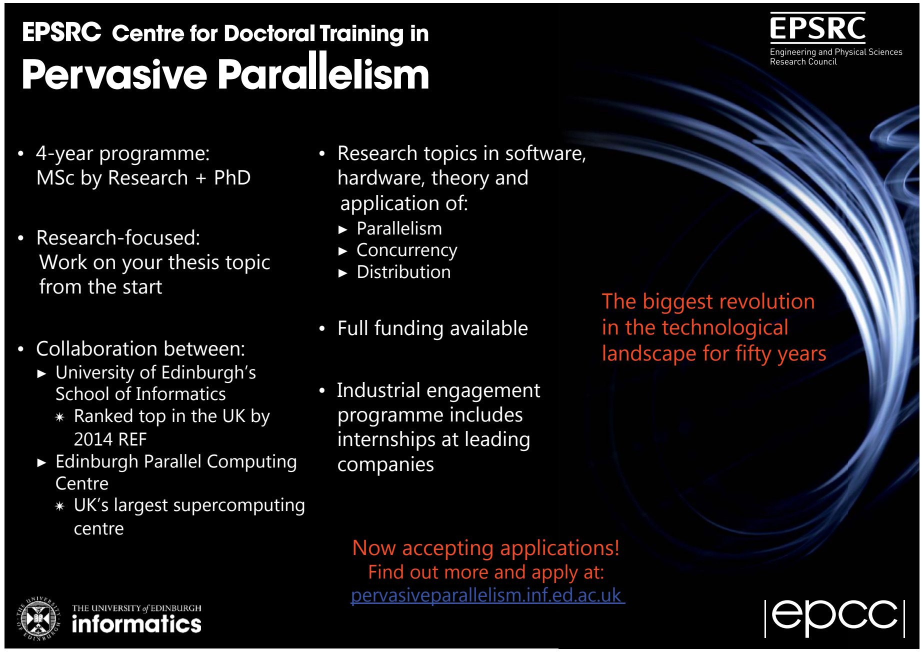
Machine Translation

02: Neural Network Basics

Rico Sennrich

University of Edinburgh



Today's Lecture

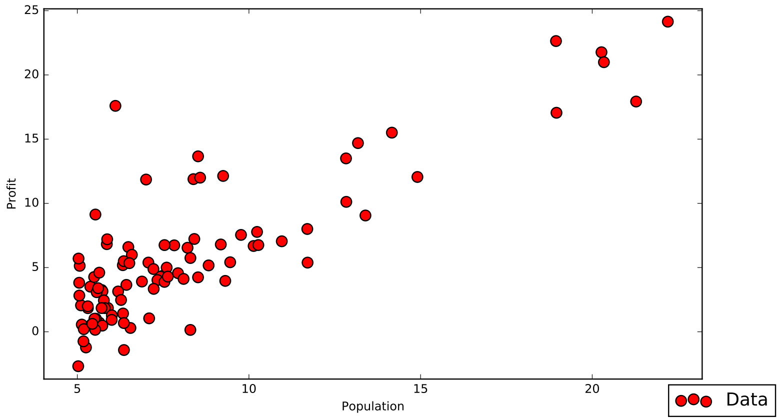
- linear regression
- stochastic gradient descent (SGD)
- backpropagation
- a simple neural network

Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$ Model: $h_\theta(x) = \theta_0 + \theta_1 x$

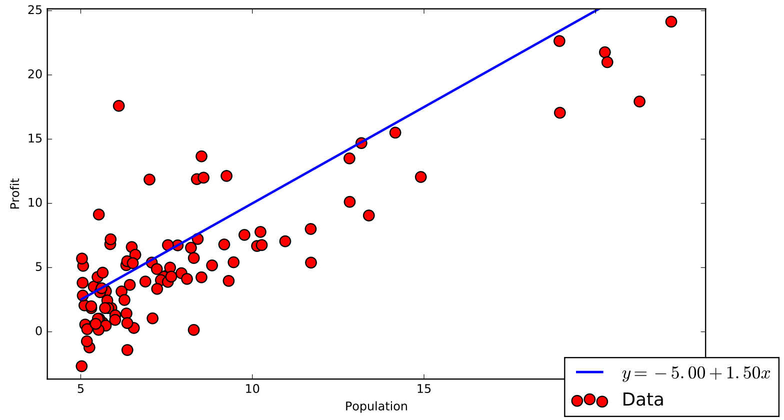
Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$ Model: $h_\theta(x) = \theta_0 + \theta_1 x$



