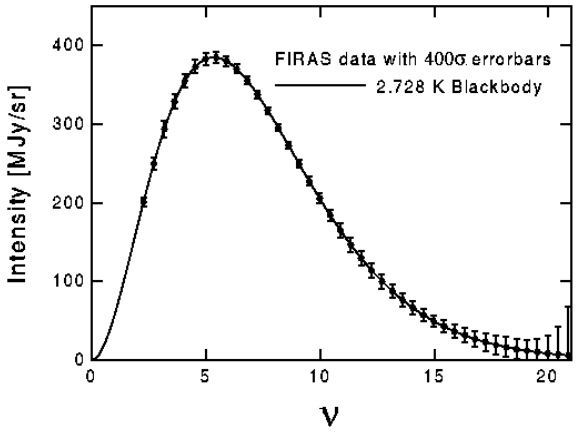
Inference and neural networks

Inference

Given some measurements, what can we learn about the world? Often we **fit a theoretical model** to data to see (a) if the model is valid and (b) constrain any parameters of the model.



Linear regression and chi-squared

Given N data points y_i sampled at points x_i with errors σ_i, find "best" values of m and c in a straight-line model Y:

$$Y_i = mx_i + c$$

We often use a goodness of fit statistic called **chi-squared**:

$$\chi^2(m,c) = \sum_{i=0}^{N-1} \left(\frac{y_i - mx_i - c}{\sigma_i} \right)^2$$

and choose m, c which minimises χ^2. What we are doing is maximising the **likelihood** of the data given the model:

$$L = Pr(\{y_i\} \,|\, m, c) \propto \prod_{i=0}^{N-1} \exp\left(-\frac{(y_i - mx_i - c)^2}{2\sigma_i^2}\right)$$
$$= \exp(-\chi^2/2)$$

Likelihood and Bayesian inference

Note that what we really want to know is the inverse of this, i.e. the probability of our model parameters m, c given the data y_i, which is proportional to the likelihood:

$$Pr(m, c \,|\, \{y_i\}) \propto Pr(\{y_i\} \,|\, m, c) Pr(m, c)$$

This is **Bayes' Theorem** — beyond the scope of this course, but Bayesian data analysis is becoming extremely popular in many areas of science, and provides a consistent framework for inference.

For the rest of this analysis, we assume that the *prior* $Pr(m, c)$ is uniform and so the least-squares solution gives us the model parameters which maximise the *posterior* probability $Pr(m, c \,|\, \{y_i\})$.

General least-squares fitting

The least-squares solution for fitting a straight line can be derived analytically. A more general model could have M parameters $= \{\theta_i\}$, $i = 0 \ldots (M-1)$, so that χ^2 becomes:

$$\chi^2(\{\theta_i\}) = \sum_{i=0}^{N-1} \left(\frac{y_i - f(\{\theta_i\}; x_i)}{\sigma_i}\right)^2,$$

where the function f expresses the model.

We now need to find the minimum value of χ^2 as a function of the M parameters θ_i.

Linear least squares

There are many cases where the model is not a straight line, but nevertheless problem is **linear in the model parameters** θ k . The most common case is where the model can be expressed as

$$y(x) = \sum_{k=1}^{M} \theta_k \phi_k(x),$$

where $\{\theta_k\}$ are the model parameters and $\{\phi_k(x)\}$ are the basis functions for the problem and can be **non-linear functions of** x, for example polynomials.

Matrix formulation

We require a least-squares solution to the linear problem

$$\mathbf{A} \ = \mathbf{b} \tag{11.1}$$

where A is the **design matrix** for the problem given by

$$A_{ij} = \frac{\phi_j(x_i)}{\sigma_i},$$

and

$$b_i = \frac{y_i}{\sigma_i}.$$

Note that in general the problem is overdetermined and $\mathbf{A}$ is not square.

Singular value decomposition

There are multiple ways to solve this linear least-squares problem, but the most robust involves the use of **Singular Value Decomposition** (SVD).

SVD decomposes an arbitrary matrix $\mathbf{A}$ into three matrices $\mathbf{U}$, $\mathbf{V}$, and w such that

$$\begin{pmatrix} \\ \mathbf{A} \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \mathbf{U} \\ \\ \end{pmatrix} \cdot \begin{pmatrix} w_1 & & & \\ & w_2 & & \\ & & \cdots & \\ & & \cdots & \\ & & & w_N \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{V}^T \\ \\ \end{pmatrix}$$

The matrices $\mathbf{U}$ and $\mathbf{V}$ are *unitary* i.e. $\mathbf{U}^T \cdot \mathbf{U} = \mathbf{I}$ and $\mathbf{V} \cdot \mathbf{V}^T = \mathbf{V}^T \cdot \mathbf{V} = \mathbf{I}$ and the diagonal matrix $\mathbf{w}$ contains the so-called singular values w_j.

