is it Good for?

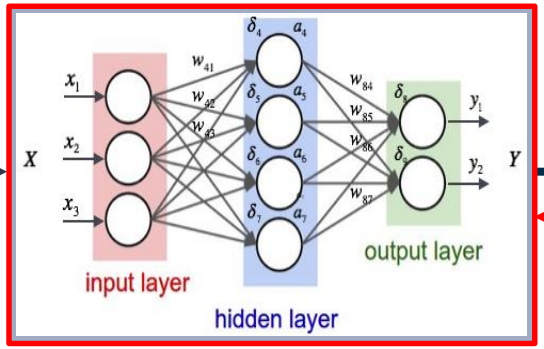
- Let‘s use a neural network instead.
- We set up a network with a couple of nonlinear functions and assign random weights to each of the functions.
- Initially, the predictions will be wrong. That‘s where ***learning*** comes into play!



„banana“

What is it Good for?

- For each prediction, we compare it to the true label and adjust its parameters to reduce the error

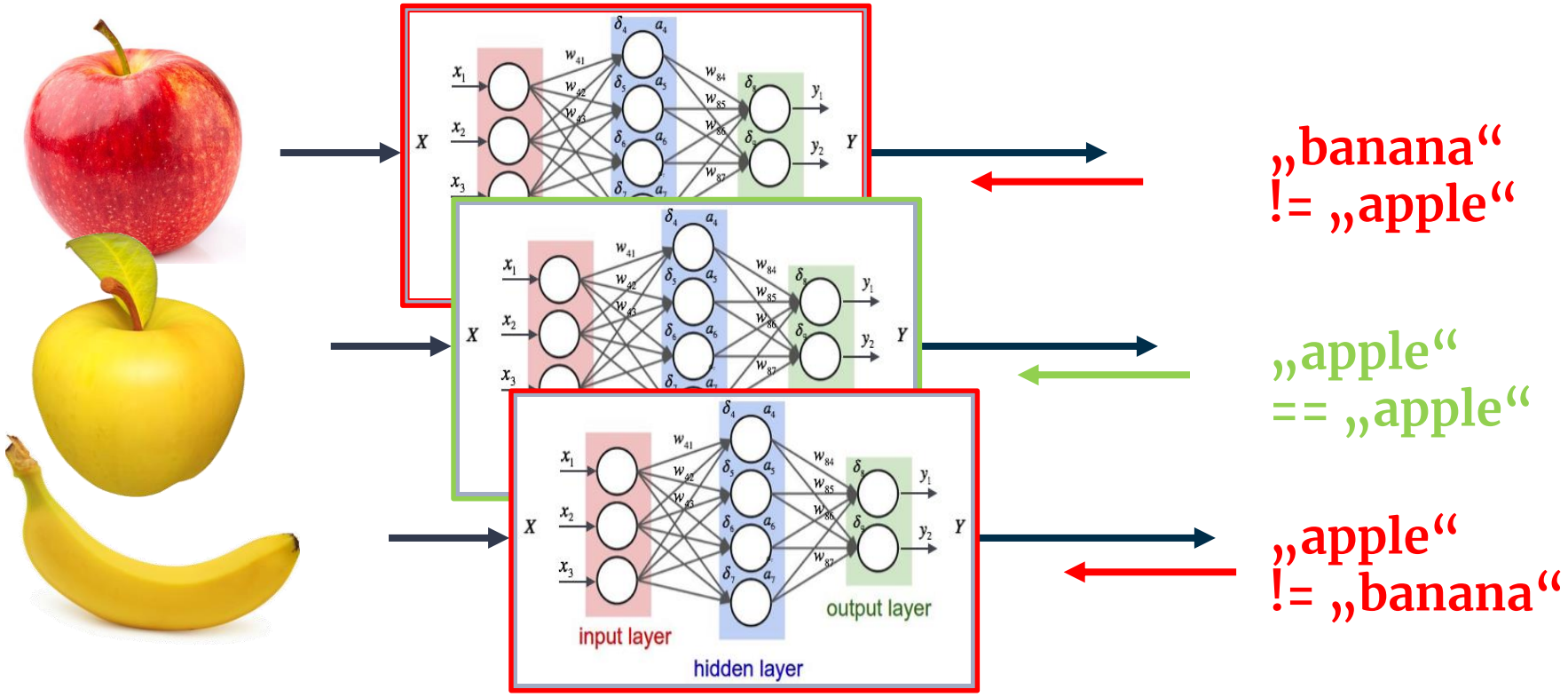


„banana“

!= „apple“

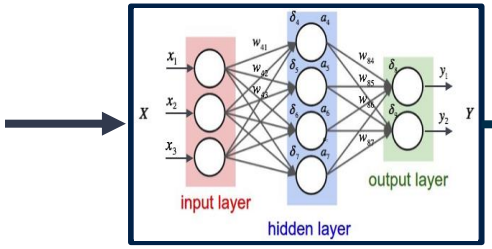
What is it Good for?

- For each prediction, we compare it to the true label and adjust its parameters to reduce the error
- **We do this for many different inputs, hundreds of times!**

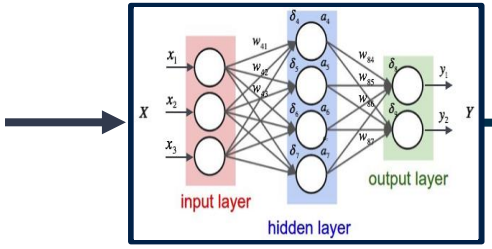


What is it Good for?

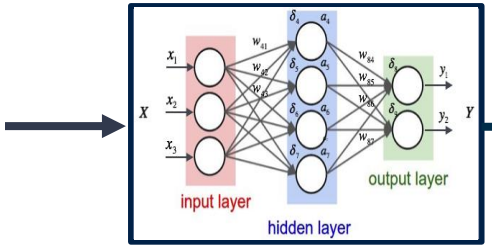
- In the end, we‘ve **trained** our network to **approximate** the unknown function that assigns labels of apples and bananas to arbitrary images of apples and bananas!



„apple“



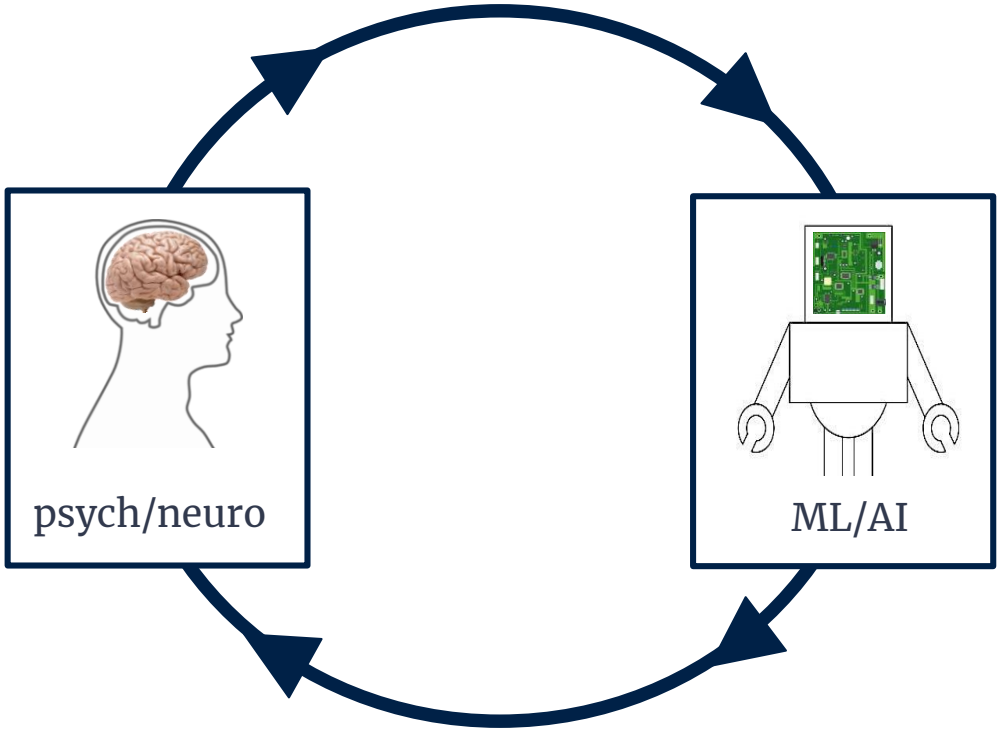
„banana“



„apple“

Why Should a Neuroscientist Care?

By understanding brains,
we can build them better



By building brains,
we can understand them better

In neuroscience, we lack mature theories of representation learning

We can import tools from machine learning as computational theory (e.g. deep networks)

But we should not do this in an unexamined way...

The Roadmap for Today

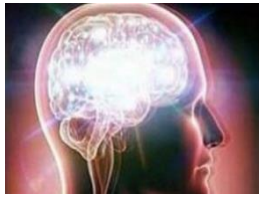
Maths Refresher (1 slide!)

1

Building blocks of neural networks and supervised learning

- Linear regression
- Logistic regression
- Gradient descent

2



Neural networks as compositions of functions

- Architecture
- Training a neural network: backpropagation of errors

3

Maths Refresher

- Derivative: Rate of change of function wrt a variable. $f\colon \mathbb{R} \longrightarrow \mathbb{R}$

$$f(x) = ax^n \quad \frac{\mathrm{d}}{\mathrm{dx}}\,\mathrm{ax^n} = \mathrm{nax^{n-1}}$$

- Partial derivative: For functions with more than one variable $f\colon \mathbb{R}^n \longrightarrow \mathbb{R}$

$$f(x_1, x_2) = ax_1^n + bx_2^m \qquad \frac{\partial}{\partial x_1}\, ax_1^n + bx_2^m = nax_1^{n-1} \qquad \frac{\partial}{\partial x_2}\, ax_1^n + bx_2^m = mbx_2^{m-1}$$

- Gradient: Vector of Partial Derivatives $f\colon \mathbb{R}^n \longrightarrow \mathbb{R}$

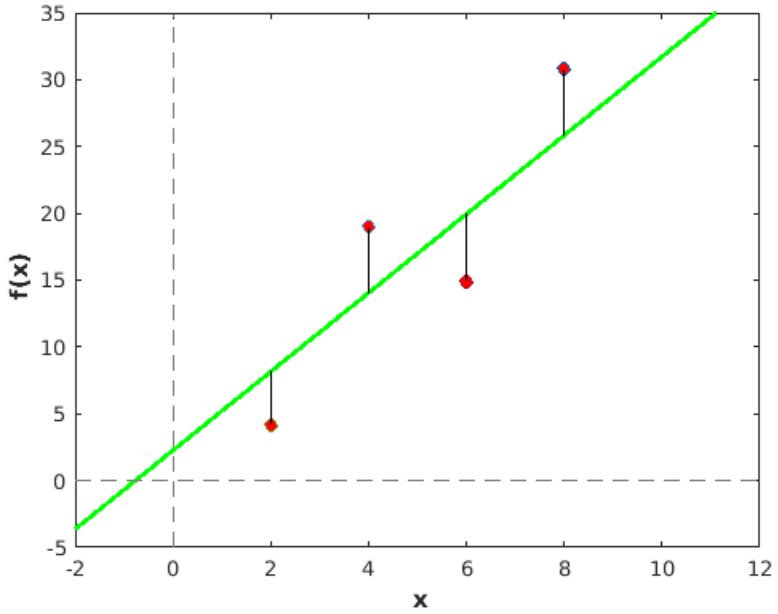
$$\nabla_{\mathrm{x}} f(\boldsymbol{x}) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]$$

- Jacobian: Matrix of Partial Derivatives $f\colon \mathbb{R}^n \longrightarrow \mathbb{R}^m$

$$J_x(fx) = \begin{bmatrix} \frac{\partial f}{\partial x_{11}} & \cdots & \frac{\partial f}{\partial x_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial x_{n1}} & \cdots & \frac{\partial f}{\partial x_{nm}} \end{bmatrix}$$

- Chain Rule of Calculus: Derivative for composition of functions

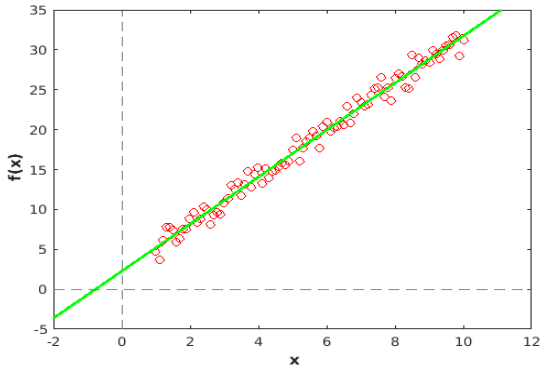
$$f(x) = z\big(g(x)\big) \qquad \frac{df}{dx} = \frac{dz}{dg} * \frac{dg}{dx}$$



Basics 1/3: Linear Regression

Linear Regression – Goal

- You‘ve observed pairs of continous-variable data points, generated by an unknown process:
$$\{(x_i, y_i)\}_{i=1}^{N}$$
- Your task is to approximate the relationship by a function
$$f(x_i) = \hat{y}_i$$
- The relationship appears to be linear, so a straight line will suffice:
$$f(x_i) = x_i$$
- Unless the underlying process is an identity mapping, you need to be able to change the slope and bias of your approximator:
$$f(x_i) = \theta_0 + \theta_1 * x_i = \hat{y}_i$$
- With an *optimal* set of parameters, our function should approximate the observed data very well:



Linear Regression – Approach

- To assess the goodness of fit, we introduce an "objective (or loss) function":

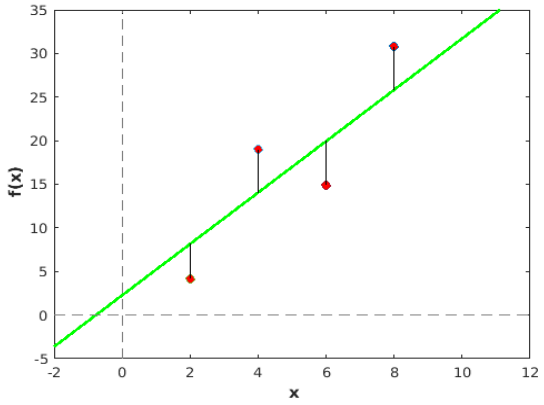
$$J\left(\{(x_i, y_i)\}_{i=1}^{N}, f(x, \boldsymbol{\theta})\right) = "mismatch"(y, \hat{y})$$

- The loss is a function of our model parameters. For linear regression, we choose a squared error loss:

$$J(\boldsymbol{\theta}) = J(\theta_0, \theta_1) = \sum_i^N \left(y_i - f(x_i)\right)^2 = \sum_{i=1}^N (y_i - (\theta_0 + \theta_1 * x_i))^2 = \sum_{i=1}^N (y_i - \theta_0 - \theta_1 * x_i)^2$$

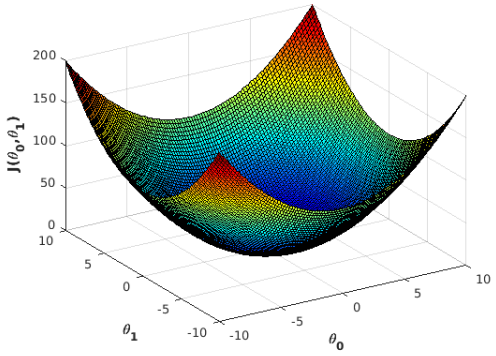
- Our task is now to find the values for our parameters that minimise the loss, i.e. find the best-fitting line:

$$\hat{\theta}_0, \hat{\theta}_1 = \arg\min_{\theta_0, \theta_1} J(\theta_0, \theta_1) = \arg\min_{\theta_0, \theta_1} \sum_{i=1}^N (y_i - \theta_0 - \theta_1 * x_i)^2$$



Linear Regression – Solution

The error function is convex and has a unique minimum:



$$\hat{\theta}_0, \hat{\theta}_1 = \arg\min_{\theta_0,\theta_1} J(\theta_0, \theta_1) = \arg\min_{\theta_0,\theta_1} \sum_{i=1}^{N} (y_i - \theta_0 - \theta_1 * x_i)^2$$

Welcome back to high school! To minimize a function, we only need to set its first derivative to zero and solve for theta. As J has two inputs (bias and slope), we need to compute both partial derivatives:

$$\frac{\partial J}{\partial \theta_0} = \frac{\partial}{\partial \theta_0} \sum_{i=1}^{N} (y_i - \theta_0 - \theta_1 * x_i)^2 = -2 * \sum_{i=1}^{N} (y_i - \theta_0 - \theta_1 * x_i)$$

$$\frac{\partial J}{\partial \theta_1} = \frac{\partial}{\partial \theta_1} \sum_{i=1}^{N} (y_i - \theta_0 - \theta_1 * x_i)^2 = -2 * \sum_{i=1}^{N} x_i * (y_i - \theta_0 - \theta_1 * x_i)$$

For both equations exist „closed form solutions“. That is, we can directly solve for each theta, which gives us the desired OLS estimators:

$$\hat{\theta}_0 = \bar{y} - \theta_1 * \bar{x} \qquad \hat{\theta}_1 = \frac{\sum_{i=1}^{N} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{N} (x_i - \bar{x})^2}$$

With $\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$

Linear Regression – The Multivariable Case

This whole procedure can be easily extended to multivariable regression (with constant term $x_1 = 1$)

$$f(x_1, ..., x_n) = x_1 * \theta_1 + x_2 * \theta_2 + ... + x_n * \theta_n = \sum_{i=1}^{n} \theta_i * x_i$$

rewrite as matrices (for m observations):

$$y_i = \sum_{j=1}^{n} x_{ij} * \theta_j \qquad \mathbf{y} = \mathbf{X}\theta \qquad X_{m,n} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} \qquad \theta_n = \begin{pmatrix} \theta_1 & \theta_2 & \cdots & \theta_n \end{pmatrix}^T$$

$$y_m = \begin{pmatrix} y_1 & y_2 & \cdots & y_m \end{pmatrix}^T$$

Objective function:

$$\hat{\theta} = \arg\min_{\theta} J(\theta) = \arg\min_{\theta} \sum_{i=1}^{m} |y_i - \sum_{i=1}^{n} X_{ij}\theta_j|^2 = \arg\min_{\theta} ||\mathbf{y} - \mathbf{X}\theta||^2$$

Ordinary Least Squares (OLS) solution (with Moore-Penrose Pseudo Inverse):

$$\mathbf{y} = \mathbf{X}\theta$$

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

Linear Regression – Programming Example

1. Import Modules

The first step is always to import all the necessary libraries.
We need a library for matrix computation and a library to make fancy graphs

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

2. Define helper functions

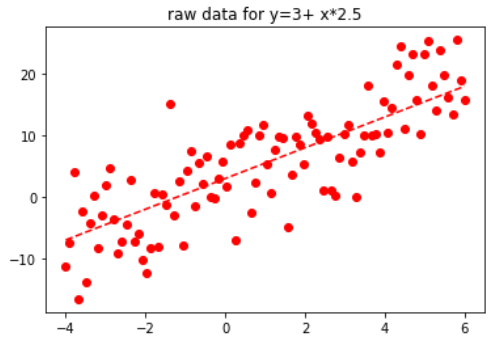
As good programmers, we like modularised code!
This means that we write little functions for each subtask of the regression problem :)

In [7]:
```
def generateData(thetas,x,sigma):
  # generate some data by adding gaussian noise (sigma) to a
  # linear function of x, parametrised by theta
  return linearFunction(x,thetas)+(sigma*np.random.randn(x.shape[0])+0)


def linearFunction(x,thetas):
  # defines a simple linear function - the line we're going to fit
  return thetas[0] + thetas[1]*x
```

3. Generate and visualise some toy data

Let's first generate some toy data and display it as scatter plot!

In [4]:
```
# define parameters
xRange   = [-4,6]
noiseVar = 5
thetas   = [3,2.5]

# generate data
x      = np.linspace(xRange[0],xRange[1],100)
y      = linearFunction(x,thetas)
yNoisy = generateData(thetas,x,noiseVar)

# display data
plt.plot(x,y,'--',color='red')
plt.plot(x,yNoisy,'o',color='red')
plt.xlabel = 'x'
plt.ylabel = 'y'
plt.title('raw data for y=' + str(thetas[0]) +'+ x*' + str(thetas[1]))
plt.show()
```



Linear Regression – Programming Example

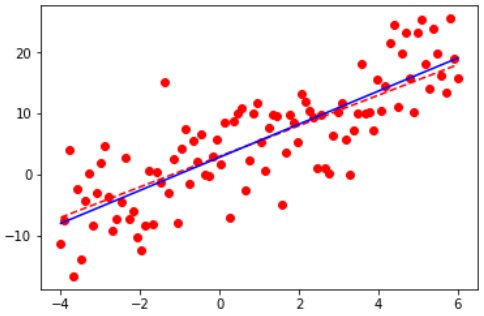
4. Compute the simple least squares solution

First we'll compute the ols solutions as outlined in the slides.

Remember that we just need to compute the mean, covariance and variance

In [5]:
```
def computeThetaHats(x,y):
  # computes the slope and intercept based on the mean, covariance
  # and variance terms
  thetaHat = np.empty(2)
  # 1. slope
  thetaHat[1] = np.cov(x,y)[0,1]/np.var(x)
  # 2. intercept
  thetaHat[0] = np.mean(y) - thetaHat[1]*np.mean(x)
  return thetaHat



# let's fit a line and print estimated parameters:
thetaHats = computeThetaHats(x,yNoisy)
yHat = linearFunction(x,thetaHats)

print('true thetas: ' + str(thetas))
print('theta hats: ' + str(thetaHats))

# plot the results:
plt.plot(x,y,'--',color='red')
plt.plot(x,yNoisy,'o',color='red')
plt.plot(x,yHat,'-',color='blue')
plt.xlabel = 'x'
plt.ylabel = 'y'
plt.show()
```

```
true thetas: [3, 2.5]
theta hats: [2.85037936 2.70498185]
```



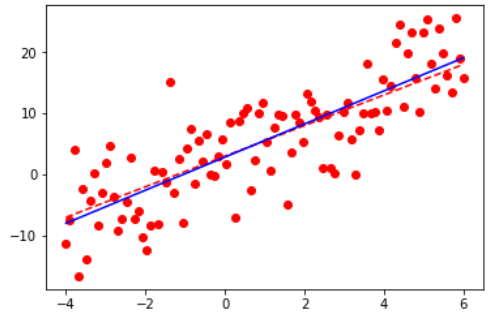
Linear Regression – Programming Example

4. Compute the simple least squares solution

First we'll compute the ols solutions as outlined in the slides.
Remember that we just need to compute the mean, covariance and variance

In [5]:
```
def computeThetaHats(x,y):
  # computes the slope and intercept based on the mean, covariance
  # and variance terms
  thetaHat = np.empty(2)
  # 1. slope
  thetaHat[1] = np.cov(x,y)[0,1]/np.var(x)
  # 2. intercept
  thetaHat[0] = np.mean(y) - thetaHat[1]*np.mean(x)
  return thetaHat



# let's fit a line and print estimated parameters:
thetaHats = computeThetaHats(x,yNoisy)
yHat = linearFunction(x,thetaHats)

print('true thetas: ' + str(thetas))
print('theta hats: ' + str(thetaHats))

# plot the results:
plt.plot(x,y,'--',color='red')
plt.plot(x,yNoisy,'o',color='red')
plt.plot(x,yHat,'-',color='blue')
plt.xlabel = 'x'
plt.ylabel = 'y'
plt.show()
```

```
true thetas: [3, 2.5]
theta hats: [2.85037936 2.70498185]
```

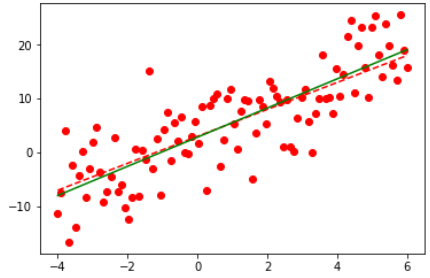


5. OLS solution

Now we generalise our solution into a matrix form that allows us to compute results for multivariable regression!
Remember: we just need to include a constant column to move the intercept inside the formula and solve for theta using the Moore-Penrose Pseudoinverse!

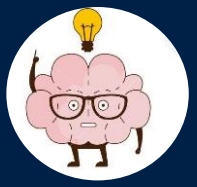
In [8]:
```
def computeOLSsolution(X,y):
  # computes the OLS solution for all thetas using matrix algebra
  return np.dot(np.linalg.inv(np.dot(X.T,X)),np.dot(X.T,y))

# set up the design matrix (add column of ones)
intercept = np.ones((len(x),1))
X = np.concatenate((intercept,np.expand_dims(x,axis=1)),axis=1)

# obtain thetaHats:
thetaHats = computeOLSsolution(X,yNoisy)
print('true thetas: ' + str(thetas))
print('theta hats: ' + str(thetaHats))

# display results
plt.plot(x,y,'--',color='red')
plt.plot(x,yNoisy,'o',color='red')
plt.plot(x,yHat,'-',color='green')
plt.xlabel = 'x'
plt.ylabel = 'y'
plt.show()
```

```
true thetas: [3, 2.5]
theta hats: [2.87742917 2.67793203]
```

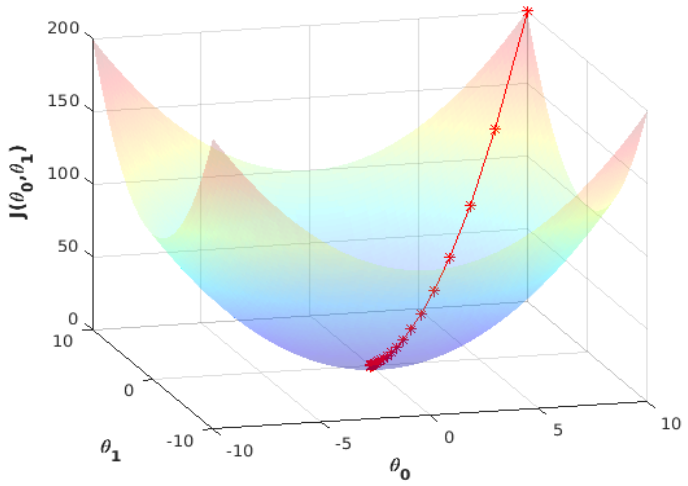


In [0]:

Linear Regression – Key Insights



- Linear Regression allows us to fit a straight line to continuous data
- It's easy to extend the formulation to multiple input variables
- Coding X_0 as vector of 1s and moving the bias term into the equation highly simplifies the maths.
- To minimise the objective function, we just need the derivatives of the linear term, which are the inputs, and the derivatives of the objective function, thanks to the chain rule of calculus!
- The maths is very easy (with a little bit of practice) and there exist an analytic solution (i.e. we can solve directly for theta)



Basics 2/3: Gradient Descent

Gradient Descent – Concept

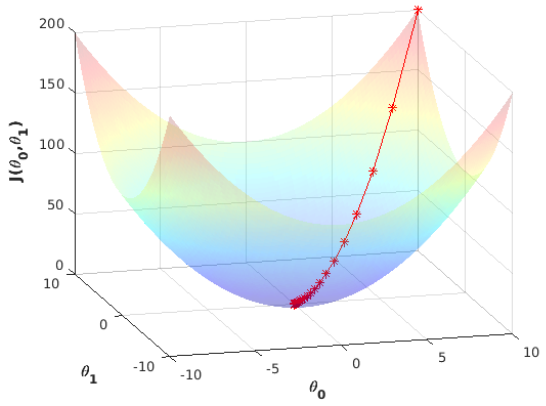
- The case of linear regression is very easy, as we can directly solve for theta and therefore obtain the best fitting parameters with one line of code!