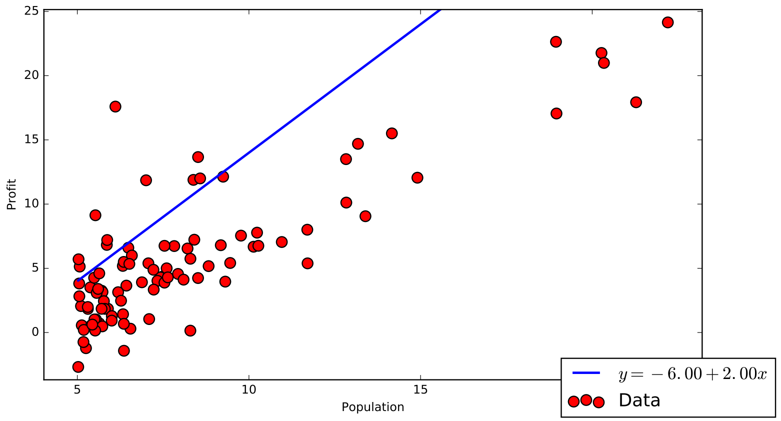
Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$ Model: $h_\theta(x) = \theta_0 + \theta_1 x$



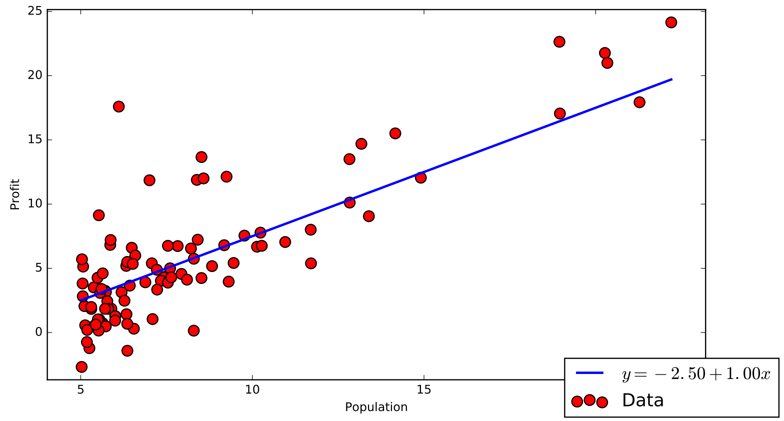
Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$ Model: $h_\theta(x) = \theta_0 + \theta_1 x$



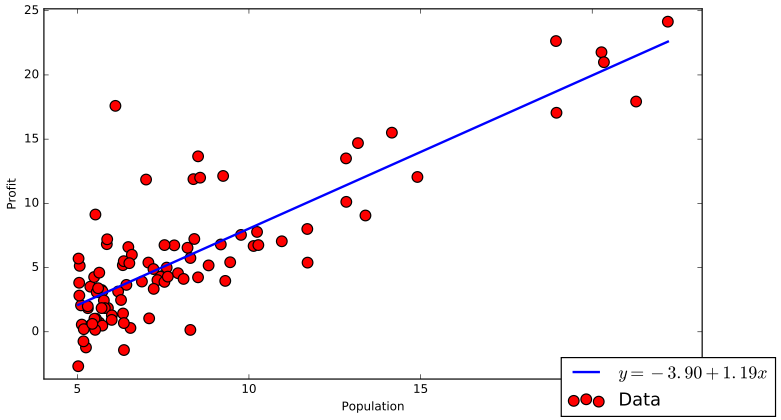
Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$ Model: $h_\theta(x) = \theta_0 + \theta_1 x$



Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$ Model: $h_\theta(x) = \theta_0 + \theta_1 x$



The cost (or loss) function

- We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

$$J : \mathbb{R}^2 \to \mathbb{R}$$

$$\hat{\theta} = \arg\min_{\theta \in \mathbb{R}^2} J(\theta)$$

The cost (or loss) function

- We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

$$J : \mathbb{R}^2 \to \mathbb{R}$$

$$\hat{\theta} = \arg\min_{\theta \in \mathbb{R}^2} J(\theta)$$

- Mean Square Error:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

The cost (or loss) function

- We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

$$J : \mathbb{R}^2 \to \mathbb{R}$$

$$\hat{\theta} = \arg\min_{\theta \in \mathbb{R}^2} J(\theta)$$

- Mean Square Error:

$$\begin{aligned} J(\theta) &= \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \\ &= \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2 \end{aligned}$$