The Moore-Penrose pseudo-inverse

The least-squares solution to equation 11.1 is given by

$$= \mathbf{A}^{+}\mathbf{b}$$

where $\mathbf{A}^{+}$ is the pseudo-inverse of $\mathbf{A}$ given by

$$A^{+} = \mathbf{V} \cdot [\mathrm{diag}(1/w_j)] \cdot \mathbf{U}^{T}$$

Small or zero values of w_j indicate a singularity/degeneracy in the problem and in this case $1/w_j$ should be **replaced by zero** (in this case $1/0 = 0$!).

The SVD-derived matrix inversion is very **robust** — it tells us when the problem is malformed and can still give sensible solutions. There are many more uses for SVD — see "Numerical Recipes".

Fitting a quadratic: create the design matrix

```
import numpy as np
import matplotlib.pyplot as plt

def quadratic(a,x):
    """
    Evaluate a quadratic with coefficients a at ordinate(s) x
    """
    return a[0]+a[1]*x+a[2]*x**2

def design_matrix(x):
    return np.column_stack(
        [quadratic([1,0,0],x),
         quadratic([0,1,0],x),
         quadratic([0,0,1],x),])
```

```
x = np.linspace(9, 11, 6)
A = design_matrix(x)
print(A)
```

```
[[  1.     9.    81.  ]
 [  1.     9.4   88.36]
 [  1.     9.8   96.04]
 [  1.    10.2  104.04]
 [  1.    10.6  112.36]
 [  1.    11.   121.  ]]
```

Fitting a quadratic: solve for θ_i

```
# Generate data
y_true = quadratic([10, -2, 0.1], x)
y_noisy = y_true + np.random.normal(0, 0.01, len(x))
# Solve
Ainv = np.linalg.pinv(A)
theta_true = Ainv @ y_true
print(theta_true)
```

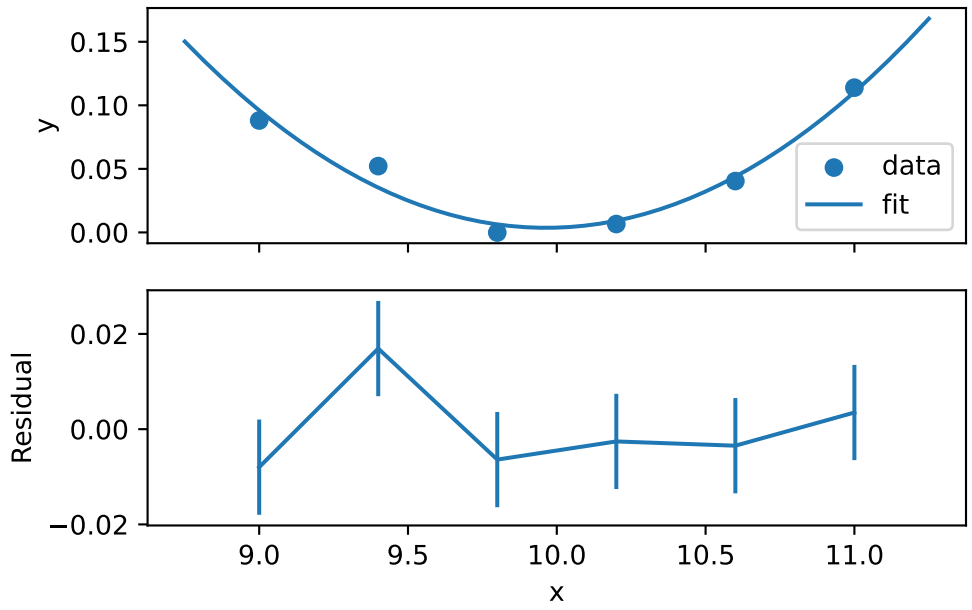
```
[10.  -2.   0.1]
```

```
theta_fit = Ainv @ y_noisy
print(theta_fit)
```

```
[ 9.87871099 -1.98217081  0.09946749]
```

Fitting a quadratic: plot the residuals

```
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
ax1.scatter(x, y_noisy, label="data")
px = np.linspace(8.75, 11.25, 200)
py = quadratic(theta_fit, px)
ax1.plot(px, py, label="fit")
ax1.set_ylabel("y")
ax1.legend()
residuals = y_noisy - quadratic(theta_fit, x)
ax2.errorbar(x, residuals, yerr=0.01)
ax2.set_ylabel("Residual")
ax2.set_xlabel("x");
```