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

- In many real world scenarios with more complex models (hint: neural networks), we can't solve directly for theta

- (Stochastic) Gradient Descent is an *iterative procedure* for finding the best fitting parameters of complex models

- The idea is simple: We incrementally change the value of the thetas in the direction that minimises the error, until we're not able to minimise the error function further (= "convergence")



Gradient Descent – What is a gradient?

- In the previous section, we've worked with partial derivatives, e.g. derivatives of functions with more than one variable. The variables were the parameters/weights/thetas we seeked to optimise.
- A gradient is simply the vector of all partial derivatives of a function, and is denoted by the *Nabla* or *Del* operator (inverted triangle):

$$f(\theta_1, \theta_2, \theta_3) = \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$$

$$\nabla f(\theta_1, \theta_2, \theta_3) = \left[\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \frac{\partial f}{\partial \theta_3}\right] = [x_1, x_2, x_3]$$

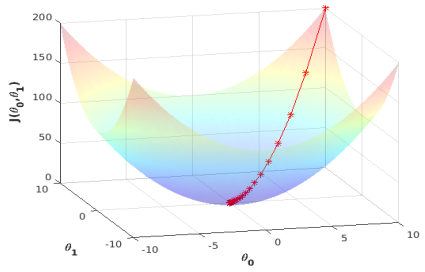
- You can think of this as vector that describes the rate of change of a function along multiple dimensions

Gradient Descent – Going Downhill!

- If the gradient describes the rate of change for a set of values of a function - and we seek to minimise this function - we just need to walk in the direction opposite of the gradient.
- This is basically the idea behind gradient descent!
- Mathematically, we do the following: On each iteration, we update the value of our parameter vector by subtracting the gradient, scaled by the "learning rate", which defines how big our steps are (here, theta is a vector of thetas):

$$\Delta\theta^t = \nabla f(\theta^t)$$
$$\theta^{t+1} = \theta^t - \epsilon * \Delta\theta^t$$

- Conceptually speaking, imagine you're trappend on a foggy mountain and trying to get down to the valley. The best thing to do is to "measure" the steepness of the terrain and slowly walk downhill, until you seem to be standing on a flat surface.



Gradient Descent – Terminology

- If you do this on your entire data set, it's called ***Gradient Descent***:

$$\theta^{t+1} = \theta^t - \epsilon * \nabla f(\theta^t) = \theta^t - \epsilon * \frac{1}{N} \sum_{i=1}^{N} \nabla f_i(\theta^t)$$

- ...if compute it on one sample (pair of $<x_i,y_i>$) per iteration (e.g. "online") , it's called ***Stochastic Gradient Descent***:

$$\theta^{t+1} = \theta^t - \epsilon * \nabla f_i(\theta^t)$$

- .. if you perform it on *minibatches* (randomly choosen set of m pairs, with m << n and n = size of dataset), it's usually also called Stochastic Gradient Descent, but sometimes ***Minibatch Gradient Descent***:

$$\theta^{t+1} = \theta^t - \epsilon * \frac{1}{M} \sum_{i=1}^{M} \nabla f_i(\theta^t)$$

Gradient Descent – 1D Example

- Let's have a look at a simple example. We'll try to find the minimum of a simple quadratic function:

$$f(\theta) = (\theta + 5)^2$$

- Of course, we don't even need Gradient Descent, as there is a simple analytic solution and we only need to apply the **chain rule** to find the derivative of after theta and set it to zero before solving for theta:

$$f(g) = g^2$$
$$g(\theta) = \theta + 5$$

$$\frac{df}{d\theta} = \frac{df}{dg} * \frac{dg}{d\theta}$$

$$\frac{df}{d\theta} = 2 * (\theta + 5) * 1 = 2\theta + 10$$

$$2\theta + 10 = 0$$

$$\hat{\theta} = -5$$

Gradient Descent – 1D Example

- Let's pretend for now that this solution doesn't exist. As the function takes only a single variable as input, the gradient is simply the first derivative after theta:

$$\nabla f(\theta) = \frac{df}{d\theta} = 2\theta + 10$$

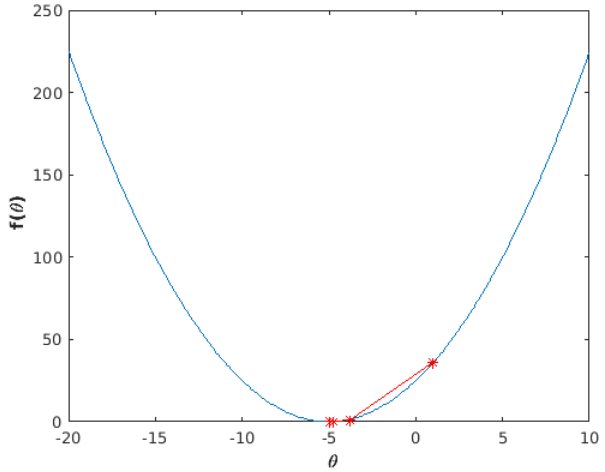
- Thus, our parameter update becomes:

$$\theta^{t+1} = \theta^t - \epsilon * \nabla(\theta^t) = \theta^t - \epsilon * (2\theta^t + 10)$$

- Let's have a look at a few iterations with epsilon = 0.4 and starting value for theta = 1

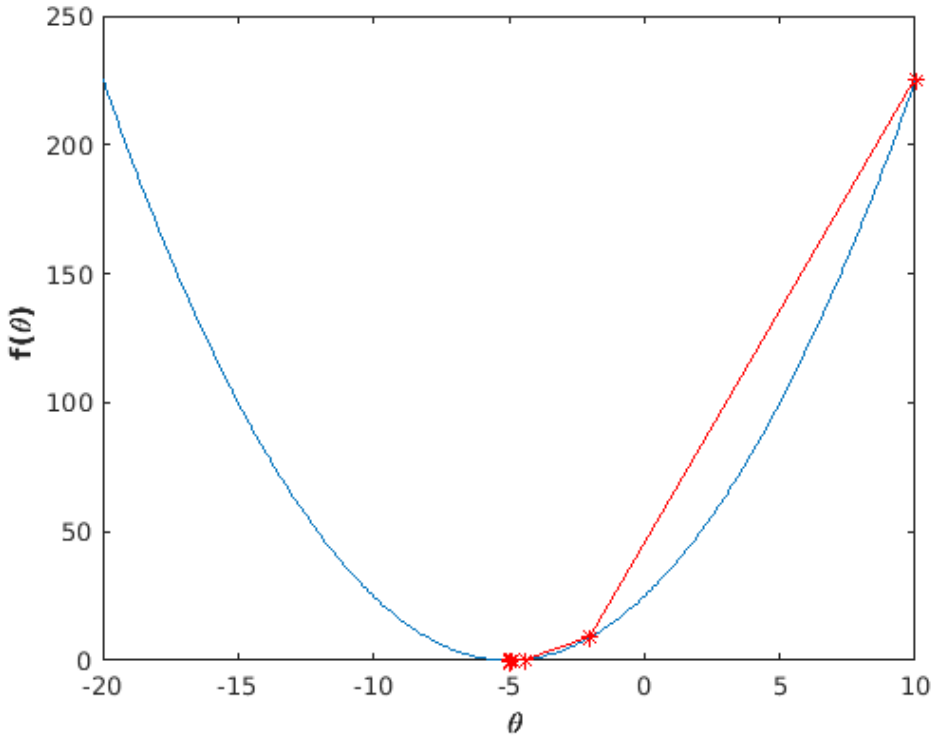
$$\epsilon = 0.4, \theta = 1$$

$$\theta^1 = 1$$
$$\theta^2 = 1 - 0.4 * (2 * 1 + 10) = -3.8$$
$$\theta^3 = -3.8 - 0.4 * (2 * -3.8 + 10) = -4.76$$
$$\theta^4 = -4.76 - 0.4 * (2 * -4.76 + 10) = -4.95$$
$$\theta^5 = -4.95 - 0.4 * (2 * -4.95 + 10) = -4.99$$



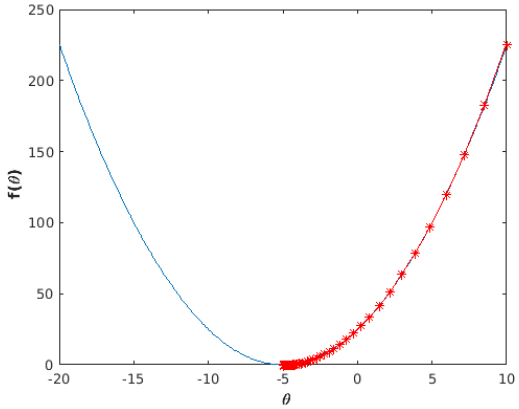
Gradient Descent – Effect of Learning Rate

- We were lucky, even with a very bad initial guess (e.g. theta=10), we would have reached convergence in less than 10 iterations:

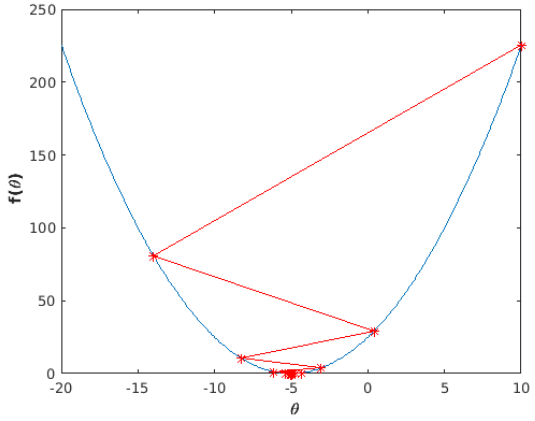


Gradient Descent – Effect of Learning Rate

- However, if the learning rate is too small, it takes ages for the algorithm to converge (eps=0.005):



- Likewise, too large of a learning rate leads to oscillations around the (local) minimum (eps=0.8):

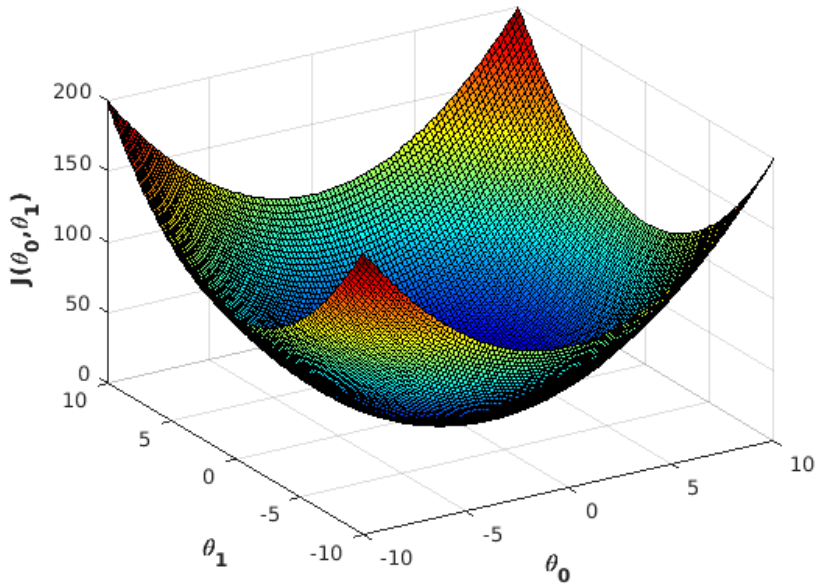


Gradient Descent – 2D Example

- Now let's consider a slightly more complex function, a 2D paraboloid:

$$J(\theta_0, \theta_1) = \theta_0^2 + \theta_1^2$$

- it looks like this:



- The gradient is now a 2D vector:

$$\nabla J(\theta_0, \theta_1) = \left\langle \frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1} \right\rangle = \langle 2\theta_0, 2\theta_1 \rangle$$

Gradient Descent – 2D Example

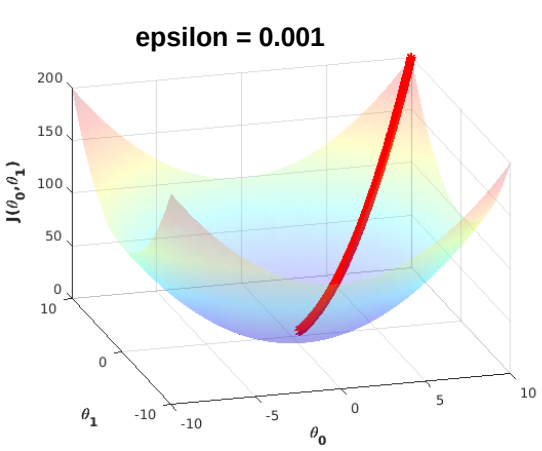
- Remember that theta is now a vector:

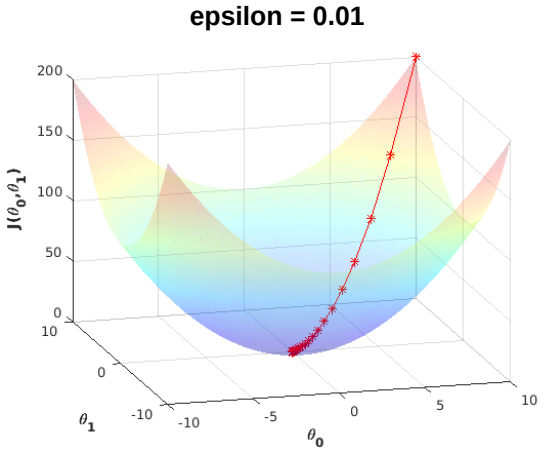
$$\boldsymbol{\theta} = (\theta_0, \theta_1)$$

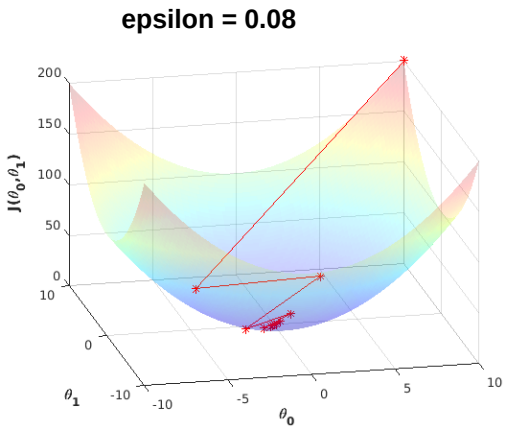
- The gradient update is:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \epsilon * \nabla J(\boldsymbol{\theta}^t)$$

- Which leads to the following gradient updates for various values of epsilon:







Gradient Descent – Programming Example

2. Define helper functions

let's define all necessary functions right away!

In [0]:

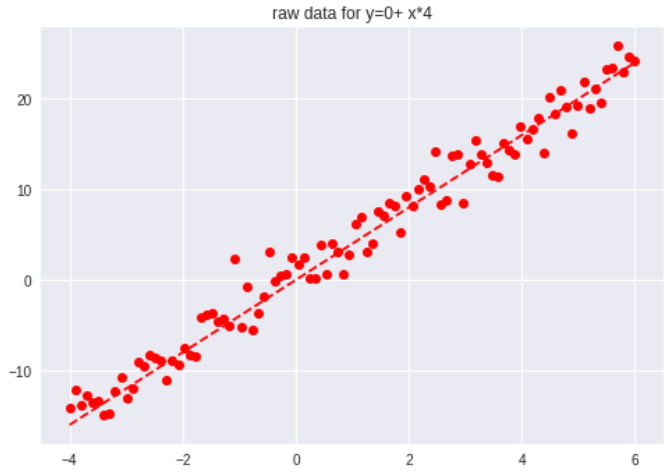
```
def generateData(thetas,x,sigma):
  # generate some data by adding gaussian noise (sigma) to a
  # linear function of x, parametrised by theta
  return linearFunction(x,thetas)+(sigma*np.random.randn(x.shape[0])+0)


def linearFunction(x,thetas):
  # defines a simple linear function - the line we're going to fit
  return thetas[0] + thetas[1]*x

def lossFunction(y_true,y_hat):
  # squared euclidean loss
  loss = np.mean([error**2 for error in (y_true-y_hat)])
  return loss

def lossGradient(x,y_true,y_hat):
  # compute the gradient of the loss function wrt to the weights
  # as vector of partial derivates (for intercept and slope)
  gradIntercept =   -2*np.mean((y_true-y_hat))
  gradSlope     = -2*np.mean(x*(y_true-y_hat))
  gradients = np.array([gradIntercept,gradSlope])
  return gradients

def runGD(x,y_true,thetas,epsilon,numIters):
  # performs gradient descent on data
  losses    = np.empty((numIters))
  thetaHats = np.empty((numIters,2))
  for ii in range(numIters):
    # get predictions with current parameter value
    y_hat = linearFunction(x,thetas)
    # store intermediate results:
    losses[ii] = lossFunction(y_true,y_hat)
    thetaHats[ii,:] = thetas
    # compute gradients (on whole dataset)
    gradients = lossGradient(x,y_true,y_hat)
    # update parameters
    thetas = thetas-epsilon*gradients

  return losses,thetaHats
```

Gradient Descent – Programming Example

3. Generate and visualise some toy data

Let's first generate some toy data and display it as scatter plot!

In [13]:
```
# define parameters
xRange   = [-4,6]
noiseVar = 2
thetas   = [0,4]

# generate data
x      = np.linspace(xRange[0],xRange[1],100)
y      = linearFunction(x,thetas)
yNoisy = generateData(thetas,x,noiseVar)

# display data
plt.plot(x,y,'--',color='red')
plt.plot(x,yNoisy,'o',color='red')
plt.xlabel = 'x'
plt.ylabel = 'y'
plt.title('raw data for y=' + str(thetas[0]) +'+ x*' + str(thetas[1]))
plt.show()
```



Gradient Descent – Programming Example

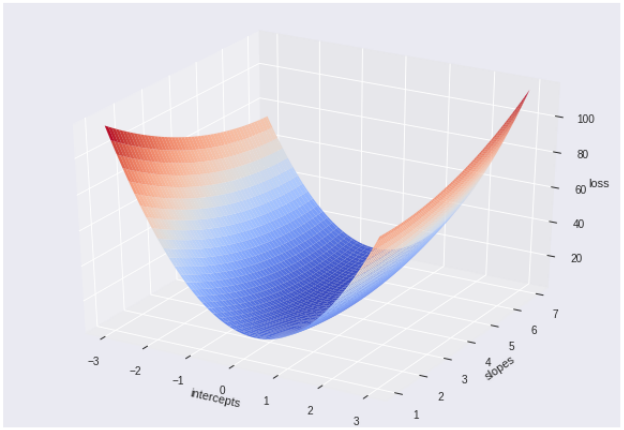
4. Explore the surface of the loss function

Next, we're going to plot the loss function of the linear model and the data above for different values of theta. This is a bit like "qualitative" exhaustive search - we display the loss for each combination of the slope and intercept parameters within a sensible range

In [0]:
```
intercepts = np.linspace(thetas[0]-3,thetas[0]+3,100)
slopes     = np.linspace(thetas[1]-3,thetas[1]+3,100)
[i,s]      = np.meshgrid(intercepts,slopes)
loss       = np.empty((100,100))
for ii,intercept in enumerate(intercepts):
  for ss,slope in enumerate(slopes):
    loss[ii,ss] = lossFunction(yNoisy,linearFunction(x,[intercept,slope]))
```

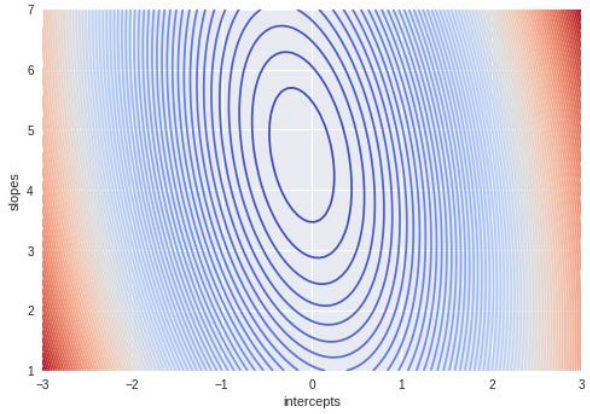
In [15]:
```
# 1. surface plot
fig = plt.figure()
ax = Axes3D(fig)
ax.plot_surface(i,s,loss,cmap=cm.coolwarm)
# ax.contour3D(i,s,loss,100,cmap=cm.coolwarm)
ax.set_xlabel('intercepts')
ax.set_ylabel('slopes')
ax.set_zlabel('loss')
```

Out[15]: `Text(0.5,0,'loss')`



In [16]:
```
# 2. contour plot
fig = plt.figure()
ax = fig.gca()
ax.contour(i,s,loss,100,cmap=cm.coolwarm)
ax.set_xlabel('intercepts')
ax.set_ylabel('slopes')
```

Out[16]: `Text(0,0.5,'slopes')`

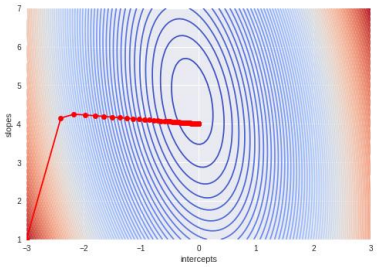


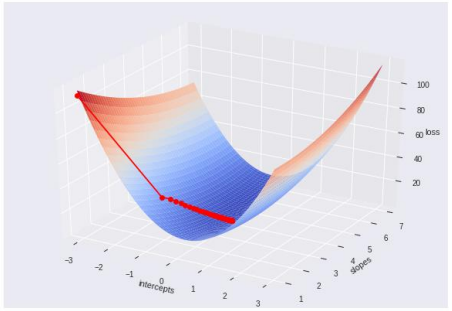
Gradient Descent – Programming Example

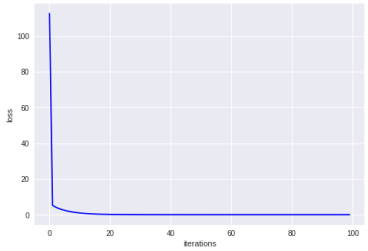
5. Perform Gradient Descent

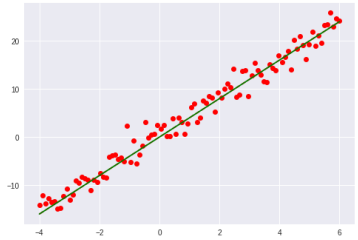
Ok let's perform gradient descent and plot the results!

In [17]:
```
# parameters
epsilon = 0.05
numIters = 100

# initial guesses for the thetas
# note: we usually use random initialisation, this here
# is for illustration purposes
thetas = np.array([-3,1])


# let's go!
lossVect = np.empty(numIters)
thetaVect = np.empty((numIters,2))
lossVect,thetaVect = runGD(x,y,thetas,epsilon,numIters)

# plot results (contour)
fig = plt.figure()
ax = fig.gca()
ax.contour(i,s,loss,100,cmap=cm.coolwarm)
ax.plot(thetaVect[:,0],thetaVect[:,1],'-o',color='red')
ax.set_xlabel('intercepts')
ax.set_ylabel('slopes')

# plot results (surface)
fig = plt.figure()
ax = Axes3D(fig)
ax.plot_surface(i,s,loss,cmap=cm.coolwarm)
ax.plot3D(thetaVect[:,0],thetaVect[:,1],lossVect,'-o',color='red')
ax.set_xlabel('intercepts')
ax.set_ylabel('slopes')
ax.set_zlabel('loss')

# plot loss function
fig = plt.figure()
ax = fig.gca()
ax.plot(np.arange(numIters),lossVect,'-',color='blue')
ax.set_xlabel('iterations')
ax.set_ylabel('loss')

# show fitted function
fig = plt.figure()
ax = fig.gca()
ax.plot(x,y,'--',color='red')
ax.plot(x,yNoisy,'o',color='red')
ax.plot(x,linearFunction(x,thetaVect[-1,:]),'-',color='green')
```

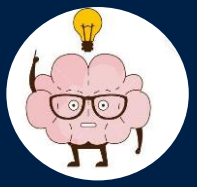




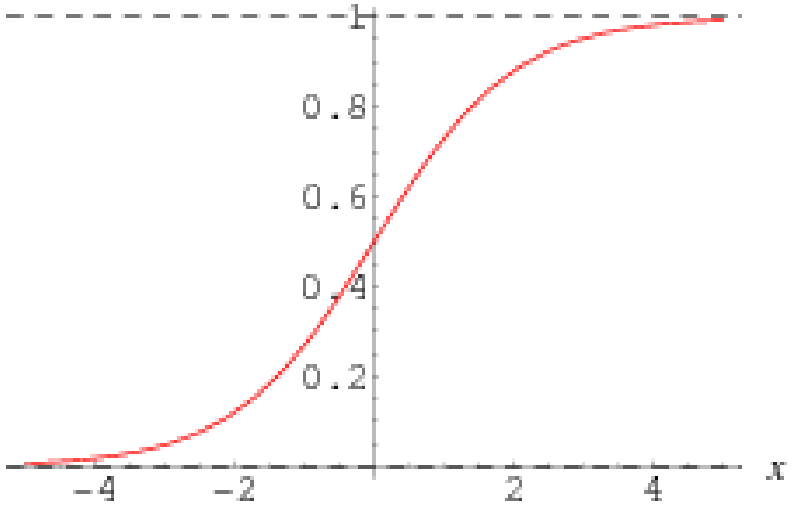




Gradient Descent – Key Insights



- Gradient Descent is an iterative procedure to find parameter values that minimise a differentiable function
- It is analogous to walking down a steep hill and measuring the steepness of the terrain until we've reached the valley (steepness equals zero)
- We can use it to find the parameter values that minimise the loss function of a regression.
- We don't need it for linear regression, as there are closed form solutions (OLS estimators)
- But we'll need it for more complex models, such as Logistic Regression and neural networks