The cost (or loss) function

- We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

$$J : \mathbb{R}^2 \rightarrow \mathbb{R}$$

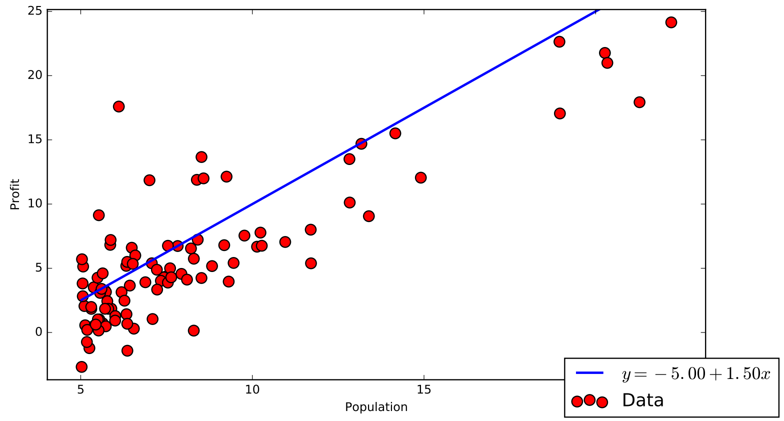
$$\hat{\theta} = \underset{\theta \in \mathbb{R}^2}{\arg\min}\ J(\theta)$$

- Mean Square Error:

$$\begin{aligned} J(\theta) &= \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \\ &= \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2 \end{aligned}$$

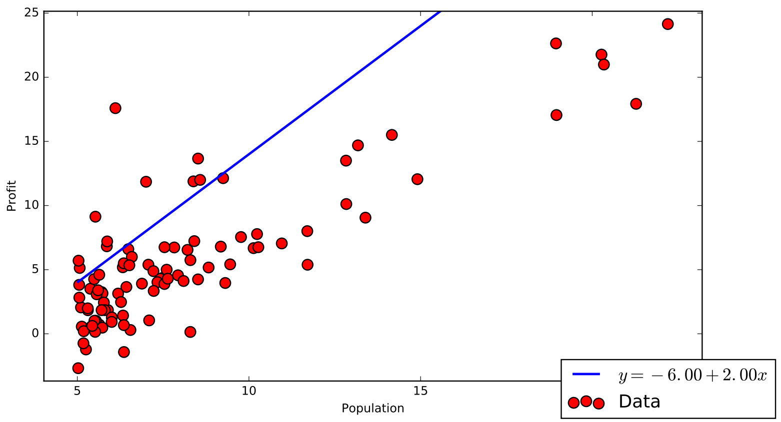
where m is the number of data points in the training set.

The cost (or loss) function



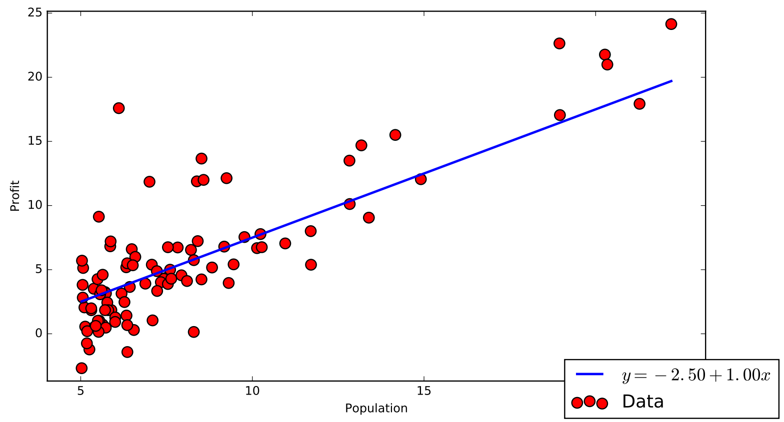
$$J\left(\begin{bmatrix} -5.00 \\ 1.50 \end{bmatrix}\right) = 6.1561$$

The cost (or loss) function



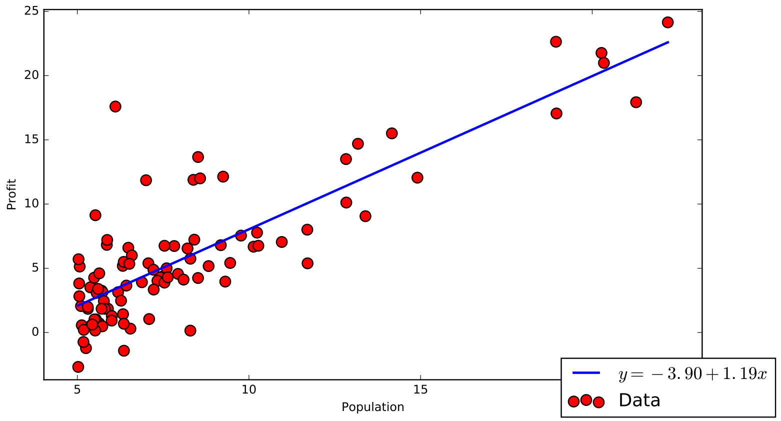
$$J\left(\begin{bmatrix} -6.00 \\ 2.00 \end{bmatrix}\right) = 19.3401$$