We can sometimes use the residuals to estimate the data errors

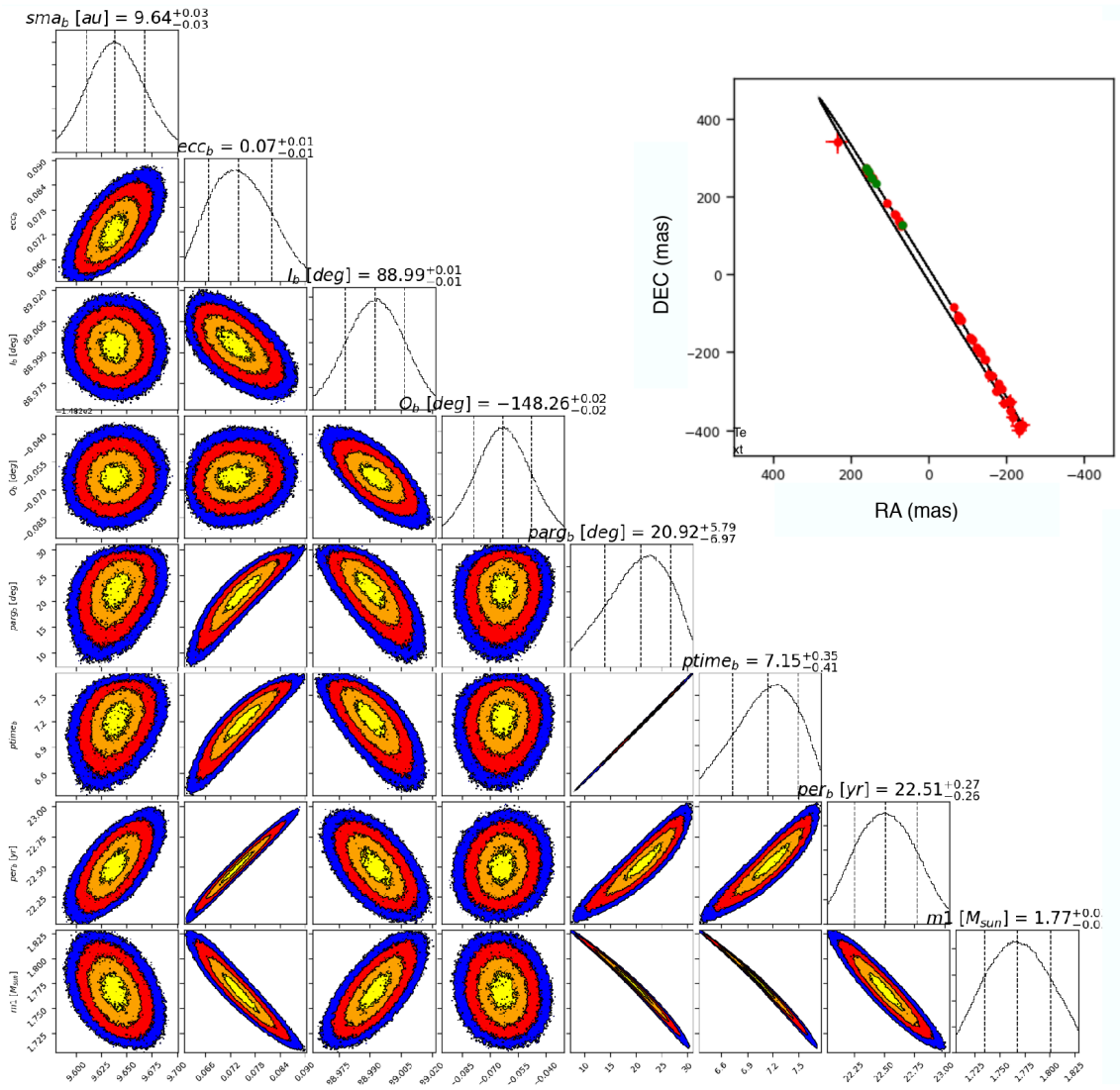
For linear models, it can be shown that the χ^2 of the best-fit model follows the chi-squared distribution with $\nu = N - M$ degrees of freedom. For large values of ν, χ^2 has mean ν and standard deviation $\sqrt{2\nu}$.

Hence, we expect $\chi^2 \sim (N - M)$ for a good fit. If we don't know $\{\sigma_i\}$, we sometimes assume the best fit has $\chi^2 = (M - N)$ then estimate σ_i, **assuming it is the same for all data points**.

Our results must include the uncertainties on the model parameters

These uncertainties are represented by the **posterior probability distribution**.

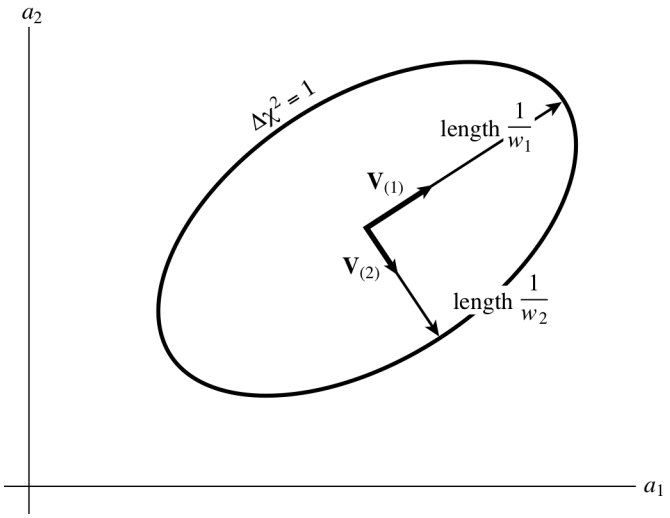
The model space is usually many-dimensional, so we plot projected ("marginalised") distributions over pairs of parameters $\{\theta_i, \theta_j\}$



Singular Value Decomposition directly returns the parameters of the uncertainty ellipse

With a uniform prior, the contours of constant χ^2 are contours of constant posterior probability.

