Introduction to High Performance Computing

Autumn, 2018

Lecture 6

Assessment

Corrected assignment dates:
HW1: Assigned 23/10, due 2/11 (**20%**)
HW2: Assigned 6/11, due 16/11 (**25%**)
HW3: Assigned 20/11, due 29/11 (**25%**)
HW4 Assigned 30/11, due 14/12 (**30%**)

- **HW1 will be posted on the course webpage today around 7pm**
- **Submitting this assignment commits you to the course**
- **The assignment is designed to be challenging but not to make you suffer: please come to office hours and ask questions!**

Today

- Continue discussion of optimization
- Brief, informal introduction to neural networks
- More on optimization and neural networks in this week's lab

Optimization: basic ideas

- All optimization routines require the user to specify the function to be minimized (usually called the *cost* function or the *objective* function)
- Most optimization routines use the gradient to search for a minimum (starting from an user-specified "guess")
- Some routines use the Hessian, some don't
- Today, we'll look at two methods:

  1. (Truncated) Newton's method: uses gradient and Hessian
  2. BFGS method: uses gradient

Newton's method

- **Classical method:**
  1. **Evaluate function, gradient and Hessian at a guess, x_0**
  2. **Use these values to fit a quadratic to the function:**

$$g(\mathbf{h}) = f(\mathbf{x}_0) + \mathbf{h}^T \nabla f|_{\mathbf{x}_0} + \tfrac{1}{2}\mathbf{h}^T H|_{\mathbf{x}_0}\mathbf{h}$$

  3. **Setting** $\frac{\partial g}{\partial \mathbf{h}} = 0$ **:**

$$H|_{\mathbf{x}_0}\mathbf{h} = -\nabla f|_{\mathbf{x}_0}$$

  4. **This can be solved for h and the new guess for the minimum is then:** $x_1 = x_0 + h$

- **Classical method only works well if guess is good**

- **In practice, a quadratic fit may be inaccurate or inappropriate**
  - **Truncated-Newton and BFGS methods both address this**

Truncated Newton's method

- **Classical method:**
  1. **Evaluate function, gradient and Hessian at a guess, x_0**
  2. **Use these values to fit a quadratic to the function:**

$$g(\mathbf{h}) = f(\mathbf{x}_0) + \mathbf{h}^T \nabla f|_{\mathbf{x}_0} + \tfrac{1}{2}\mathbf{h}^T H|_{\mathbf{x}_0}\mathbf{h}$$

  3. **Setting** $\frac{\partial g}{\partial \mathbf{h}} = 0$ **:**

$$H|_{\mathbf{x}_0}\mathbf{h} = -\nabla f|_{\mathbf{x}_0}$$

  4. **This can be solved for h and the new guess for the minimum is then:**

$x_1 = x_0 + h$

Truncated approach:

1. **Iteratively solve for h as in the classical approach (but truncate iterations if local curvature is negative)**
2. **Then search in the direction of h for the minimum along this direction**

- **Step 2 is called a *line search*, idea is similar to finding the zero of a 1-d function (e.g. Newton-Rhapson, secant, Brent methods)**

Optimizers in Scipy

- **We will use the** scipy.optimize **package**
- **And particularly, the function,** scipy.optimize.minimize
- **The optimization *method* can be specified when calling this function, e.g.:**

  method = 'Newton-CG'

- **This will use truncated Newton's method**
- **The CG indicates the conjugate gradient method is used to solve for** h
- **There is a separate CG method for optimization**

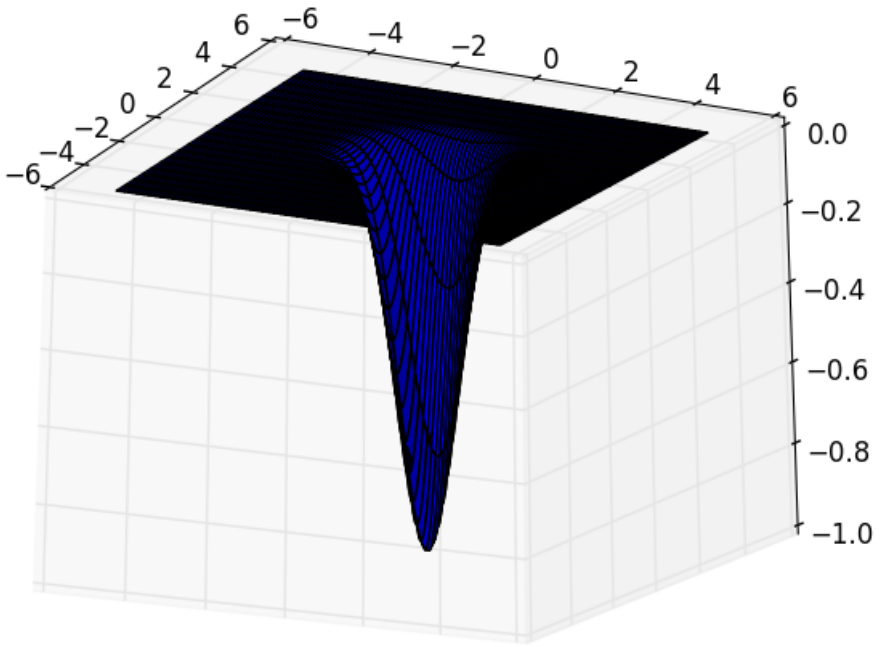
Optimizers in Scipy

- Newton-CG **requires the user to specify:**
  - **A python function which computes the cost function**
  - **A function for computing the gradient**
  - **A function for computing the Hessian (or instruct the optimizer to approximate it)**
  - **A guess,** x0**, where the optimizer starts its search for a minimum**