The cost (or loss) function



$$J\left(\begin{bmatrix} -2.50 \\ 1.00 \end{bmatrix}\right) = 4.7692$$

The cost (or loss) function



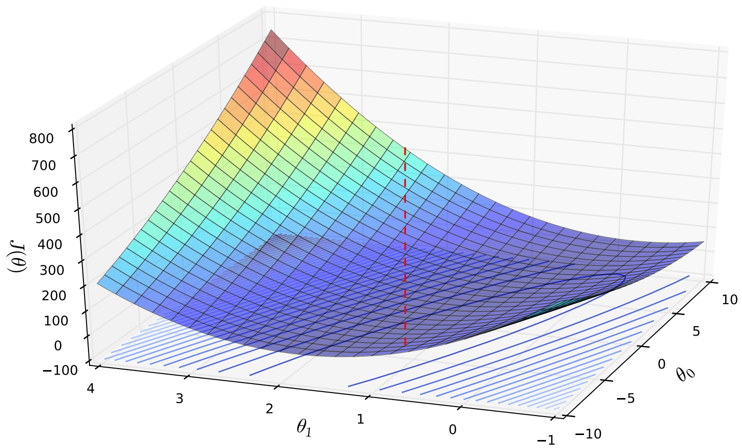
$$J(\begin{bmatrix} -3.90 \\ 1.19 \end{bmatrix}) = 4.4775$$

The cost (or loss) function

So, how do we find $\hat{\theta} = \arg\min_{\theta \in \mathbb{R}^2} J(\theta)$ computationally?

The cost (or loss) function

So, how do we find $\hat{\theta} = \arg\min_{\theta \in \mathbb{R}^2} J(\theta)$ computationally?



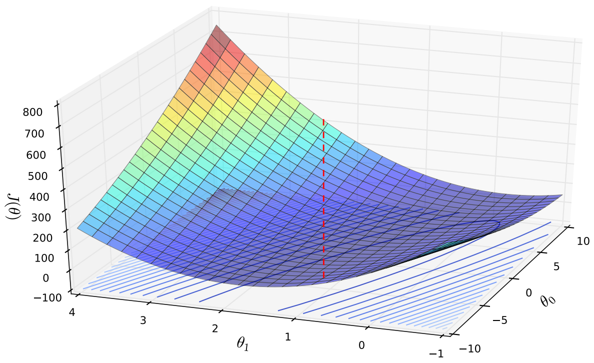
(Stochastic) gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j$$