The directions of the principal axes of the uncertainty ellipse are given by the columns of the SVD V matrix, and the inverse of the corresponding singular values give the size of the axes (see "Numerical Recipes").

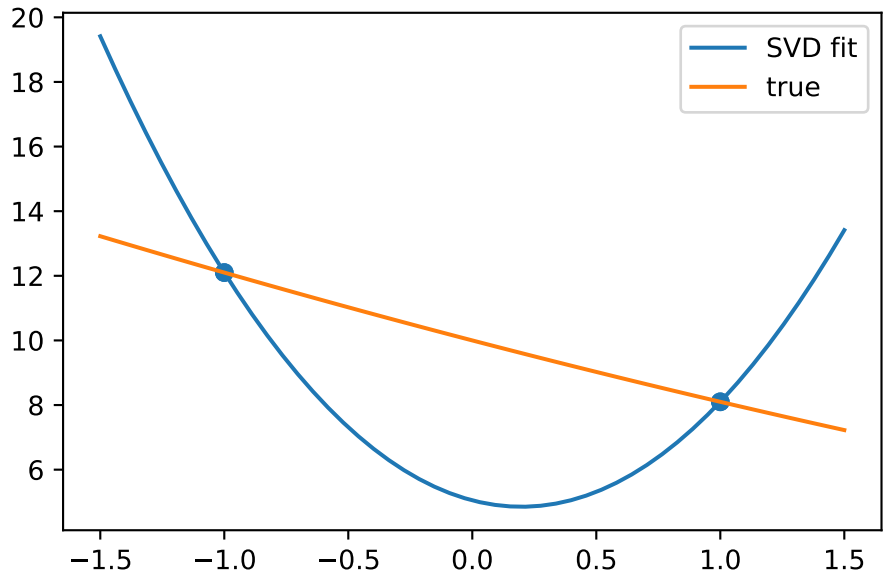


Degeneracies correspond to completely unconstrained directions in model parameter space

```
# Create some data - effectively only 2 sample points
# Underconstrains a quadratic, which has 3 parameters
x=np.array([-1, -1, 1, 1])
A = design_matrix(x)
Ainv = np.linalg.pinv(A)
y_true = quadratic([10, -2, 0.1],x)
theta = Ainv @ y_true
print(theta)
```

```
[ 5.05 -2.    5.05]
```

```
px = np.linspace(-1.5,1.5)
plt.plot(px, quadratic(theta, px), label="SVD fit")
plt.plot(px, quadratic([10, -2, 0.1], px), label="true")
plt.scatter(x,y_true)
plt.legend();
```



Degeneracies are reflected in small or zero singular values

```
u, s, vh = np.linalg.svd(A)
print(s)
```

```
[ 2.82842712  2.          -0.        ]
```

```
print(vh)
```

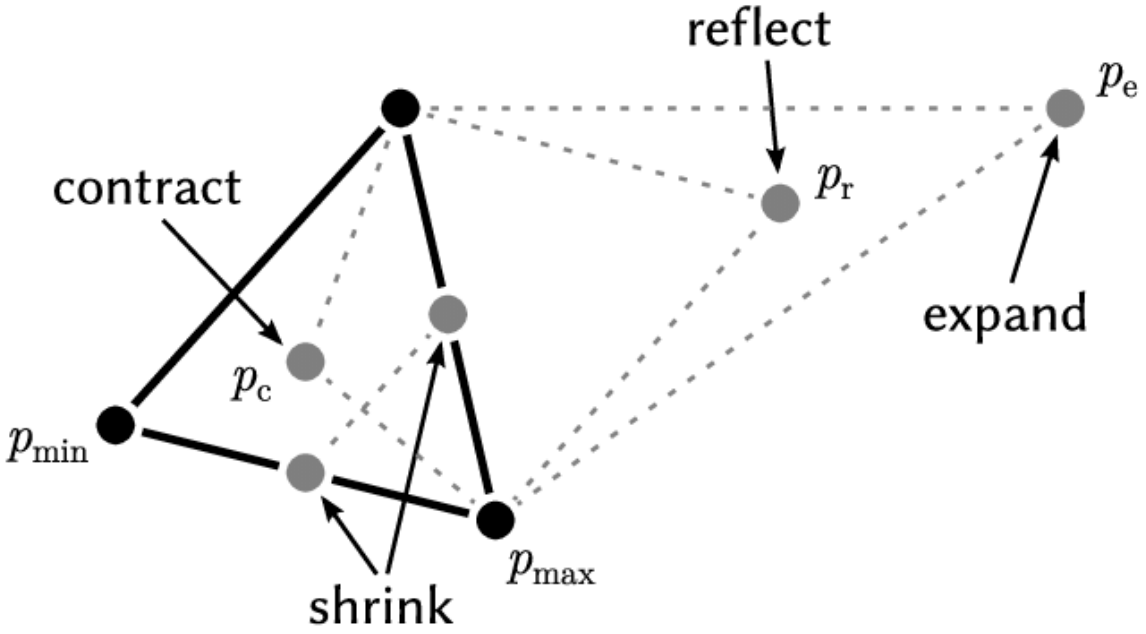
```
[[-0.70710678  0.          -0.70710678]
 [-0.         -1.           0.        ]
 [-0.70710678  0.           0.70710678]]
```