Basics 3/3: Logistic Regression

Logistic Regression – Goal

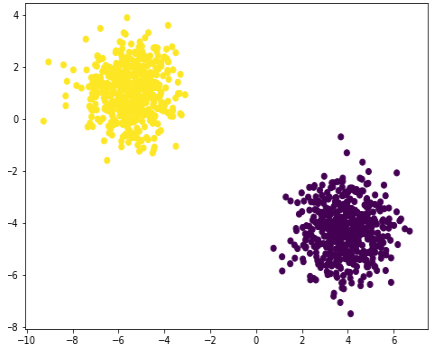
- Now consider a dataset of continuous x and binary y values, e.g. colour values of apples (as RGB triplets) and labels (e.g. „ripe“ or „not ripe“, coded as 0,1):

$$Data: \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N} \; with \; each \; \boldsymbol{x} \; as \; 1x3 \; vector, \; x_{ij} \in \mathbb{R} \; and \; y_i \in [0,1]$$

- Your task is to define a model that maps colour values onto binary labels

$$f(\boldsymbol{x}_i) = \hat{y}_i$$

- Linear regression wouldn‘t work, as the y‘s are no longer continuous
- This is a **classification problem**



Logistic Regression – Approach

- We continue with a „modular“ view on regression

- So far we've learned about two modules:
  - a ***linear function***

$$f_i(x_1, x_2, x_3, \dots, x_m) = \sum_{j=1}^{M} \theta_j * \ x_{ij}$$

  - and an ***objective/loss function***, for which we chose the so-called "**L2 loss**":

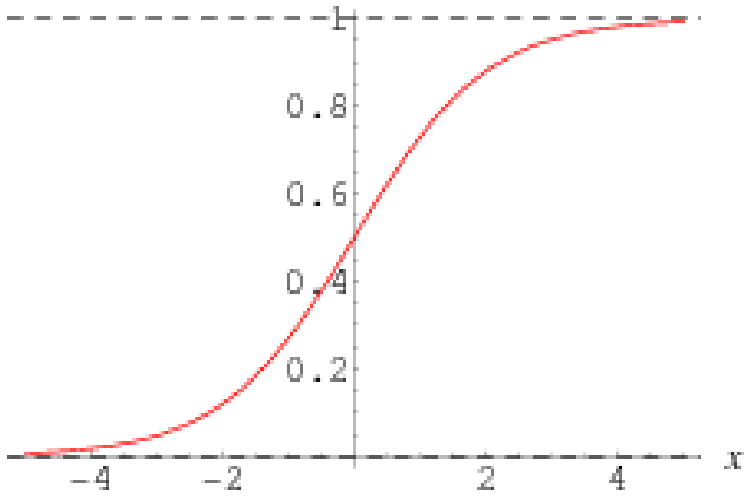
$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} (y_i - f(\boldsymbol{x}_i))^2$$

- To implement logistic regression, we need to
  a. transform the inputs into choice probabilities, using a ***nonlinear transducer function***.
  b. use a different ***objective function***.

Logistic Regression – Approach

We need a ***nonlinear transducer***. We use a sigmoidal function, which squashes outputs between 0 and 1:

$$\hat{y} = z(f) = \frac{1}{1+e^{-f}} = \frac{e^{f}}{e^{f}+1}$$



Our „neuron“ now consists of two modules: a linearity and a nonlinearity, which together map continuous inputs to choice probabilities:

Linearity: $f(x) = \theta * x$

Nonlinearity: $z(f) = \frac{1}{1+e^{-f}}$

Full Model: $z(\boldsymbol{\theta}, x) = \frac{1}{(1+e^{-(\theta * x)})}$

Logistic Regression – Approach

We also need a suitable ***objective function***. We use *cross entropy*, i.e. the number of bits needed to encode the true label y if we use our "wrong" function approximator z(f).

We assume a discrete distribution with c classes (="categories"):

$$H(y, \hat{y}) = -\sum_{c} y_c log(\hat{y}_c)$$

Note: If you have only two classes, such as in our example of binary logistic regression, you want to use this formula instead, which is the *binary cross entropy*:

$$H(y, \hat{y}) = -[ylog(\hat{y}) + (1-y)log(1-\hat{y})]$$

We can define the loss over all samples in the dataset, given the current parameters theta, as follows:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} H(y_i, \hat{y}_i) = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(\hat{y}_i) + (1-y_i)log(1-\hat{y}_i)]$$

$$= -\frac{1}{N}\sum_{i=1}^{N}[y_i log(z(\theta, x_i)) + (1-y_i)log(1-z(\theta, x_i))]$$

Logistic Regression – Solution

- Remember, the whole model is just a chain of a linearity and a nonlinearity:

$$\hat{y} = z(f(\boldsymbol{\theta}, \boldsymbol{x})) = \frac{1}{1+e^{-(\boldsymbol{\theta} * \boldsymbol{x})}}$$

- Remember, our loss function of theta is a function of the sigmoid above:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} H(y_i, \hat{y}_i) = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i)]$$

- Once again, to fit the model, we need to find the values for theta that minimise the loss:

$$\hat{\theta} = \arg\min_{\theta} J(\theta)$$

- Thus, we first need to compute the gradient of the loss wrt the parameters of the model:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}, \boldsymbol{y}, \boldsymbol{X}) = \nabla_{\boldsymbol{\theta}} J(z(f(\boldsymbol{\theta}, \boldsymbol{y}, \boldsymbol{X}))) = \nabla_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} H(z(f(\boldsymbol{\theta}, y_i, \boldsymbol{x_i}))) = \frac{1}{N}\sum_{i=1}^{N} \frac{dH(y_i, \hat{y}_i)}{d\hat{y}_i} * \frac{dz(f_i)}{df_i} * \nabla_{\theta} f_i(\boldsymbol{\theta}, \boldsymbol{x}_i)$$

$$= \frac{1}{N}\sum_{i=1}^{N} \frac{dH(y_i, \hat{y}_i)}{d\hat{y}_i} * \frac{dz(f_i)}{df_i} * \left[\frac{\partial f_i}{\partial \theta_1}, \frac{\partial f_i}{\partial \theta_2}, \ldots, \frac{\partial f_i}{\partial \theta_m}\right]$$

$$= \frac{1}{N}\sum_{i=1}^{N} \frac{dH(y_i, \hat{y}_i)}{d\hat{y}_i} * \frac{dz(f_i)}{df_i} * [x_{i1}, x_{i2}, \ldots, x_{im}]$$

$$= \frac{1}{N}\sum_{i=1}^{N} \left(-\frac{y_i}{z(f_i)} - \frac{1-y_i}{1-z(f_i)}\right) * z(f_i)(1-z(f_i)) * \boldsymbol{x_i}$$

$$= \ldots \text{magic} \ldots$$

$$= \frac{1}{N}\sum_{i=1}^{N} \left(z(f(\boldsymbol{\theta}, \boldsymbol{x}_i)) - y_i\right) * \boldsymbol{x_i}$$

Logistic Regression – Solution

- The gradient wrt theta has a nice interpretation: the derivative of the error surface is zero if the label and prediction are identical:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\left(z\left(f(\boldsymbol{\theta}, \boldsymbol{x}_i)\right) - y_i\right) * \boldsymbol{x_i}$$

- However, in contrast to linear regression, there exists no analytic solution to solve for theta.
- This means that we need numerical methods, such as Gradient Descent or IRLS, to estimate the parameters
- Today, we‘ll use what we‘ve learned so far, e.g. Gradient Descent!

$$\theta^{t+1} = \theta^t - \epsilon * \nabla_\theta J(\theta^t)$$

Logistic Regression – Programming Example

2. Define helper functions

we need

- binary data
- functions to plot results

In [193]:

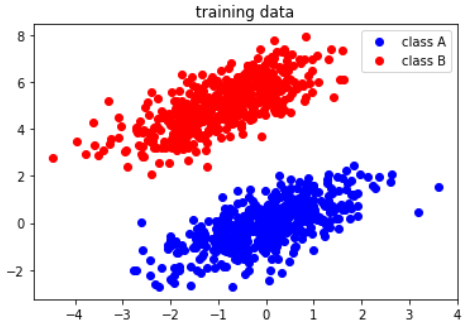
```
def generateData(numSamples=1000,mean1=[0,0],cov1=[[1,.7],[.7,1]],mean2=[-1,5],cov2=[[1,.7],[.7,1]]):
    # generate some data (two multivariate gaussians)
    x1, y1 = np.random.multivariate_normal(mean1, cov1, int(numSamples/2)).T
    x2, y2 = np.random.multivariate_normal(mean2, cov2, int(numSamples/2)).T

    x = np.concatenate((x1,x2))
    y = np.concatenate((y1,y2))

    xData = np.array([x,y]).T
    yData = np.concatenate((np.zeros(numSamples//2),np.ones(numSamples//2)))

    shuffIdces = np.random.permutation(numSamples)
    xData   = xData[shuffIdces,:]
    yData   = yData[shuffIdces]
    yData   = yData[:,np.newaxis]
    return xData,yData

def plotScatterData(x,y,legendStr=['class A','class B'],titleStr='data'):
    plt.figure()
    plt.plot(x[y[:,0]==0,0],x[y[:,0]==0,1],'o',color='blue')
    plt.plot(x[y[:,0]==1,0],x[y[:,0]==1,1],'o',color='red')
    plt.title(titleStr)
    plt.legend(legendStr)

def plotLossSurface(x,y,z,xLabel='theta1',yLabel='theta2',zLabel='Loss',titleStr='loss surface'):
    # 1. surface plot
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.plot_surface(x,y,z,cmap=cm.coolwarm)
    ax.set_xlabel(xLabel)
    ax.set_ylabel(yLabel)
    ax.set_zlabel(zLabel)
    ax.set_title(titleStr)

def plotLossTrajectory(x,y,z,th,lo,xLabel='theta1',yLabel='theta2',zLabel='Loss',titleStr='loss traject
ory'):
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.plot_surface(x,y,z,cmap=cm.coolwarm)
    ax.plot3D(th[:,1],th[:,2],lo,'-o',color='red')
    ax.set_xlabel(xLabel)
    ax.set_ylabel(yLabel)
    ax.set_zlabel(zLabel)
    ax.set_title(titleStr)

def plotLossCurve(x,y,titleStr='training loss'):
    fig = plt.figure()
    ax = fig.gca()
    ax.plot(x,y,'-',color='blue')
    ax.set_xlabel('iterations')
    ax.set_ylabel('loss')
    ax.set_title(titleStr)
```

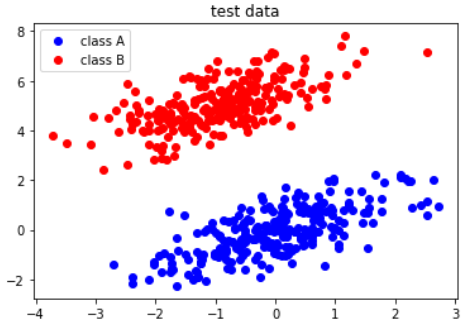
Logistic Regression – Programming Example

3. Generate and visualise some toy data

Let's first generate some toy data and display it as scatter plot!

In [185]:
```
# define parameters
nTrainingSamples = 1000
nTestSamples     =  500

# generate data
xTrain,yTrain = generateData(numSamples=nTrainingSamples)
xTest,yTest   =     generateData(numSamples=nTestSamples)

# display data
plotScatterData(xTrain,yTrain,titleStr='training data')
plotScatterData(xTest,yTest,titleStr='test data')

# add constant terms
xTrain = np.concatenate((np.ones((nTrainingSamples,1)),xTrain),axis=1)
xTest = np.concatenate((np.ones((nTestSamples,1)),xTest),axis=1)
```



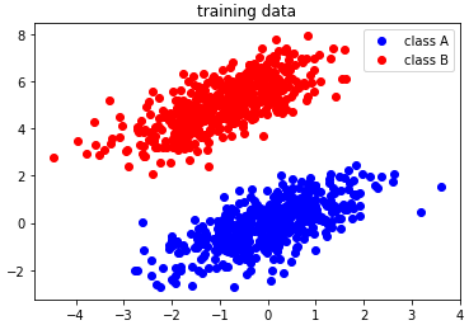


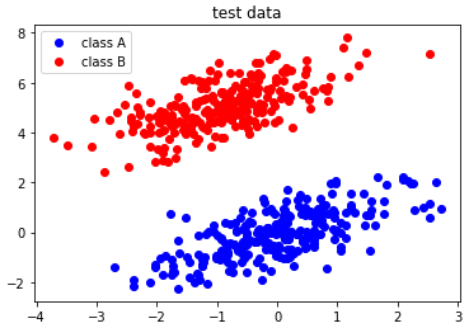
Logistic Regression – Programming Example

3. Generate and visualise some toy data

Let's first generate some toy data and display it as scatter plot!