(Stochastic) gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j$$

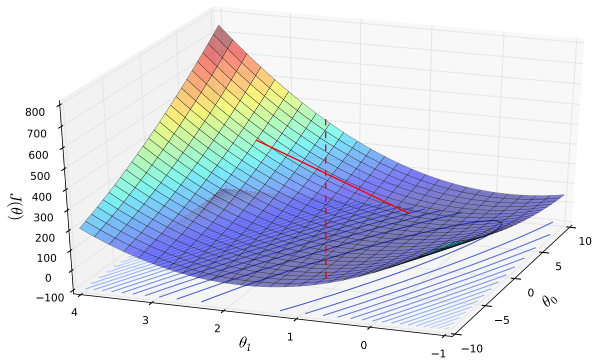
Step 0, $\alpha = 0.01$



(Stochastic) gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j$$

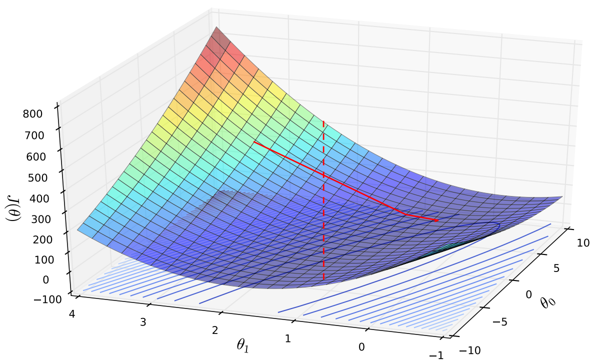
Step 1, $\alpha = 0.01$



(Stochastic) gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j$$

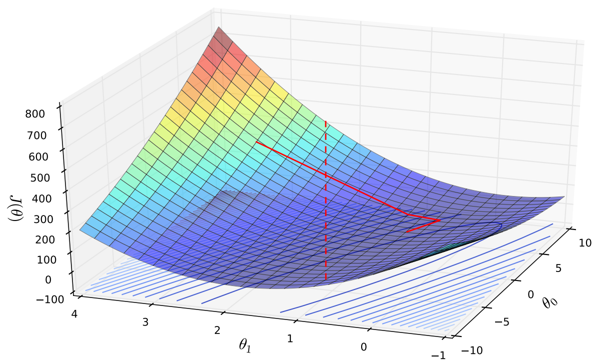
Step 20, $\alpha = 0.01$



(Stochastic) gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j$$

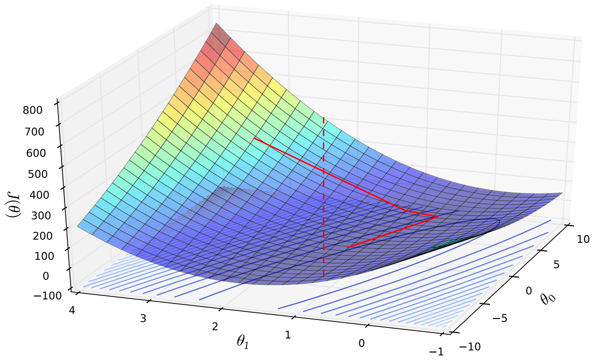
Step 200, $\alpha = 0.01$



(Stochastic) gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j$$

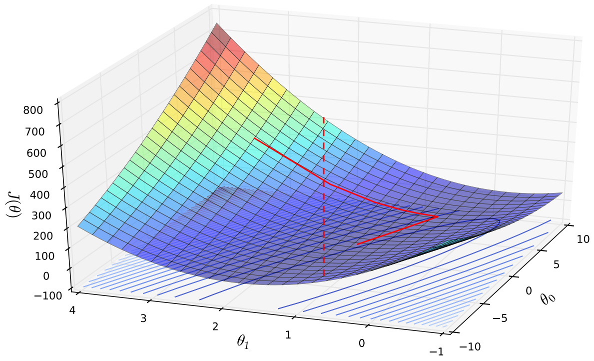
Step 10000, $\alpha = 0.01$



(Stochastic) gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j$$

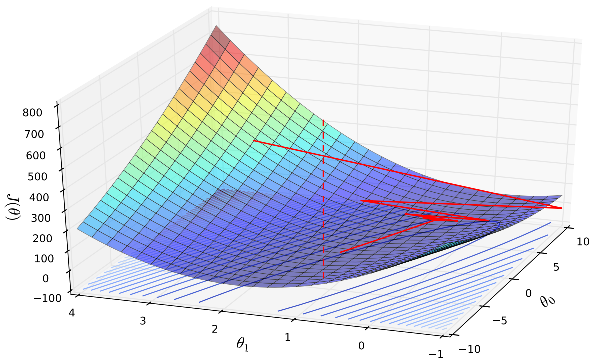
Step 10000, $\alpha = 0.005$



(Stochastic) gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j$$

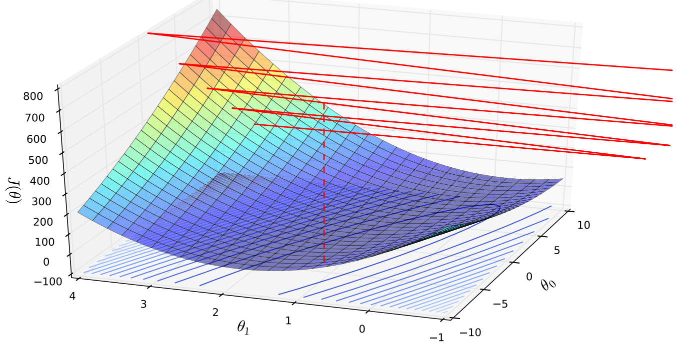
Step 10000, $\alpha = 0.02$



(Stochastic) gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j$$

Step 10, $\alpha = 0.025$



Backpropagation

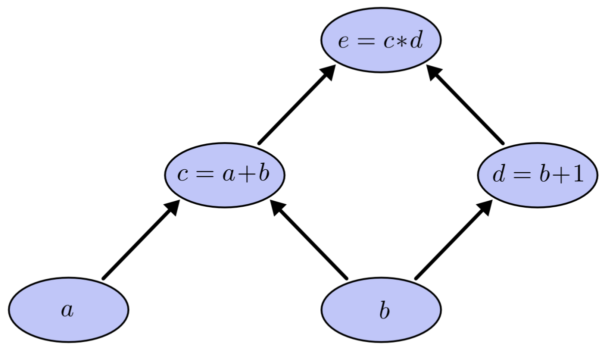
How do we calculate $\frac{\partial}{\partial \theta_j} J(\theta)$?

In other words:
how sensitive is the loss function to the change of a parameter θ_j?