Simple example

- **Simple illustrative example:**

Find x and y that minimize $f = -exp(-\alpha(x - x_0)^2 - \beta(y - y_0)^2)$



Simple example

The code, *gauss2d.py*, applies a few different approaches to this problem

- **It has three functions:** gauss2d, gauss2d_grad, gauss2d_hess
- gauss2d_grad **returns the two components of the gradient:**

$$\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}) = (-2\alpha(x - x_0)f, -2\beta(y - y_0)f)$$

- gauss2d_hess **returns the 2 x 2 Hessian matrix:**

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y \partial y} \end{bmatrix} = \begin{bmatrix} -2\alpha(f + (x - x_0)\frac{\partial f}{\partial x}) & 4\alpha\beta(x - x_0)(y - y_0)f \\ 4\alpha\beta(x - x_0)(y - y_0)f & -2\beta(f + (y - y_0)\frac{\partial f}{\partial y}) \end{bmatrix}$$

- **The minimizer is called near the bottom with different inputs**

Simple example

- **The code, *gauss2d.py*, applies a few different approaches**
- **Let's first look at the Newton-CG, approximate gradient case (ncg1=True)**
- **The call to the minimizer is:**

```
minimize(gauss2d,xguess,args=parameters,method='Newton-CG',jac=gauss2d_grad)
```

Note: since the name of the Hessian function is not specified, it is approximated from the gradient

Simple example

- **Running the code:**

```
In [44]: run gauss2d
Gauss2d, Newton-CG, approximate Hessian
optimum: [ 0.5   0.25]
info:    status: 0
 success: True
    njev: 17
    nfev: 6
     fun: -1.0
       x: array([ 0.5 ,  0.25])
 message: 'Optimization terminated successfully.'
    nhev: 0
     jac: array([ -5.37315725e-09,  -2.68657863e-09])
```

Notes:

- gauss2d **sets (**x0,y0**) to (**0.5,0.25**) so the answer is correct**
- **17 gradient evaluations and 6 function evaluations were needed**
- **What happens if we now specify the exact Hessian (ncg2=True)?**

Simple example

Running the code with:

minimize(gauss2d,xguess,args=parameters,method='NewtonCG',
jac=gauss2d_grad,hess=gauss2d_hess)

```
Gauss2d, Newton-CG, exact Hessian
optimum: [ 0.5   0.25]
info:    status: 0
 success: True
    njev: 9
    nfev: 6
     fun: -1.0
       x: array([ 0.5 ,  0.25])
 message: 'Optimization terminated successfully.'
    nhev: 4
     jac: array([ -2.28865316e-09,  -1.14432658e-09])
```

Notes:

- **The number of gradient evaluations reduced from 17 to 9**
- gauss2d **also has options for BFGS and Nelder-Mead methods – how do these work?**

Overview of BFGS

- BFGS is generally the method-of-choice for unconstrained optimization problems involving smooth functions
- Works surprisingly well for non-smooth/noisy functions as well
- Can be memory-intensive, use L-BFGS-B for large problems
- Motivating idea:
  - Computing and inverting the Hessian can be very expensive
  - Instead, approximate the (inverse) Hessian, and update this approximation at each step (quasi-Newtonian methods)

Overview of BFGS

- **Motivating idea:**
  - **Computing and inverting the Hessian can be very expensive**
  - **Instead, approximate the (inverse) Hessian, and update this approximation at each step (quasi-Newtonian methods)**
- **Newton-CG:**

1. **Solve for** h: $H|_{\mathbf{x}_0}\mathbf{h} = -\nabla f|_{\mathbf{x}_0}$
2. **Search along direction of** h **for minimum →** x_1 **and repeat step 1**

- **BFGS:**

1. **Approximate inverse Hessian and solve for** h: $M^i \approx H^{-1}$, $\mathbf{h} = -M^i \nabla f|_{\mathbf{x}_i}$
2. **Similar to Newton-CG**
3. **Update** M: $M^{i+1} = f(M^i, h)$

- **The update should result in a positive-definite (approximate) Hessian, and the BFGS update formula produces particularly good results**

- **Now back to** *gauss2d*

Simple example -- BFGS

Results for BFGS with approximate gradient

minimize(gauss2d,xguess,args=parameters,method='BFGS',jac=False)

```
Gauss2d, BFGS, approximate gradient
optimum: [ 0.49999556  0.24999778]
info:     status: 0
  success: True
     njev: 5
     nfev: 20
 hess_inv: array([[ 0.60077592, -0.19961208],
       [-0.19961208,  0.90019394]])
      fun: -0.9999999999753861
        x: array([ 0.49999556,  0.24999778])
  message: 'Optimization terminated successfully.'
      jac: array([ -8.85874033e-06,  -4.43309546e-06])
```

Simple example -- BFGS

Results for BFGS with exact gradient

minimize(gauss2d,xguess,args=parameters,method='BFGS',jac=gauss2d_grad)

```
Gauss2d, BFGS, exact gradient
optimum: [ 0.49999557  0.24999779]
info:     status: 0
  success: True
     njev: 5
     nfev: 5
 hess_inv: array([[ 0.60077589, -0.19961205],
       [-0.19961205,  0.90019397]])
      fun: -0.9999999999754784
        x: array([ 0.49999557,  0.24999779])
  message: 'Optimization terminated successfully.'
      jac: array([ -8.85828506e-06,  -4.42914253e-06])
```

Notes:

1. **BFGS requires less function, gradient evaluations than Newton-CG**
2. **For large (slow) problems, specifying the gradient (and Hessian if appropriate) can make a big difference**

Neural network

Motivating idea:

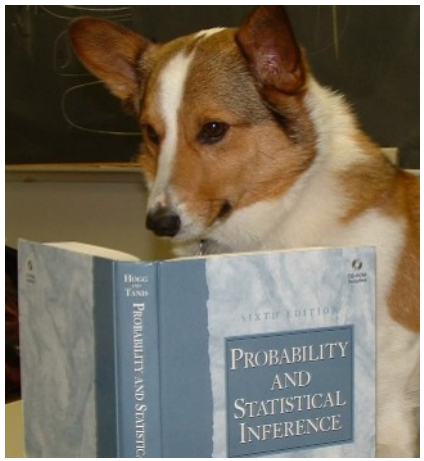
- **Classifying "objects" – assigning objects to categories – is intuitive for humans**
- **But often difficult to produce sufficiently accurate rules for classification that could be used by computers**

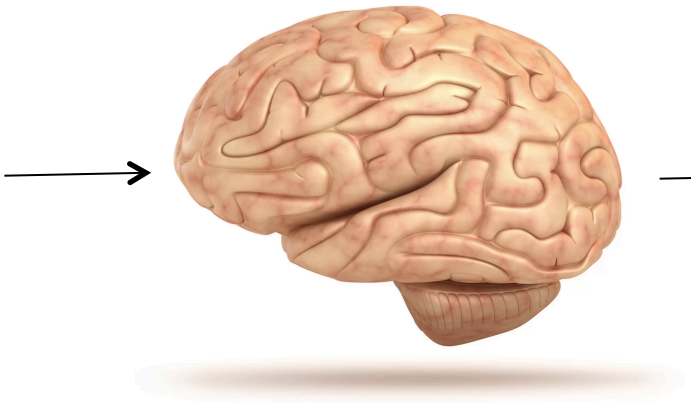
Neural network

Motivating idea:

- **Classifying "objects" – assigning objects to categories – is intuitive for humans**
- **But often difficult to produce sufficiently accurate rules for classification that could be used by computers**

Example:





"mathematician"

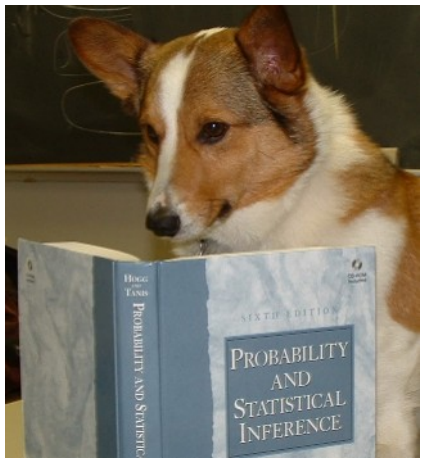
Neural network

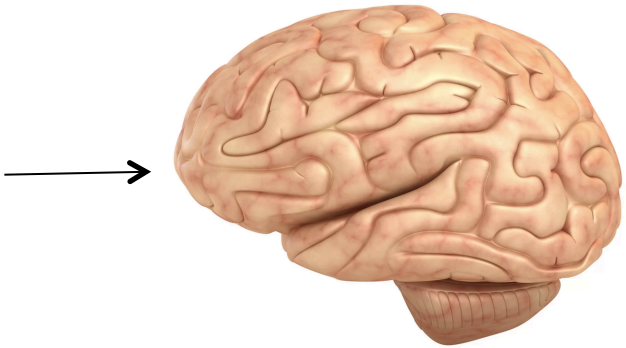
Motivating idea:

- Classifying "objects" – assigning objects to categories – is
- intuitive for humans

- But often difficult to produce sufficiently accurate rules for classification that could be used by computers

Example:





"~~mathematician~~"
"dog"

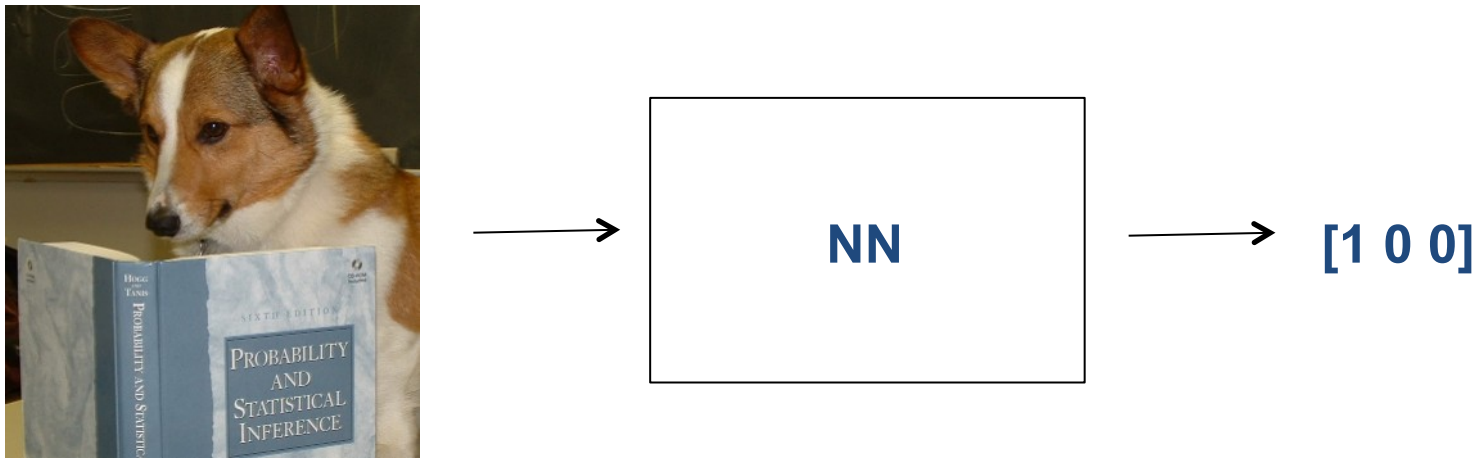
But how can we write a program to classify various breeds of dogs ovbserved from different angles?