In [185]:
```
# define parameters
nTrainingSamples = 1000
nTestSamples     =  500

# generate data
xTrain,yTrain = generateData(numSamples=nTrainingSamples)
xTest,yTest   =     generateData(numSamples=nTestSamples)

# display data
plotScatterData(xTrain,yTrain,titleStr='training data')
plotScatterData(xTest,yTest,titleStr='test data')

# add constant terms
xTrain = np.concatenate((np.ones((nTrainingSamples,1)),xTrain),axis=1)
xTest = np.concatenate((np.ones((nTestSamples,1)),xTest),axis=1)
```





4. Build the model

We'll take a modular approach, similar to how we'll later build a neural network

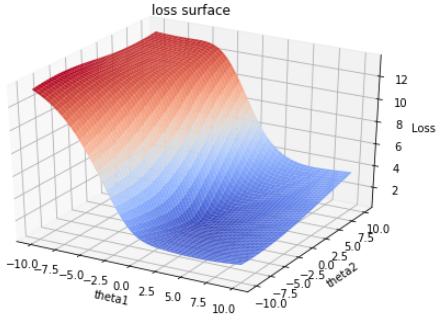
In [158]:
```
def linearFunction(x,thetas):
    # defines a simple linear function
    return np.dot(x,thetas.T)

def logisticFunction(x):
    # defines a logistic function
    return 1.0/(1.0+np.exp(-x))

def lossFunction(y_true,y_hat):
    # cross entropy loss
    y_true = y_true.reshape((-1,1))
    loss = -np.mean(y_true*np.log(y_hat+1e-10)+(1-y_true)*np.log(1-y_hat+1e-10))
    return loss

def lossGradient(x,y_true,y_hat):
    # compute the gradient of the loss function wrt to the weights
    gradients = np.mean((np.dot(x.T,(y_hat-y_true))),axis=1)
    return gradients

def regressionModel(x,thetas):
    z = linearFunction(x,thetas)
    y = logisticFunction(z)
    return y
```

5. Explore the error surface

Once again, we explore the error surface for various values of theta

In [163]:
```
theta1 = np.linspace(-10,10,50)
theta2 = np.linspace(-10,10,50)
[i,s]      = np.meshgrid(theta1,theta2)
loss       = np.empty((50,50))
for ii,t1 in enumerate(theta1):
    for jj,t2 in enumerate(theta2):
        loss[ii,jj] = lossFunction(yTrain,regressionModel(xTrain,np.array([[0,t1,t2]])))
```

In [187]:
```
plotLossSurface(i,s,loss)
```



Logistic Regression – Programming Example

6. Implement the training algorithm

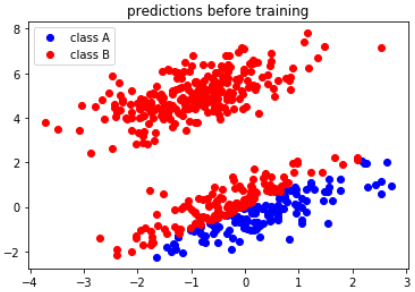
this implements gradient descent

In [134]:
```
def runGD(x,y_true,thetas,epsilon,numIters):
    # performs gradient descent on data
    losses    = np.empty((numIters))
    thetaHats = np.empty((numIters,3))
    for ii in range(numIters):
        # get predictions with current parameter value
        y_hat = regressionModel(x,thetas)
        # store intermediate results:
        losses[ii] = lossFunction(y_true,y_hat)
        thetaHats[ii,:] = thetas
        # compute gradients (on whole dataset)
        gradients = lossGradient(x,y_true,y_hat)

        # update parameters
        thetas = thetas-epsilon*gradients

    return losses,thetaHats
```

7. Train the model

Now we train the model

In [196]:
```
# parameters
epsilon = 1e-3
numIters = 1000
# initial guesses for the thetas
# note: we usually use random initialisation, this here
# is for illustration purposes
thetas = np.array([[0,-10,10]])


# show results for initialised function
yTest_pred = regressionModel(xTest,thetas) > 0.5

plotScatterData(xTest[:,1:],yTest_pred,titleStr='predictions before training')


# train the model
lossVect = np.empty(numIters)
lossVect,thetaVect = runGD(xTrain,yTrain,thetas,epsilon,numIters)
```



Logistic Regression – Programming Example

6. Implement the training algorithm

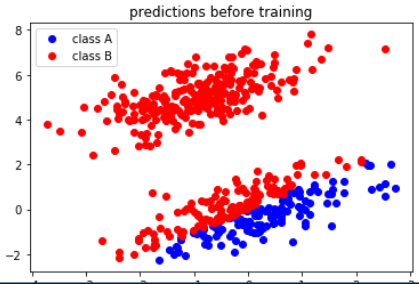
this implements gradient descent

In [134]:
```
def runGD(x,y_true,thetas,epsilon,numIters):
    # performs gradient descent on data
    losses    = np.empty((numIters))
    thetaHats = np.empty((numIters,3))
    for ii in range(numIters):
        # get predictions with current parameter value
        y_hat = regressionModel(x,thetas)
        # store intermediate results:
        losses[ii] = lossFunction(y_true,y_hat)
        thetaHats[ii,:] = thetas
        # compute gradients (on whole dataset)
        gradients = lossGradient(x,y_true,y_hat)

        # update parameters
        thetas = thetas-epsilon*gradients

    return losses,thetaHats
```

7. Train the model

Now we train the model

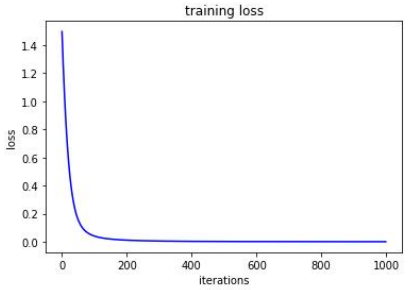
In [196]:
```
# parameters
epsilon = 1e-3
numIters = 1000
# initial guesses for the thetas
# note: we usually use random initialisation, this here
# is for illustration purposes
thetas = np.array([[0,-10,10]])


# show results for initialised function
yTest_pred = regressionModel(xTest,thetas) > 0.5

plotScatterData(xTest[:,1:],yTest_pred,titleStr='predictions before training')


# train the model
lossVect = np.empty(numIters)
lossVect,thetaVect = runGD(xTrain,yTrain,thetas,epsilon,numIters)
```

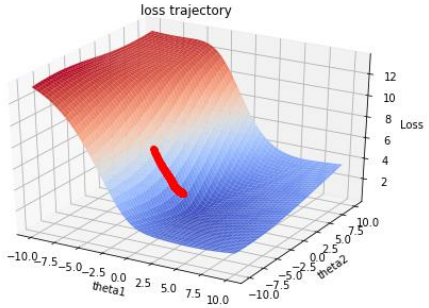


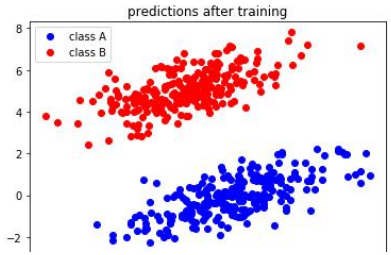
8. Evaluate the model

Let's have a look at the results

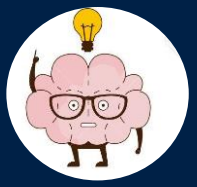
In [197]:
```
# plot loss function
plotLossCurve(np.arange(numIters),lossVect)

# plot gradient trajectory
plotLossTrajectory(i,s,loss,thetaVect,lossVect)

# show fitted function
yTest_pred = regressionModel(xTest,thetaVect[-1,:]) > 0.5
yTest_pred = yTest_pred[:,np.newaxis]
plotScatterData(xTest[:,1:],yTest_pred,titleStr='predictions after training')
```



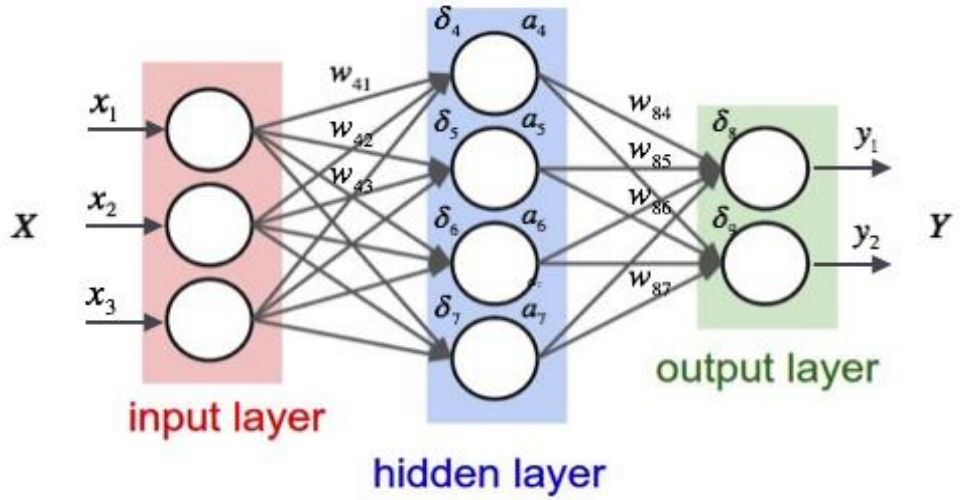




Logistic Regression – Key Insights



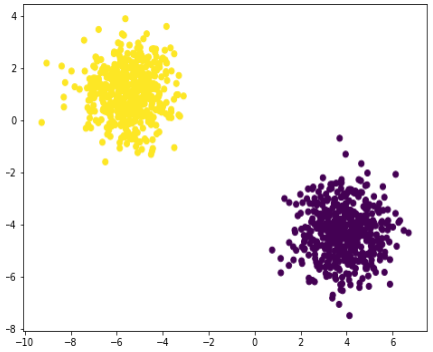
- Adding a sigmoidal nonlinearity allows us to solve binary classification problems!
- Adding a nonlinearity resulted in a much more complicated expression for the loss function
- we have to apply the chain rule to obtain the derivatives of the loss function wrt the weights of the linear term
- The final expression for the first derivative (e.g. gradient) of the loss function involves a nonlinearity (the sigmoid) which is a function of the weights.
- Furthermore, we sum up these terms over all the inputs.
- This means that - in contrast to linear regression - there is *no closed form / analytic solution* for the thetas.
- Thus, we have to apply a numerical procedure to compute the optimal weights, such as Gradient Descent!

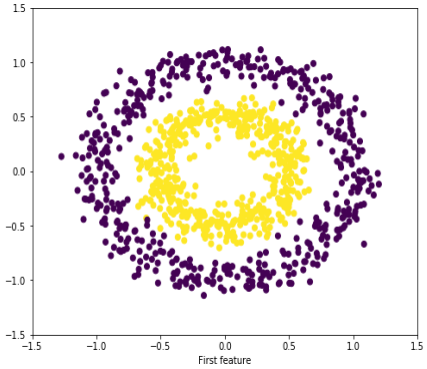


Neural Networks (yay!) ½ - Architecture

Can I Model the Brain with Logistic Regression?

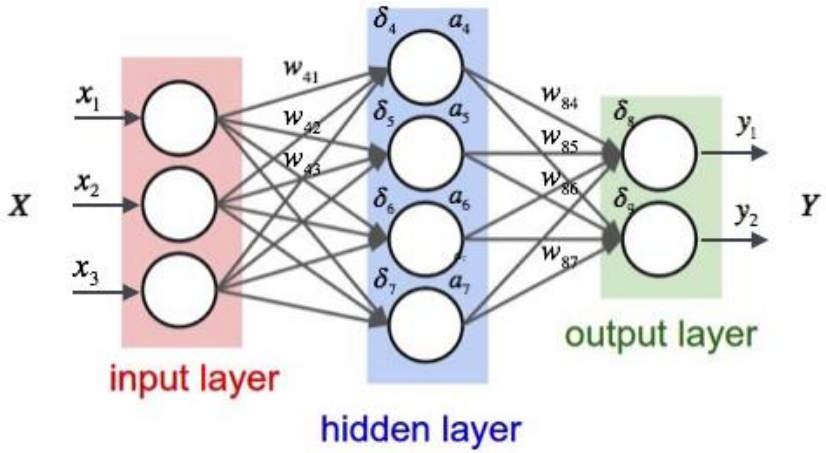
- For the banana-apple example, logistic regression might have been sufficient.
- But the real world is complex!
- We probably have more than one neuron for a reason!
- Even in our small world of toy datasets, one can easily think of a case where linear classifiers fail:





What is a Neural Network?

- It‘s a composition of many very simple functions!
- ..which allows us to **approximate** very complicated functions!
- It consists of **nodes** and **layers**. Each layer has several nodes, and layers are chained together.



y = sigmoid(linear(sigmoid(linear(sigmoid(linear(x))))))

What is a Neural Network?