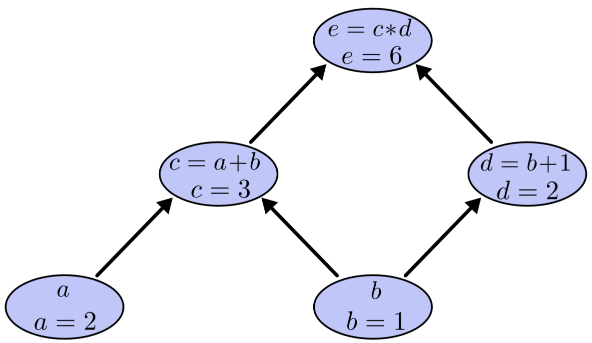
why backpropagation?

we could do this by hand for linear regression...
but what about complex functions?
$\rightarrow$ *propagate error backward*
(special case of *automatic differentiation*)

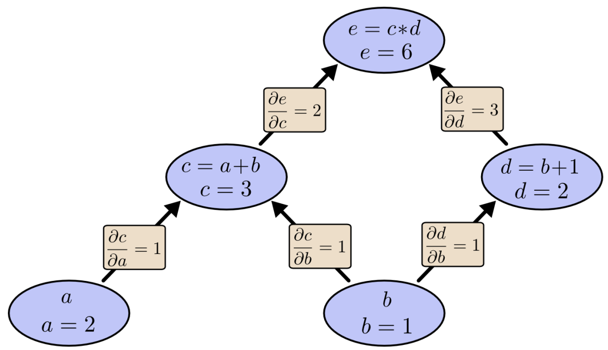
Computation Graphs



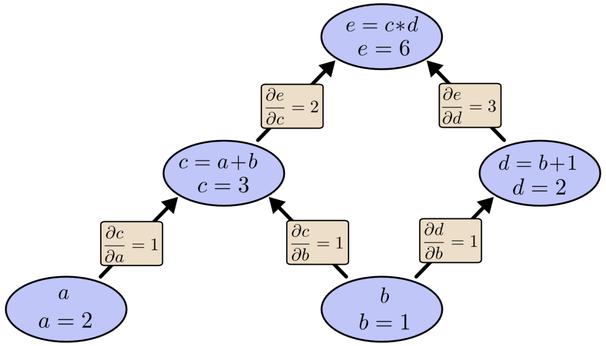
Computation Graphs



Computation Graphs



Computation Graphs

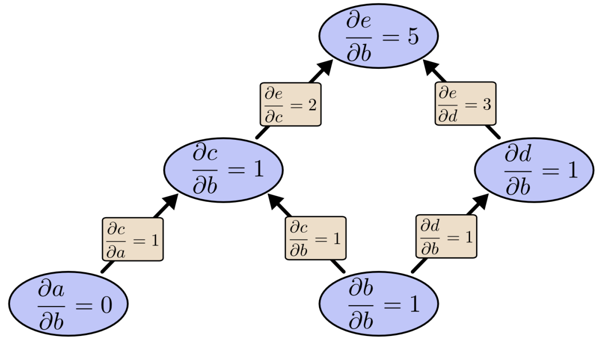


applying chain rule:

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = 1 \cdot 2 + 1 \cdot 3 = 5$$

next, let's use *dynamic programming*
to avoid re-computing intermediate results...

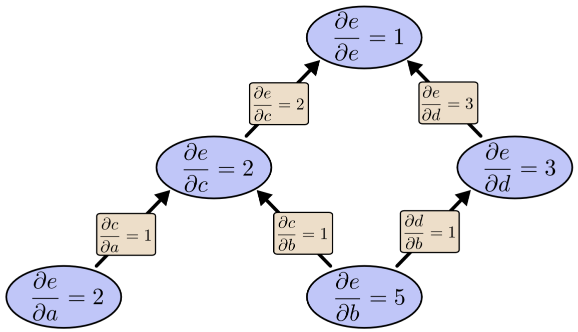
Backpropagation



forward-mode differentiation lets us compute partial derivatives $\frac{\partial x}{\partial b}$ for all nodes x
$\rightarrow$ still inefficient if you have many inputs

Christopher Olah http://colah.github.io/posts/2015-08-Backprop/

Backpropagation



backward-mode differentiation lets us efficiently compute $\frac{\partial e}{\partial x}$ for all inputs x in one pass
$\rightarrow$ also known as *error backpropagation*

Christopher Olah http://colah.github.io/posts/2015-08-Backprop/

To summarize what we have learned

When approaching a machine learning problem, we need:

To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;

To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
- a suitable cost (or loss) function;

To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
- a suitable cost (or loss) function;
- an optimization algorithm;

To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
- a suitable cost (or loss) function;
- an optimization algorithm;
- the gradient(s) of the cost function (if required by the optimization algorithm).