Non-linear optimisation

The most general least-squares problem is the minimisation of the function $\chi^2(\,)$ for non-linear $f(\,)$.

A simple "grid search" becomes inefficient in high-dimensional spaces.

In the absence of derivative information, we can use a bracketing method such as the Nelder-Mead "simplex" method to find the minimum.

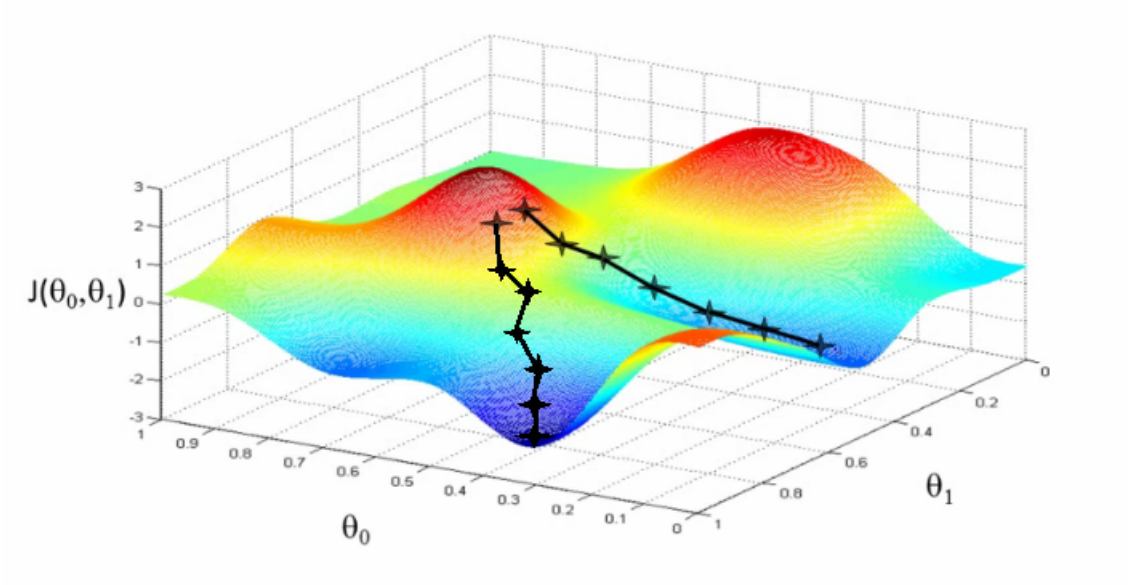


If we have derivative information, "hill-climbing" or gradient-descent type algorithms are typically much faster

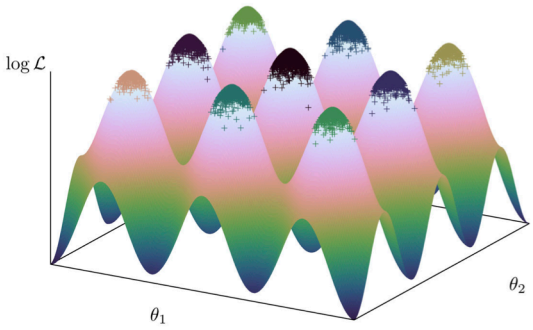
The **Levenberg-Marquardt** algorithm is specifically designed for least-squares problems given gradient information $\partial f(\{\theta_i\})/\partial\theta_j$.

Conjugate gradient methods can be used to solve more general non-linear problems.

There are a range of least-squares fitting functions and multidimensional minimisation algorithms in `scipy.optimize`.