Neural network

Solution:

- **Work with a highly flexible model with several fitting parameters**
- **Set the fitting parameters by calibrating the model with a large *labeled* dataset**

Example:

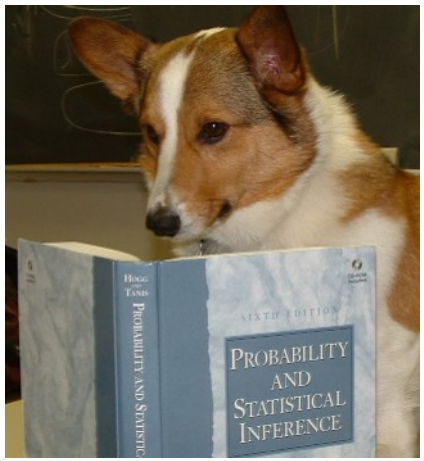


Neural network

Solution:

- **Work with a highly flexible model with several fitting parameters**
- **Set the fitting parameters by calibrating the model with a large *labeled* dataset**

Example:



→ **NN** → **[1 0 0]**

where [1 0 0] = “dog”
[0 1 0] = “cat”
[0 0 1] = “mouse”

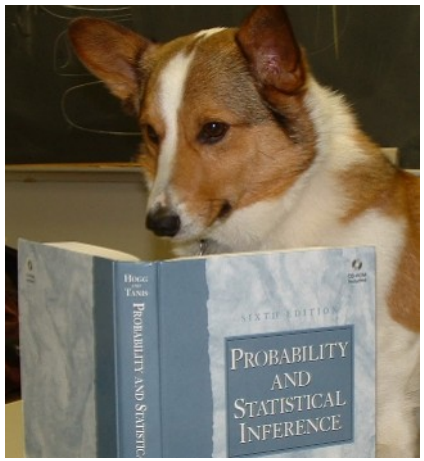
Neural network

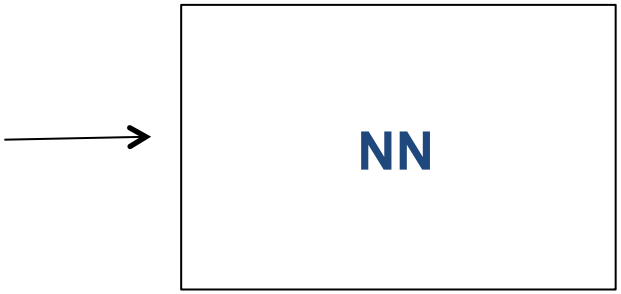
The NN is “trained” with images labeled as “dog”, “cat”, or “mouse”

or

The model parameters of the NN have been calibrated to minimize the classification error when labeled images are provided as input

Example:



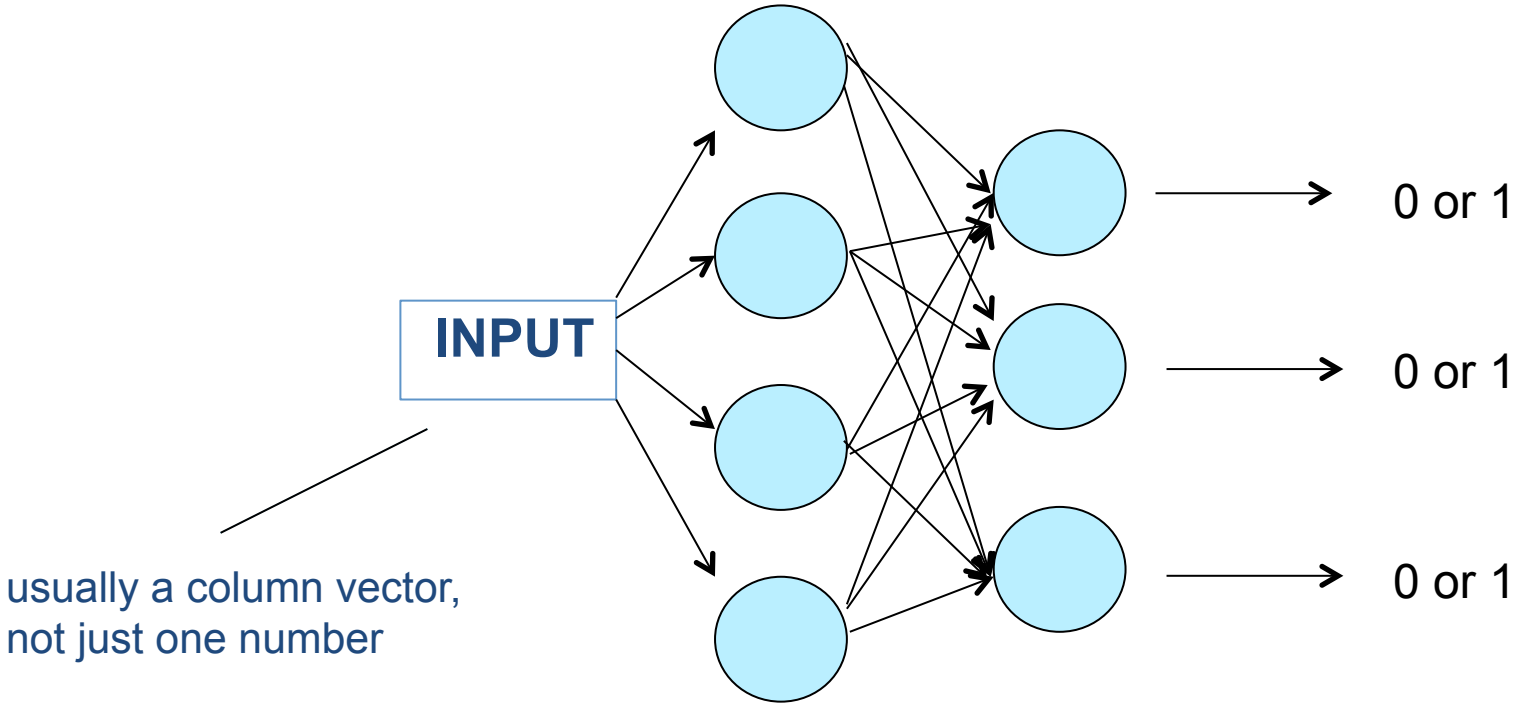


[1 0 0]

where [1 0 0] = “dog”
[0 1 0] = “cat”
[0 0 1] = “mouse”