To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model; (here: a linear model)
- a suitable cost (or loss) function; (here: mean square error)
- an optimization algorithm; (here: a variant of SGD)
- the gradient(s) of the cost function (if required by the optimization algorithm).

What is a Neural Network?

- A complex non-linear function which:
  - is built from simpler units (neurons, nodes, gates, . . .)
  - maps vectors/matrices to vectors/matrices
  - is parameterised by vectors/matrices

What is a Neural Network?

- A complex non-linear function which:
  - is built from simpler units (neurons, nodes, gates, . . .)
  - maps vectors/matrices to vectors/matrices
  - is parameterised by vectors/matrices
- Why is this useful?
  - very expressive
  - can represent (e.g.) parameterised probability distributions
  - evaluation and parameter estimation can be built up from components

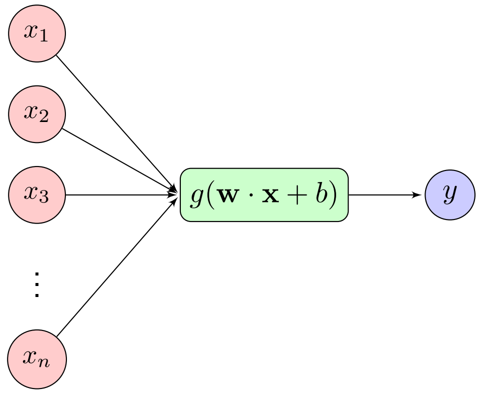
What is a Neural Network?

- A complex non-linear function which:
  - is built from simpler units (neurons, nodes, gates, . . .)
  - maps vectors/matrices to vectors/matrices
  - is parameterised by vectors/matrices
- Why is this useful?
  - very expressive
  - can represent (e.g.) parameterised probability distributions
  - evaluation and parameter estimation can be built up from components

relationship to linear regression

- more complex architectures with *hidden* units (neither input nor output)
- neural networks typically use non-linear activation functions

An Artificial Neuron



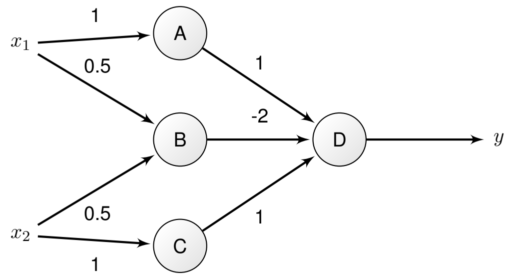
- $\mathbf{x}$ is a vector input, y is a scalar output
- $\mathbf{w}$ and b are the *parameters* (b is a *bias* term)
- g is a (non-linear) *activation function*

Why Non-linearity?

Functions like XOR cannot be separated by a *linear* function

XOR
Truth table

x_1	x_2	output
0	0	0
0	1	1
1	0	1
1	1	0



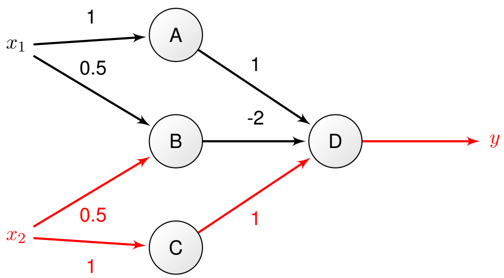
(neurons arranged in layers, and fire if input is ≥ 1)

Why Non-linearity?

Functions like XOR cannot be separated by a *linear* function

XOR
Truth table

x_1	x_2	output
0	0	0
0	1	1
1	0	1
1	1	0



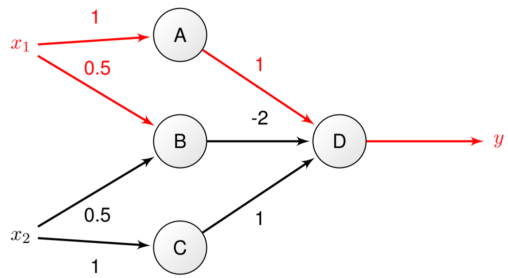
(neurons arranged in layers, and fire if input is ≥ 1)

Why Non-linearity?

Functions like XOR cannot be separated by a *linear* function

XOR
Truth table

x_1	x_2	output
0	0	0
0	1	1
1	0	1
1	1	0



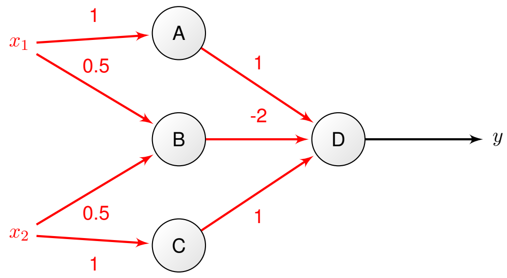
(neurons arranged in layers, and fire if input is ≥ 1)

Why Non-linearity?