- Several nodes together that receive the same input define a layer.
- This can be easily implemented as vector of inputs multiplied by matrix of weights, which - passed through a nonlinearity – yield a vector of outputs
- Several layers chained together define a neural network, yay!
- Even adding only one „hidden“ layer allows us to solve numerous problems

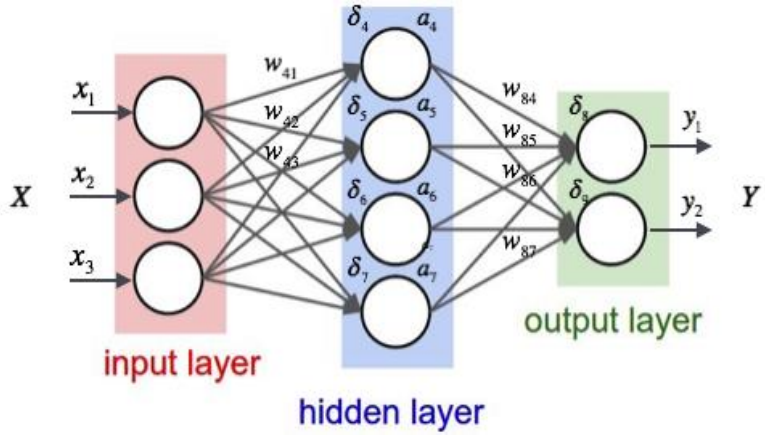
$$\boldsymbol{x}\ is\ 1x3, e.g. [x_1, x_2, x_3]$$

$$\boldsymbol{W}_{ax}\ is\ 3x4, e.g. \begin{bmatrix} w_{11} & \cdots & w_{14} \\ \vdots & \ddots & \vdots \\ w_{31} & \cdots & w_{34} \end{bmatrix}$$

$$\boldsymbol{a}\ is\ 1x4, e.g. [a_1, a_2, a_3, a_4]$$

$$\boldsymbol{W}_{ya}\ is\ 4x2, e.g. \begin{bmatrix} w_{11} & w_{12} \\ \vdots & \vdots \\ w_{41} & w_{42} \end{bmatrix}$$

$$\boldsymbol{y}\ is\ 2x1, e.g. [y_1, y_2]^T$$

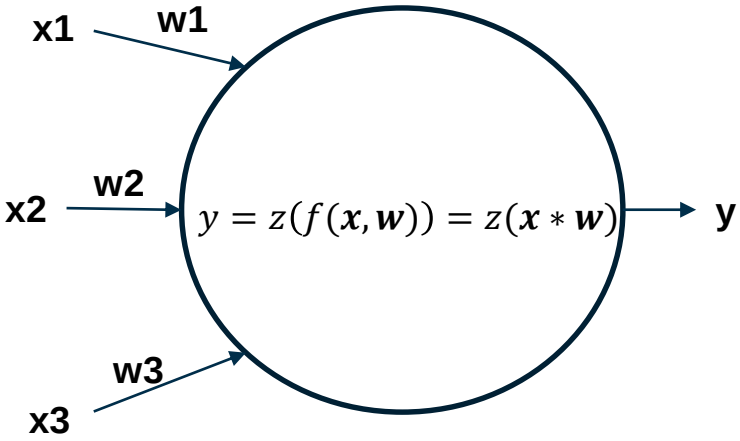


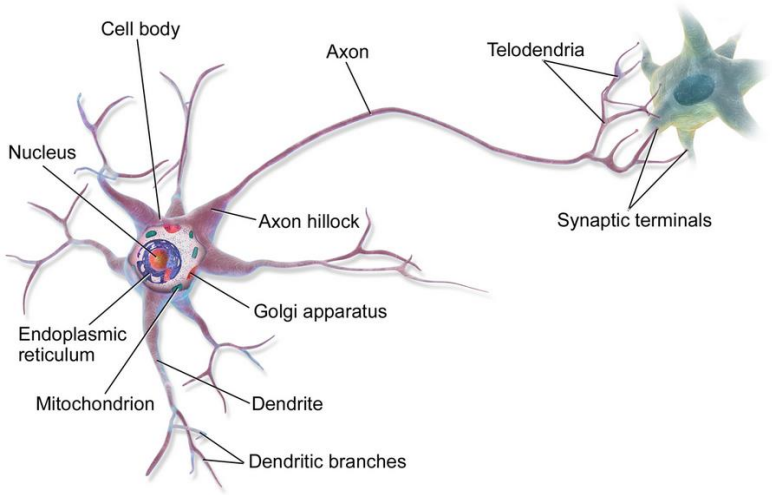
$$\mathbf{y} = \mathrm{z}\left(\mathbf{a} * \mathbf{W}_{\mathbf{ya}}\right) = \mathrm{z}(\mathrm{z}(\mathbf{x} * \mathbf{W}_{\mathrm{ax}}) * \mathbf{W}_{\mathrm{ya}})$$

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

What is a Node?

- A node consists of a linearity (a weighted input), and a nonlinearity, which transforms the weighted input.
- The weights are the network parameter (i.e. our thetas) that we wish to train
- A node probably reminds you of the logistic regression example
- ...or of a neuron (x*w are dendrites, y the spike train, perhaps?)

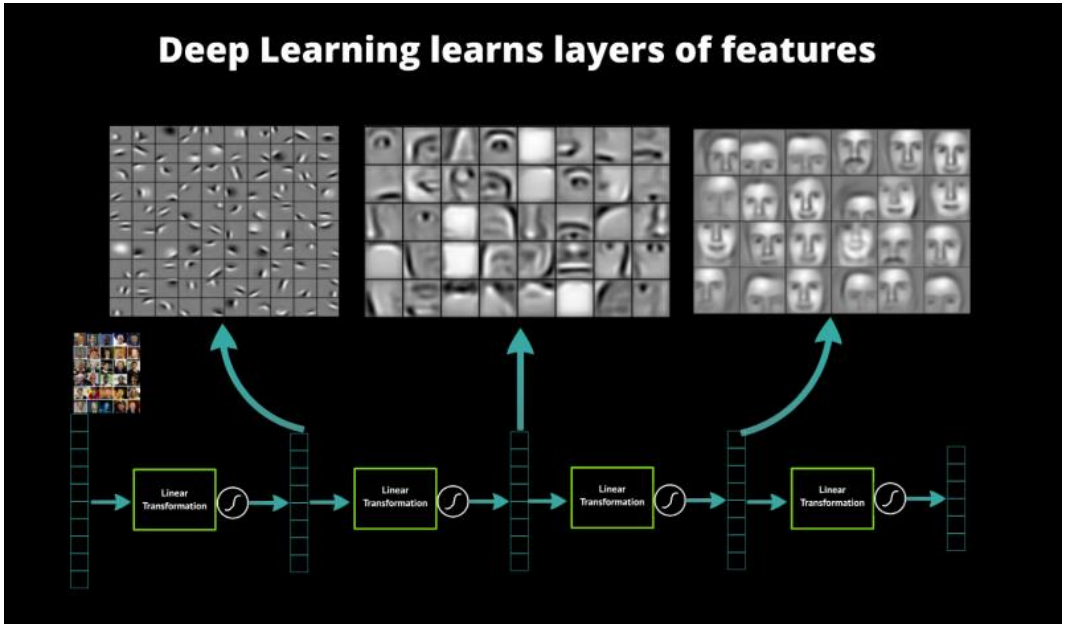




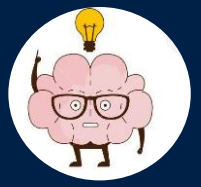
$$\hat{y} = z(f) = \frac{1}{1 + e^{-f}} = \frac{1}{1 + e^{-\boldsymbol{x}*\boldsymbol{w}}}$$

Is One Hidden Layer Enough?

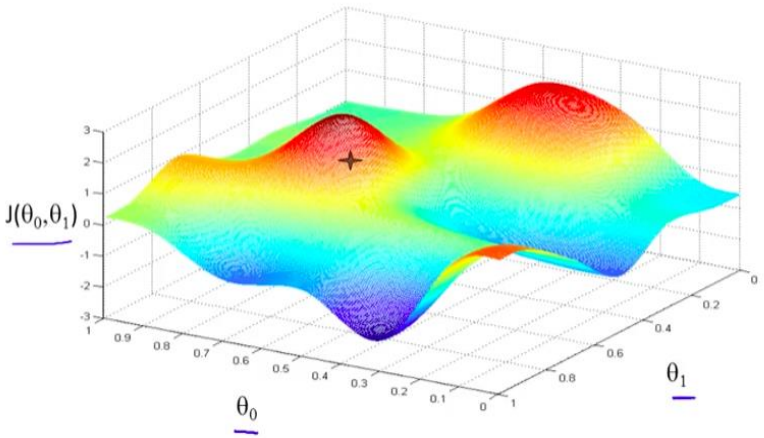
- Although networks with one hidden layer can learn complex functions, would need a lot of hidden units
- Adding more layers more powerful than adding more hidden units (depth over breadth)
- Allows the complex functional mapping to be broken into a series/hierarchy of intermediate representations
    - i.e. hidden layers can learn useful features (early layers simpler features, later layers more complex ones)
    - "Representation Learning“

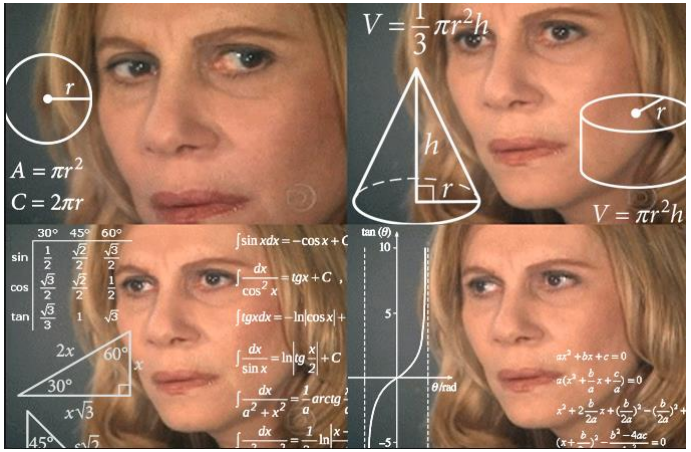


Neural Network Architecture – Key Insights



- Neural Networks are function approximators, loosely inspired by the brain
- We can think of them as compositions of many simple linear and nonlinear functions
- Training a Deep Network ~ Learning useful Representations of data





Neural Networks 2/2 – Parameter Estimation

Neural Network – Goal

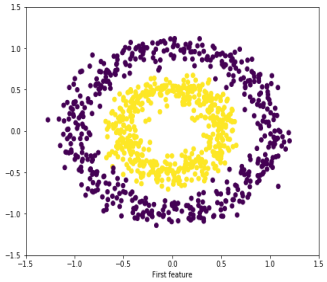
- Once again, we have a dataset of real-valued inputs and binary labels

$$Data: \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N} \text{ with each } \boldsymbol{x} \text{ as } 1x2 \text{ vector}, x_{ij} \in \mathbb{R} \text{ and } y_i \in [0,1]$$

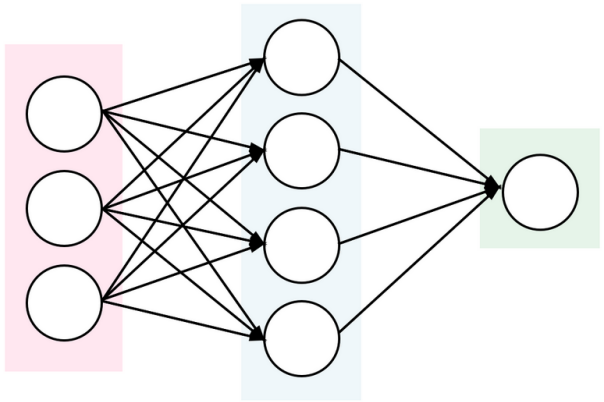
- Again, we're interested in the function that maps inputs to labels

$$f(x_i) = \hat{y}_i$$

- This time, we assume that the data is not linearly separable, i.e. Logistic Regression wouldn't work



- We therefore construct a simple neural network with one hidden layer:



$\boldsymbol{x}$ *is* $1x3, e.g. [x_1, x_2, x_3]$

$$\boldsymbol{W}_{hx} \text{ is } 3x4, e.g. \begin{bmatrix} w_{11} & \cdots & w_{14} \\ \vdots & \ddots & \vdots \\ w_{31} & \cdots & w_{34} \end{bmatrix}$$

$\boldsymbol{h}$ *is* $1x4, e.g. [a_1, a_2, a_3, a_4]$

$\boldsymbol{w}_{yh}$ *is* $4x1, e.g. [w_{11}, w_{21}, w_{31}, w_{41}]^T$

y *is a scalar*

Neural Network - Approach

- We use sigmoids (as for logistic regression) as nonlinearities/activation functions.

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

- Once again, the model is a composition of functions, but this time obv. with several nonlinearities:

$$y = f(x) = f\left(z\big(z(x * W_{hx}) * w_{\text{yh}}\big)\right) = f\left(z\left(l\big(z\big(l(x, W_{hx})\big), w_{yh}\big)\right)\right)$$

- Again, we use binary cross-entropy as loss function

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} H(y_i, \hat{y}_i) = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)]$$

- Remember: We want to find the values for our parameters that minimise the loss

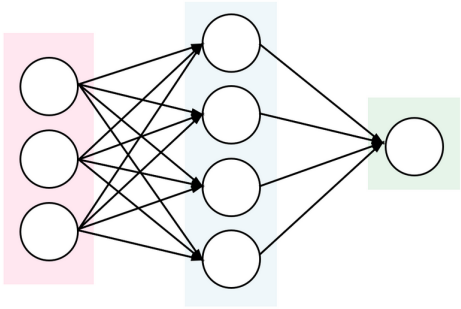
$$\widehat{W}_{hx}, \widehat{w}_{yh} = \arg\min_{W_{hx}, w_{yh}} J(W_{hx}, w_{yh})$$

- Easy peasy: Compute all the particial derivatives and use gradient descent. Except that this doesn‘t work/scale nicely!