Neural network

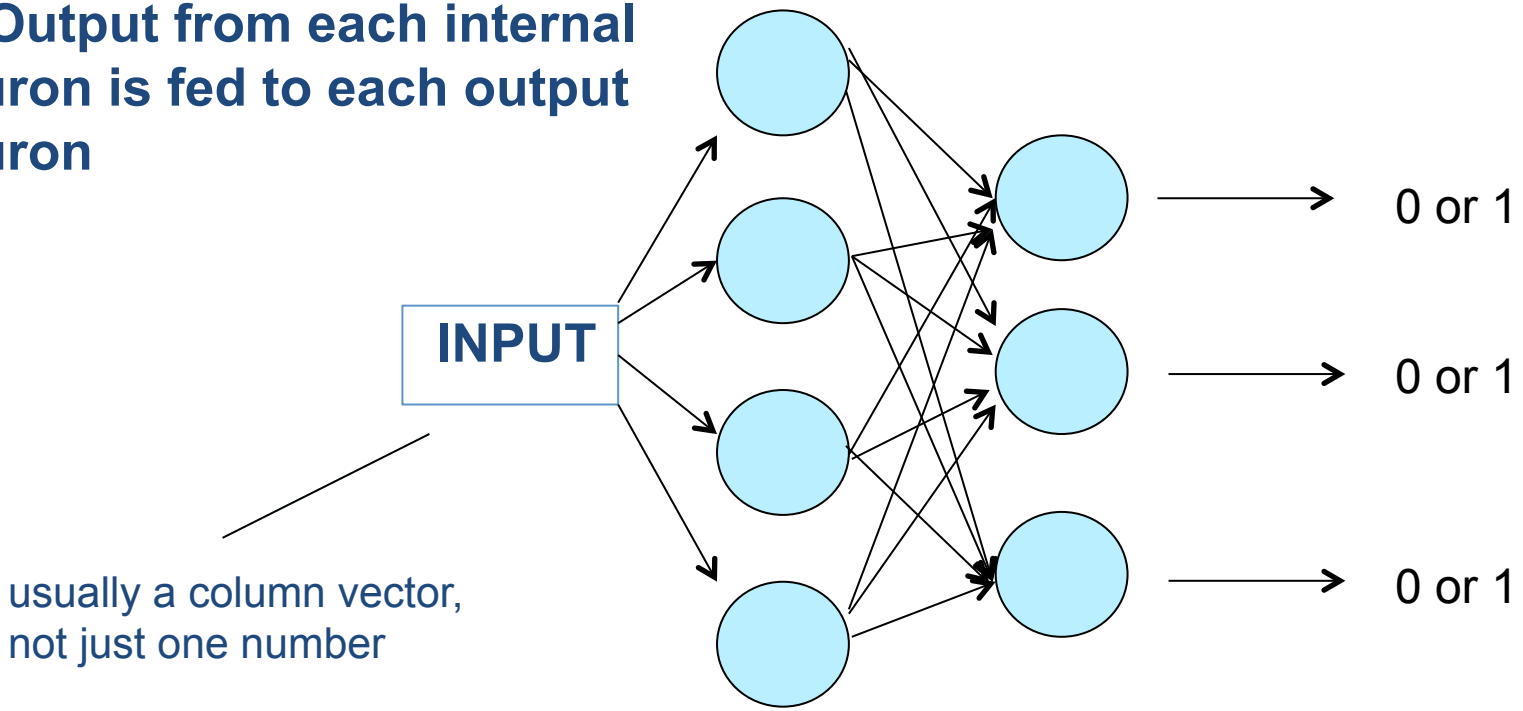
Some details: A NN is a network of "decision making units"



Neural network

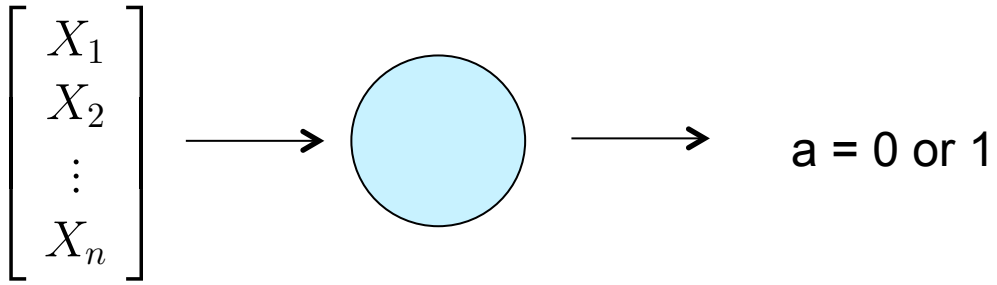
Some details: A NN is a network of "decision making units"

- **Input is fed to each neuron in internal layer**
- **Output from each internal neuron is fed to each output neuron**



Individual neuron

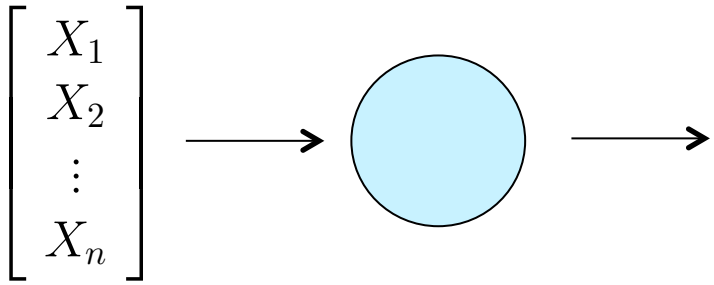
What does an individual neuron do?