Finding the global maximum/minimum

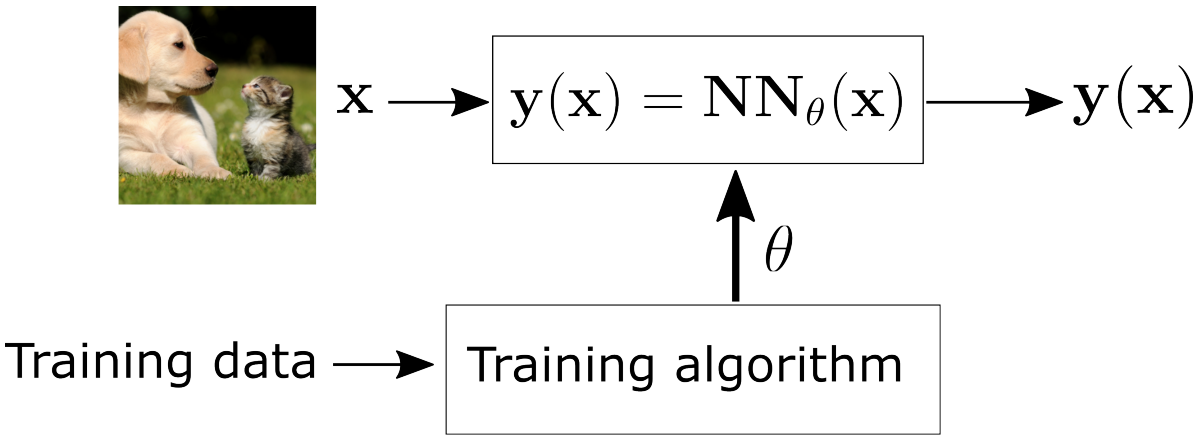


Markov-Chain Monte-Carlo (MCMC) methods use "swarms" of random sample points to explore high-dimensional spaces efficiently when there are multiple maxima/minima — this example is from a recent algorithm, `Polychord`.

Neural Networks

Machine learning: algorithm generation from data

We seek an algorithm that maps an input $\mathbf{x}$ to a desired output $\mathbf{y}$.



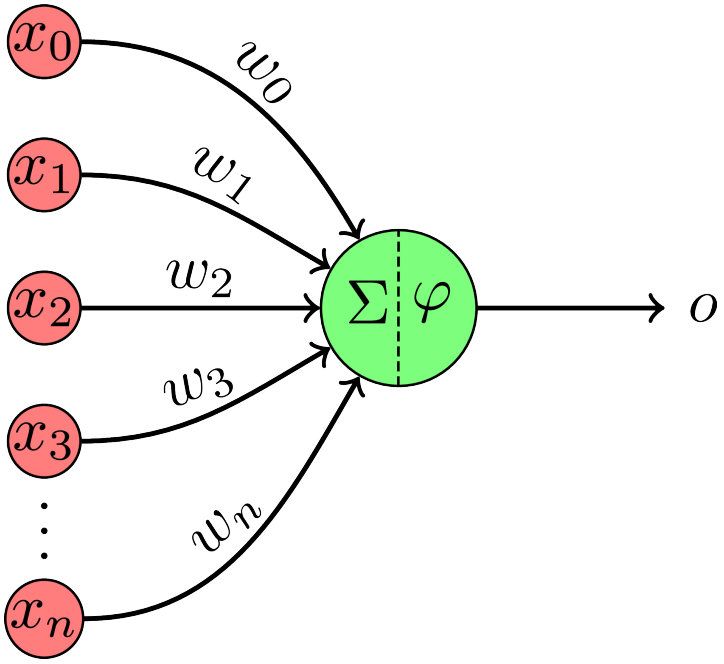
Note that $\mathbf{x}$ and $\mathbf{y}$ are typically vectors: for example $\mathbf{x}$ could be a set of pixel intensities in an image and $\mathbf{y}$ the probabilities of there being a cat and/or dog in the image.

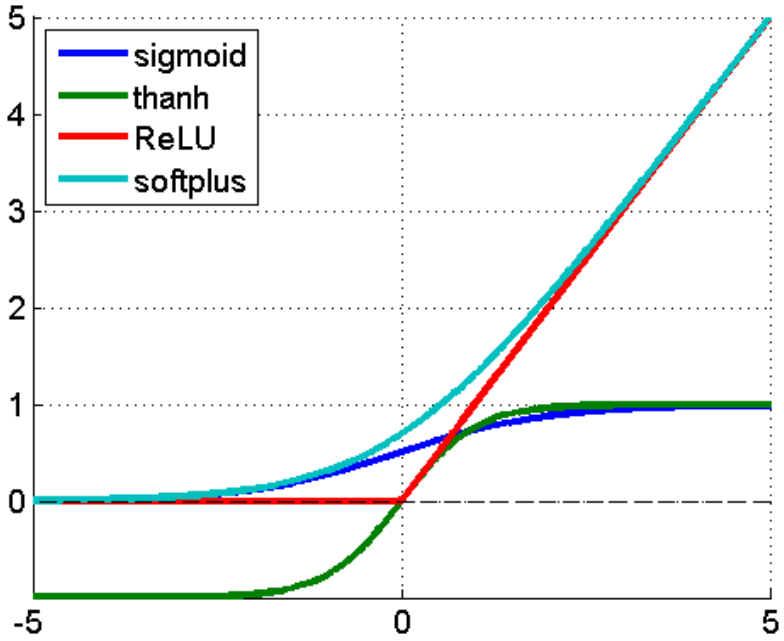
Machine learning with neural networks

$$\mathbf{y}(\mathbf{x}) = \mathsf{NN}_\theta(\mathbf{x})$$

- θ are parameters
- If family of functions is "big enough" then $\exists$ function that does "good job"
- Neural networks are a way of building this family of functions.