Functions like XOR cannot be separated by a *linear* function

XOR
Truth table

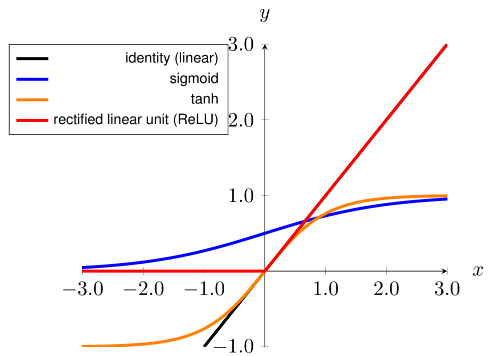
x_1	x_2	output
0	0	0
0	1	1
1	0	1
1	1	0



(neurons arranged in layers, and fire if input is ≥ 1)

Activation functions

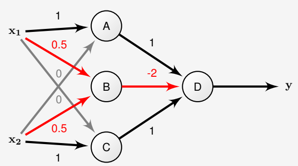
- desirable:
  - differentiable (for gradient-based training)
  - monotonic (for better training stability)
  - non-linear (for better expressivity)



A Simple Neural Network: Maths

we can use linear algebra to formalize our neural network:

the network



$$w_1 = \begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \\ 0 & 1 \end{bmatrix} \quad h_1 = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$w_2 = \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} \quad y = \begin{bmatrix} D \end{bmatrix}$$

calculation of $x \mapsto y$

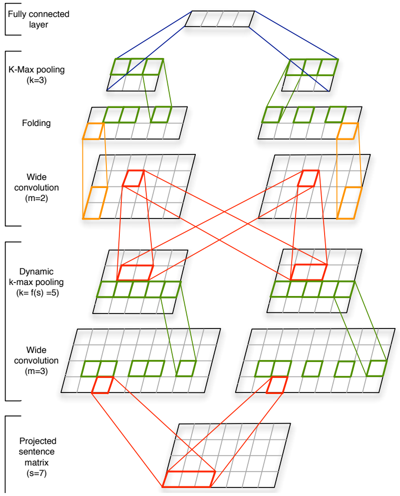
$$h_1 = \varphi(xw_1)$$
$$y = \varphi(h_1 w_2)$$

A Simple Neural Network: Python Code

```python
import numpy as np

#activation function
def phi(x):
    return np.greater_equal(x,1).astype(int)

def nn(x, w1, w2):
    h1 = phi(np.dot(x, w1))
    y  = phi(np.dot(h1, w2))
    return y

w1 = np.array([ [1, 0.5, 0], [0, 0.5, 1] ])
w2 = np.array([[1], [-2], [1]])
x  = np.array([1, 0])
print nn(x, w1, w2)
```

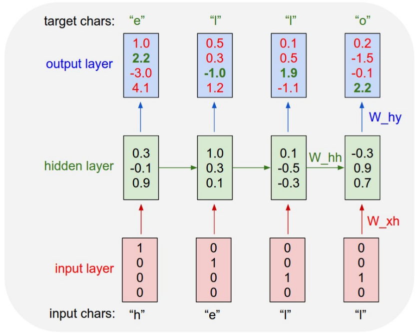
More Complex Architectures

Convolutional



[Kalchbrenner et al., 2014]

Recurrent



Andrej Karpathy

`http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

Practical Considerations

- efficiency:
  - GPU acceleration of BLAS operations
  - perform SGD in mini-batches
- hyperparameters:
  - number and size of layers
  - minibatch size
  - learning rate
  - ...
- initialisation of weight matrices
- stopping criterion
- regularization (dropout)
- bias units (always-on input)

Toolkits for Neural Networks

What does a Toolkit Provide

- Multi-dimensional matrices (tensors)
- Automatic differentiation
- Efficient GPU routines for tensor operations

torch

Torch `http://torch.ch/`

TensorFlow `https://www.tensorflow.org/`

theano

Theano `http://deeplearning.net/software/theano/`

There are many more!

Further Reading

- required reading: Koehn (2017), chapter 13.2-3.
- further reading on backpropagation:
  `http://colah.github.io/posts/2015-08-Backprop/`

Slide Credits

some slides borrowed from:

- Sennrich, Birch, and Junczys-Dowmunt (2016): Advances in Neural Machine Translation
- Sennrich and Haddow (2017): Practical Neural Machine Translation

Bibliography I

Kalchbrenner, N., Grefenstette, E., and Blunsom, P. (2014).
A Convolutional Neural Network for Modelling Sentences.
In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).