"Perceptron" model: difficult to calibrate, small changes in input can lead to large jumps in output

Individual neuron

What does an individual neuron do?



0 <= a <= 1

With a "tendency" to provide values close to zero or one

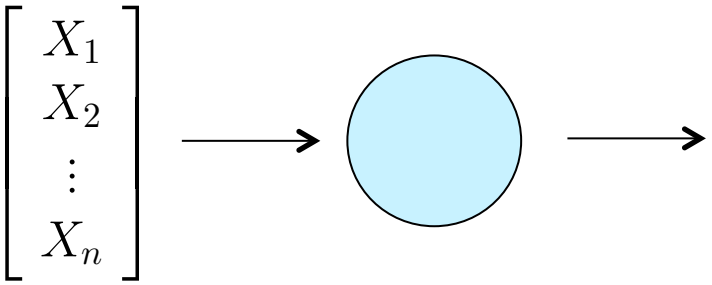
"tendency?":

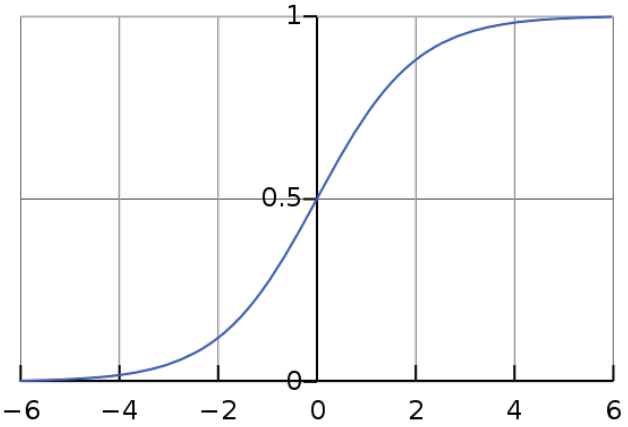
Weighted input: $z = b + w_1 X_1 + w_2 X_2 + ... + w_N X_N$

Output: $a = \frac{1}{1+e^{-z}}$

Individual neuron

What does an individual neuron do?





"tendency?":

Weighted input: $z = b + w_1 X_1 + w_2 X_2 + ... + w_N X_N$

Output: $a = \frac{1}{1+e^{-z}}$

Individual neuron

Weighted input: $z = b + w_1 X_1 + w_2 X_2 + ... + w_N X_N$

Output: $a = \frac{1}{1+e^{-z}}$

How do we set the weights, w_i and bias, b?

This is an optimization problem!:

$$\text{Find } b, w_1, w_2, ..., w_N \text{ such that } C = \frac{1}{2M} \sum_{k=1}^{M} (a(z^{(k)}) - Y^{(k)})^2 \text{ is minimized}$$

The kth input dataset, $X^{(k)}$, is labeled with $Y^{(k)}$ = 0 or 1

Optimization routines will require gradients, $\frac{\partial C}{\partial w_i}, \frac{\partial C}{\partial b}$

Individual neuron

Optimization routines will require gradients, $\frac{\partial C}{\partial w_i}, \frac{\partial C}{\partial b}$

Find $b, w_1, w_2, ..., w_N$ such that $C = \frac{1}{2M} \sum_{k=1}^{M} (a(z^{(k)}) - Y^{(k)})^2$ is minimized

$$z = b + w_1 X_1 + w_2 X_2 + ... + w_N X_N$$

$$a = \frac{1}{1+e^{-z}}$$

- **Expressions for gradients can easily be obtained with chain rule**

Individual neuron

Optimization routines will require gradients, $\frac{\partial C}{\partial w_i}, \frac{\partial C}{\partial b}$

Find $b, w_1, w_2, ..., w_N$ such that $C = \frac{1}{2M} \sum_{k=1}^{M} (a(z^{(k)}) - Y^{(k)})^2$ is minimized