A neural network is built out of artificial neurons

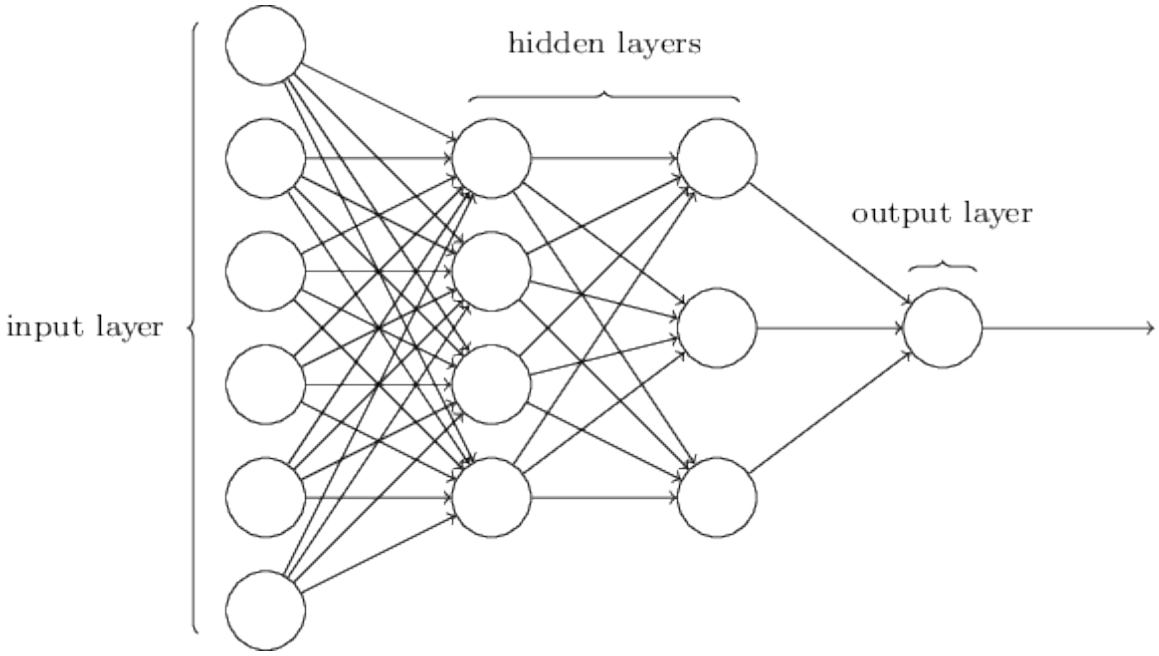




$$o(\mathbf{x}) = \phi\left(\sum_i w_i x_i\right),$$

where ϕ is a non-linear **activation function**.

A "deep" neural network



There are many possible network **architectures** adapted to different types of problems.

The **universal approximation theorem** states that a network with a single hidden layer can approximate any reasonably smooth function. In practice it needed the advent of fast GPUs to allow the training of systems with *several* hidden layers to solve real-world problems — **deep learning**.

Working with discrete labels

- We often need to map $\mathbf{x}$ to an output $\mathbf{y}$ that represents a set of labels
- Labels represent e.g. different kinds of objects that might appear in images
- Popular choice is *one hot encoding*: $\mathbf{y}$ is a vector of length N_L, where in item with label n is encoded as $(0, 0, \ldots, 1, \ldots, 0)$, with 1 in nth place.

- Use approximate $\mathbf{y}$ at output by finding the maximum component and predict corresponding label

---

Training

- Training a neural net is the process of adjusting the set of weights $= \{w_{ij}\}$ to give an output which approximates the desired output as well as possible for all the members of a (hopefully large) **training set** of data.
- Dataset of size N consisting of data $\mathbf{x}_i = 1, ... N$ together with labels l_i
- Encode labels l_i to desired outputs $\mathbf{y}_i$ using, e.g. "one-hot" formalism

---

- Want to train network (choose θ) so that $\mathsf{NN}_\theta(\mathbf{x}_i)$ is close to corresponding $\mathbf{y}_i$ that represents label
- To quantify this introduce *cost* or *loss function*. Simple example is quadratic cost

$$\mathcal{C}(\theta) = \frac{1}{2N}\sum_{i=1}^{N}\|\mathbf{y}_i - \mathsf{NN}_\theta(\mathbf{x}_i)\|^2$$

- Use usual square norm of distance between network output and encoded label
- We *average* over training data, because sometimes our network may not perform so well, confusing different labels:

Gradient descent

- Simple algorithm underlying training
- Cost function is differentiable function of parameters θ
- Idea of gradient descent is to take steps "downhill" i.e. in direction $-\mathcal{C}(\theta)$ in high dimensional space of all parameters
- Each step corresponds to an update of the parameters

Figure 1: Muffin or chihuahua?