- For a network with one hidden layer, we need to compute

$$\frac{\partial J}{\partial w_{yh}} = \frac{dH(y,\hat{y})}{d\hat{y}} * \frac{dz_y(l_y)}{dl_y} * \nabla_{\boldsymbol{w_{yh}}} l_y(\boldsymbol{w}_{yh}, \boldsymbol{h})$$

$$\frac{\partial J}{\partial W_{hx}} = \frac{dH(y,\hat{y})}{d\hat{y}} * \frac{dz_y(l_y)}{dl_y} * \frac{dl_y(z_h)}{dz_h} * \frac{dz_h(l_h)}{dl_h} * \boldsymbol{J}_{\boldsymbol{W}_{hx}} l_h(\boldsymbol{W}_{hx}, \boldsymbol{x})$$



Neural Network - Solution

- This scales very badly! Imagine we'd be after the derivative of the loss wrt to the first-layer weights in a network with ten layers:

$$\frac{\partial J}{\partial W_1} = \frac{dH(y,\hat{y})}{d\hat{y}} * \frac{dz_{10}(l_{10})}{dl_{10}} * \frac{dl_{10}(z_9)}{dz_9} * \frac{dz_9(l_9)}{dl_9} * \frac{dl_9(z_8)}{dz_8} * \frac{dz_8(l_8)}{dl_8} * \dots * \frac{dz_1(l_1)}{dl_1} * \frac{dl_1(w_1)}{dw_1}$$

- You'll also note how redundant this procedure is: To compute the hidden-layer weight derivatives, we need the same formulas as for the output-layer weight derivatives:

$$\frac{\partial J}{\partial w_{yh}} = \frac{dH(y,\hat{y})}{d\hat{y}} * \frac{dz_y(l_y)}{dl_y} * \nabla_{\boldsymbol{w_{yh}}} l_y(\boldsymbol{w}_{yh}, \boldsymbol{h})$$

$$\frac{\partial J}{\partial W_{hx}} = \frac{dH(y,\hat{y})}{d\hat{y}} * \frac{dz_y(l_y)}{dl_y} * \frac{dl_y(z_h)}{dz_h} * \frac{dz_h(l_h)}{dl_h} * \boldsymbol{J}_{\boldsymbol{W}_{hx}} l_h(\boldsymbol{W}_{hx}, \boldsymbol{x})$$

- This is where the **Backpropagation Algorithm** comes into play
- It's a clever method to compute gradients whilst avoiding redundancy!

Method:
Forward pass: compute the layer-wise outputs for a given input, until you reach the loss
Backward pass:

1. Start at the output node, compute the error signal $\frac{dH(y,\hat{y})}{d\hat{y}} * \frac{dz_y(l_y)}{dl_y}$ as well as the gradient wrt to the weights
2. Move to the preceding node, carry the previously computed error signal with you, multiply it with the derivatives of the current node wrt to the linearity and the inputs to get a new error signal. Compute the gradient
3. Rinse and repeat until you reach the input node.

Neural Network – Solution: Backpropagation of Errors

These are all the derivatives we need:

- Derivative of Loss wrt to its inputs: $\frac{dH(y,\hat{y})}{d\hat{y}} = \left(-\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right)$
- Derivative of Nonlinearity (here: Sigmoid) wrt to its inputs: $\frac{dz(l)}{dl} = z(l) * (1 - z(l))$
- Derivative of Linearity wrt its inputs: $\frac{dl(x)}{dx} = \boldsymbol{w}$
- Derivative of Linearity wrt its weights (for the gradient update!): $\frac{dl(w)}{dw} = \boldsymbol{x}$

Now we can compute:

- The error in the output layer: $\mathrm{E_y} = \frac{dH(y,\hat{y})}{d\hat{y}} * \frac{dz_y(l_y)}{dl_y}$
- The gradient in the output layer: $\nabla_{w_{yh}} = \mathrm{E_y} * \frac{dl_y(w_{yh})}{dw_{yh}}$
- The error in the hidden layer: $\mathrm{E_h} = \mathrm{E_y} * \frac{dl_y(z_h)}{dz_h} * \frac{dz_h(l_h)}{dl_h}$
- The gradient in the hidden layer: $\boldsymbol{J_{W_{hx}}} = \mathrm{E_y} * E_h * \frac{dl_h(W_{hx})}{dW_{hx}}$

… And update the weights efficiently:

$$w_{yh}^t = w_{yh}^t - \epsilon * \nabla_{w_{yh}^{t-1}}$$
$$W_{hx}^t = W_{hx}^t - \epsilon * J_{W_{hx}^{t-1}}$$

Neural Network – Programming Demonstration

1. Import Modules

Once again, let's import a few modules

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib           import cm
from mpl_toolkits.mplot3d import axes3d, Axes3D
```

2. Define helper functions

we need

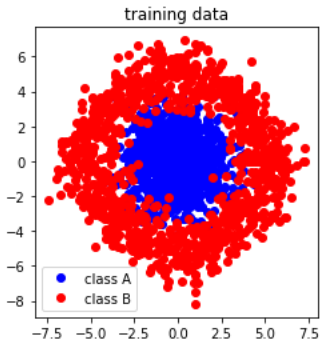
- binary data
- functions to plot results

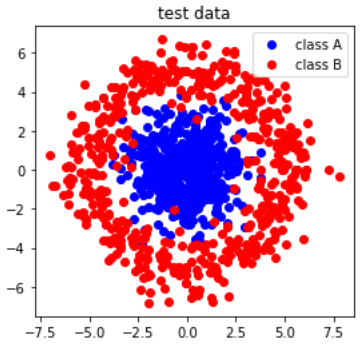
In [225]:
```
def generateData(v1=2,v2=1,numSamples=1000):
    # generates data set for binary classification that can't be separated
    # by a linear boundary in 2D
    samplesPerClass = numSamples//2
    numAngles = 100
    samplesPerAngle = samplesPerClass//numAngles

    # isotropic multivariate gaussian
    cat1 = np.random.multivariate_normal([0,0],[[v1,0],[0,v1]],numSamples//2)
    phis = np.linspace(0,2.0*np.pi,numAngles)
    phis = phis[:,np.newaxis]
    # donut (circle with univariate gaussian noise)
    cat2 = np.array([])
    for ii,phi in enumerate(phis):
        newSample = np.array([(v2*np.random.randn(samplesPerAngle)+5)*np.cos(phi),(v2*np.random.randn(s
amplesPerAngle)+5)*np.sin(phi)]).T
        cat2 = np.vstack((cat2,newSample)) if len(cat2)!=0 else newSample

    xData = np.vstack((cat1,cat2))
    yData = np.vstack((np.zeros((numSamples//2,1)),np.ones((numSamples//2,1))))
    shuffIdces = np.random.permutation(numSamples)
    xData   = xData[shuffIdces,:]
    yData   = yData[shuffIdces,:]
    return xData,yData
```

3. Generate and visualise some toy data

Let's first generate some toy data and display it as scatter plot!

|:
```
# define parameters
nTrainingSamples = 2000
nTestSamples     = 1000

# generate data
xTrain,yTrain = generateData(numSamples=nTrainingSamples)
xTest,yTest   =     generateData(numSamples=nTestSamples)


# display data
plotScatterData(xTrain,yTrain,titleStr='training data')
plotScatterData(xTest,yTest,titleStr='test data')

# add constant terms
xTrain = np.concatenate((np.ones((nTrainingSamples,1)),xTrain),axis=1)
xTest = np.concatenate((np.ones((nTestSamples,1)),xTest),axis=1)
```





Neural Network – Programming Demonstration

4. Build the model

We'll define the neural network as object

In [227]:

```
class NeuralNetwork():
    def __init__(self,numInputs,numHidden,numOutputs):
        self.numInputs  = numInputs
        self.numHidden  = numHidden
        self.numOutputs = numOutputs

        self.W_h = np.random.randn(self.numInputs,self.numHidden)
        self.W_o = np.random.randn(self.numHidden,self.numOutputs)


    def linearFunction(self,x,thetas):
        # defines a simple linear function
        return np.dot(x,thetas)

    def logisticFunction(self,x):
        # defines a logistic function
        return 1.0/(1.0+np.exp(-x))

    def lossFunction(self,y_true,y_hat):
        # cross entropy loss
        y_true = y_true.reshape((-1,1))
        loss = -np.mean(y_true*np.log(y_hat+1e-10)+(1-y_true)*np.log(1-y_hat+1e-10))
        return loss

    def fprop(self,x):
        # this implements forward-propagation
        # returns layer activations

        # input to hidden layer:
        h_l = self.linearFunction(x,self.W_h)
        h_z = self.logisticFunction(h_l)

        # hidden layer to output layer:
        o_l = self.linearFunction(h_z,self.W_o)
        o_z = self.logisticFunction(o_l)

        layerActivations = {"hidden_lin": h_l,
                            "hidden_sig":h_z,
                            "out_lin": o_l,
                            "out_sig":o_z}

        return layerActivations
```

Neural Network – Programming Demonstration

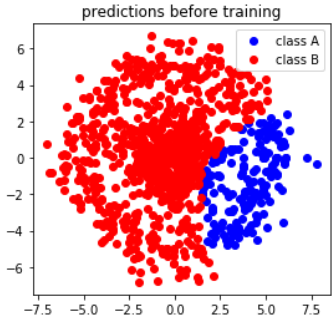
```
def bprop(self,x,y,layerActivations):
    # we called this "lossGradient" in prev
    # example.
    # this basically implements backpropagation

    # derivative of loss wrt its input
    dO_c = ((1-y)/(1-layerActivations["out_sig"]))-(y/layerActivations["out_sig"])
    #derivative of sigmoid wrt its input
    dO_z = layerActivations["out_sig"]*(1-layerActivations["out_sig"])
    # total error of output layer:
    Ey = dO_c*dO_z
    # gradient in output layer (wrt its weights)
    dO_w = np.dot(Ey.T,layerActivations["hidden_sig"])

    # total error of output layer wrt its input (the quantity we propagate to prev layers):
    dO_l = np.dot(Ey,self.W_o.T)


    # derivate of sigmoid in hidden wrt its input
    dH_z = layerActivations["hidden_sig"]*(1-layerActivations["hidden_sig"])
    # error in hidden layer:
    Eh = dO_l*dH_z
    # gradient in hidden layer (wrt its weights:
    dH_w = np.dot(Eh.T,x)


    # total error wrt input of hidden layer
    dH_l = np.dot(Eh,self.W_h.T)

    # dictionary of gradients
    gradients = {'grad_out': dO_w,
                 'grad_hidden':dH_w}
    return gradients


def runSGD(self,x,y,numIters=2,lrate=0.03):
    # performs stochastic gradient descent
    losses = np.empty((numIters))
    for ii in range(numIters):
        outs = self.fprop(x)
        losses[ii] = self.lossFunction(y,outs['out_sig'])
        gradients = self.bprop(x,y,outs)
        # update weights:
        self.W_o = self.W_o-lrate*gradients['grad_out'].T
        self.W_h = self.W_h-lrate*gradients['grad_hidden'].T
    return losses

def predict(self,x):
    return self.fprop(x)['out_sig']
```

Neural Network – Programming Demonstration

7. Train the model

Now we train the model

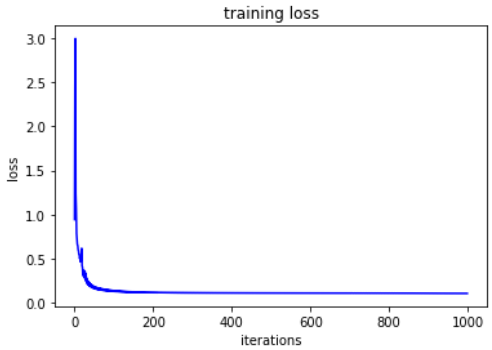
In [228]:
```
# parameters:
numIters = 1000
epsilon = 0.005

# initialise the model
nnet = NeuralNetwork(3,10,1)

# show results for initialised (i.e. untrained) function
yTest_pred = nnet.predict(xTest) > 0.5

plotScatterData(xTest[:,1:],yTest_pred,titleStr='predictions before training')

# train the model
loss = nnet.runSGD(xTrain,yTrain,numIters=numIters,lrate=epsilon)
yTest_pred = nnet.predict(xTest) > 0.5
```

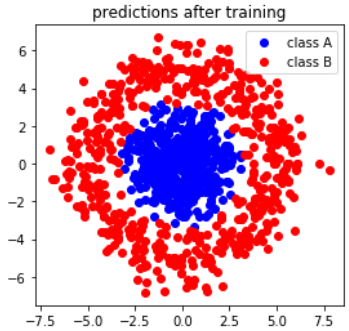


8. Evaluate the model

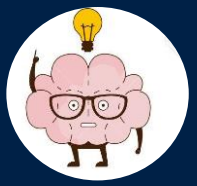
Let's have a look at the results

In [229]:
```
# plot loss function
plotLossCurve(np.arange(1000),loss)


# show fitted function
plotScatterData(xTest[:,1:],yTest_pred,titleStr='predictions after training')
```





Summary



Conceptual:

- Neural Networks are function approximators, loosely inspired by the brain
- We can think of them as compositions of many simple linear and nonlinear functions
- Training a Deep Network ~ Learning useful Representations of data
- Training neural networks on the same tasks as humans and comparing the emerging representations in network layers and the brain might reveal how the brain processes statistical information about the environment

Technical:

- Gradient Descent: Way to learn the optimal parameters by gradually changing them in a way that maximally decreases the loss (i.e. in the direction of the gradient of the loss w.r.t. the parameters)

- Backpropagation: makes use of modularity & the chain rule to compute the gradient in an efficient way by re-using previously computed gradients

Thanks!