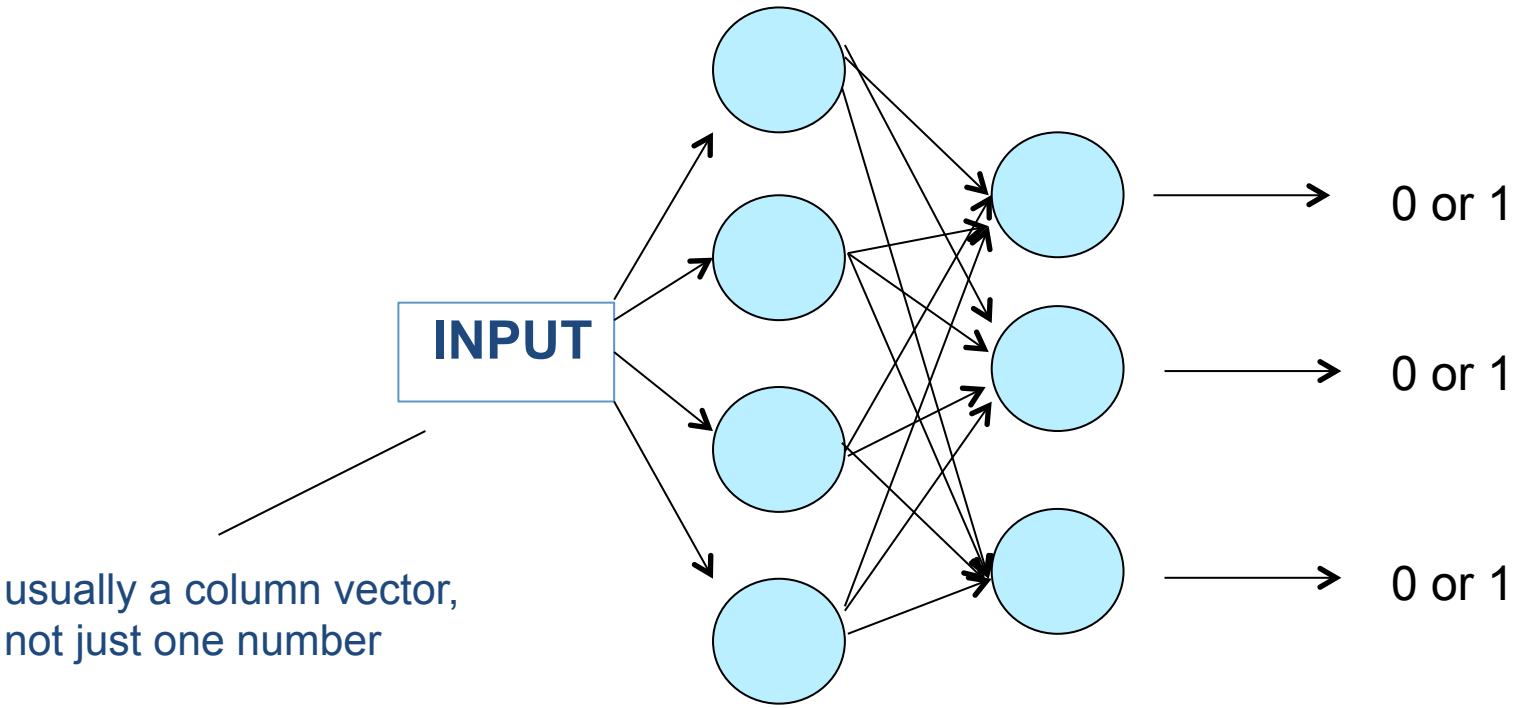
$$z = b + w_1 X_1 + w_2 X_2 + ... + w_N X_N$$

$$a = \frac{1}{1+e^{-z}}$$

- **Expressions for gradients can easily be obtained with chain rule**
- **A single neuron = logistic regression**
- **Typically use likelihood rather than least-squares error in cost function**
- **This is a neuron, what about a NN?**

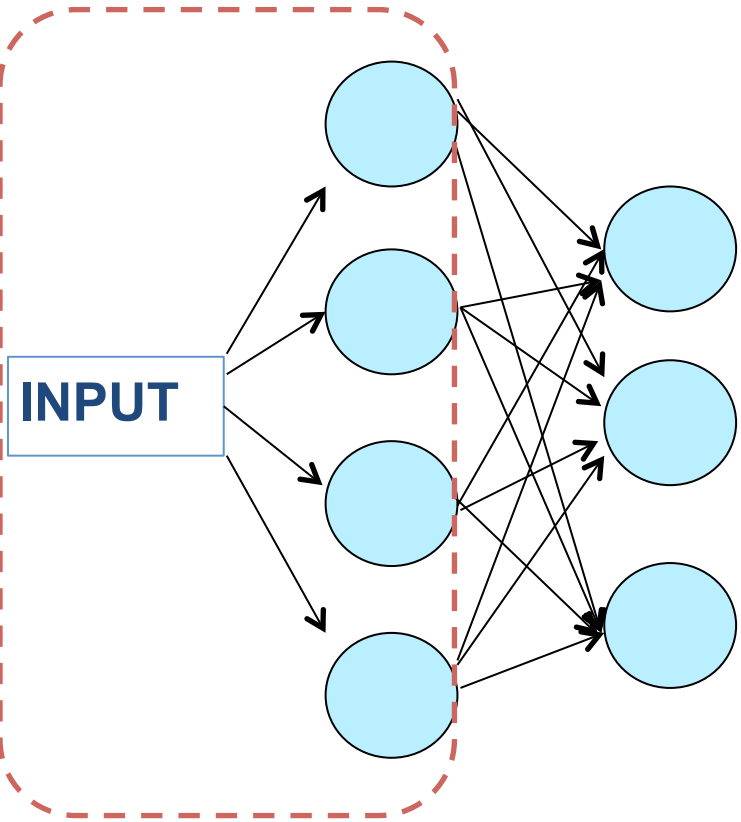
Neural network

Same idea, just more algebra!



Neural network

Same idea, just more algebra!



- **Single neuron: vector of weights, $w_1, w_2, \ldots, w_N$**
- **Inner layer: *matrix* of weights, w_{ij}**

Output of ith neuron in inner layer:

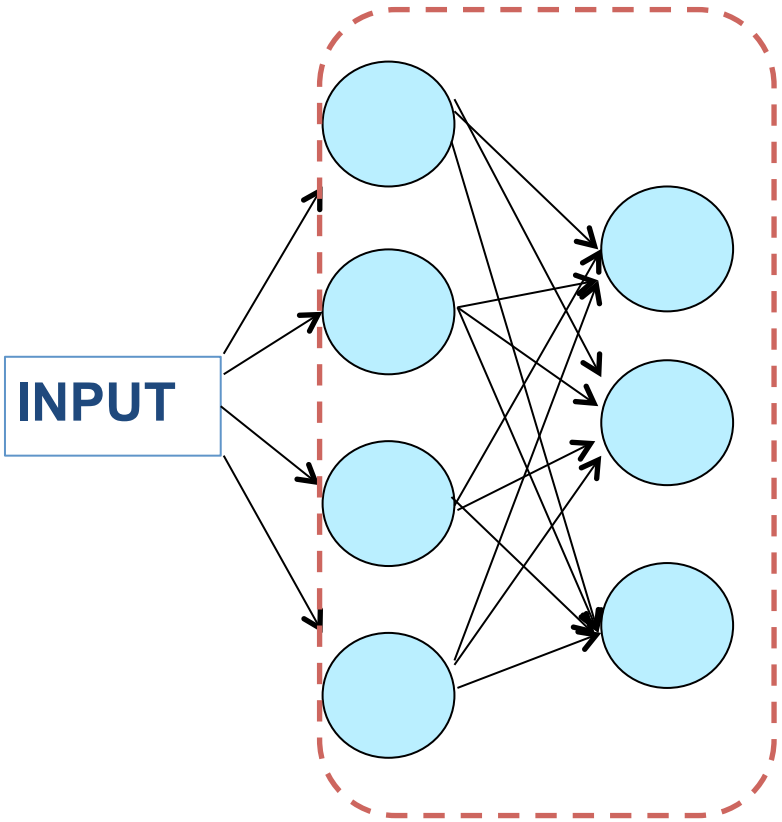
$$a_i = f(z_i)$$

$$z_i = \sum_{j=1}^{N} (w_{ij} X_j + b_i)$$

- **Here f(z) is sigmoid function as before**
- **w_{ij} is the weight connecting the ith neuron to the jth element of the input vector**

Neural network

Same idea, just more algebra!



- **Single neuron: vector of weights, $w_1, w_2, \ldots, w_N$**
- **Output layer: *matrix* of weights, $\hat{w}_{ij}$**

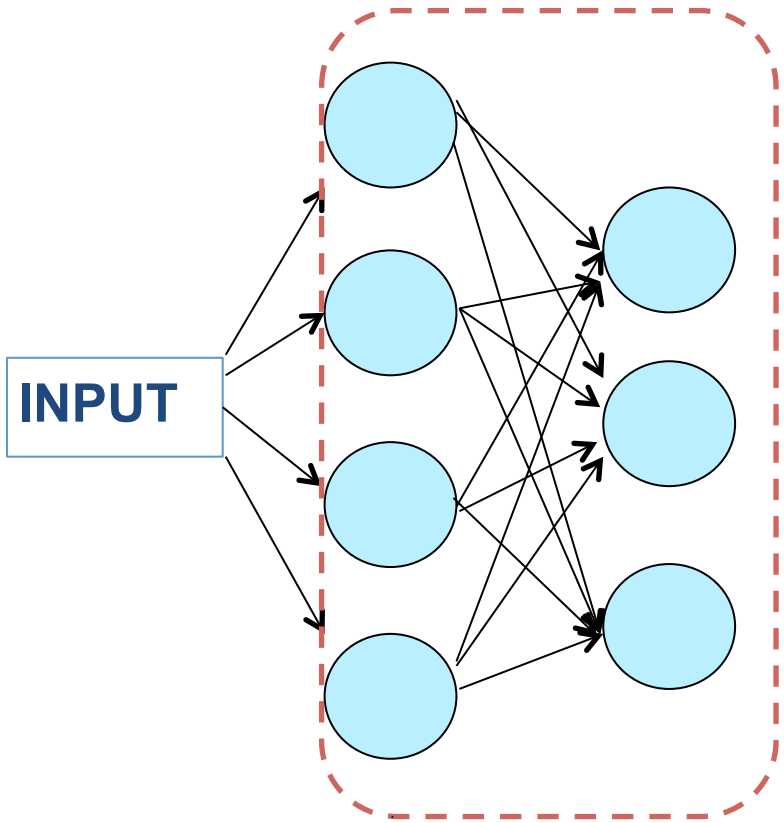
Output of ith neuron in output layer: $\hat{a}_i = f(\hat{z}_i)$

$$\hat{z}_i = \sum_{j=1}^{N} \left(\hat{w}_{ij} a_j + \hat{b}_i \right)$$

- **Here f(z) is logistic function as before**
- **$\hat{w}_{ij}$ is the weight connecting the ith neuron to the output from the jth inner neuron**

Neural network

Same idea, just more algebra!



Output of ith neuron in output layer: $\hat{a}_i = f(\hat{z}_i)$

$$\hat{z}_i = \sum_{j=1}^{N} \left(\hat{w}_{ij} a_j + \hat{b}_i \right)$$

- **Here f(z) is logistic function as before**
- **$\hat{w}_{ij}$ is the weight connecting the ith neuron to the output from the jth inner neuron**
- **Final problem statement:**

$$\text{Find } b_i, w_{ij}, \hat{b}_i, \hat{w}_{ij} \text{ such that } C = \frac{1}{2M} \sum_{k=1}^{M} (\hat{a}(\hat{z}^{(k)}) - Y^{(k)})^2 \text{ is minimized}$$

- **Optimizer needs:** $\frac{\partial C}{\partial w_{ij}}, \frac{\partial C}{\partial b_i}, \frac{\partial C}{\partial \hat{w}_{ij}}, \frac{\partial C}{\partial \hat{b}_i}$

Neural network

- **These are just the basics – there is much more to the study of NNs**
- **However, many powerful NN libraries are readily available**
  - **A basic idea is all you needed to get started with these**
- **We will look at the Python module, scikit-learn**
  - **Specifically will use the** MLPClassifier **function in** sklearn.neural_network

Example:

```
mlp = MLPClassifier(hidden_layer_sizes=(50,), max_iter=10, alpha=1e-4,
                    solver='sgd', verbose=10, tol=1e-4, random_state=1,
                    learning_rate_init=.1)
```