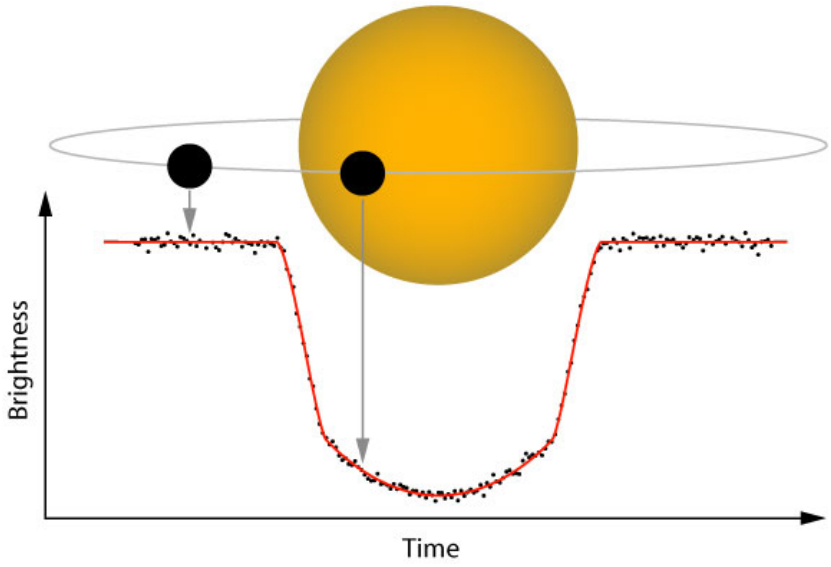
$$\theta_i \longrightarrow \theta_i' = \theta_i - \eta \frac{\partial \mathcal{C}}{\partial \theta_i}$$

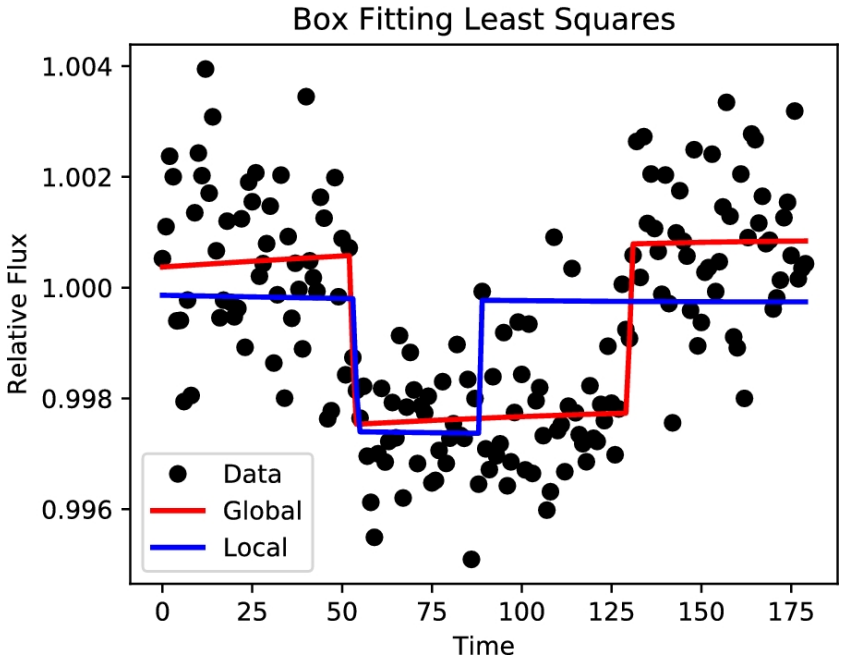
- η is a *hyperparameter* called *learning rate*
- Often *learning rate schedule* is used where rate is adjusted during training to optimize convergence

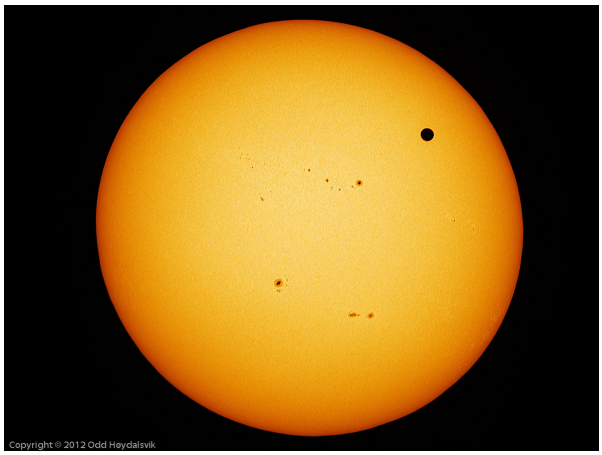
Evaluating performance

- Stardard protocol: split dataset into *training set* and *test set*
- Training set used for training model; test set for evaluating it
- After training model should perform well on training set, but will perform less well on test set
- Difference between cost function evaluated on test set and training set is a measure of how well the model generalizes to new inputs: generalization error

Example application — detecting exoplanet transits







For the Earth transiting the Sun, the fractional "dip" is of the order 10^{-4}.

Training and test datasets