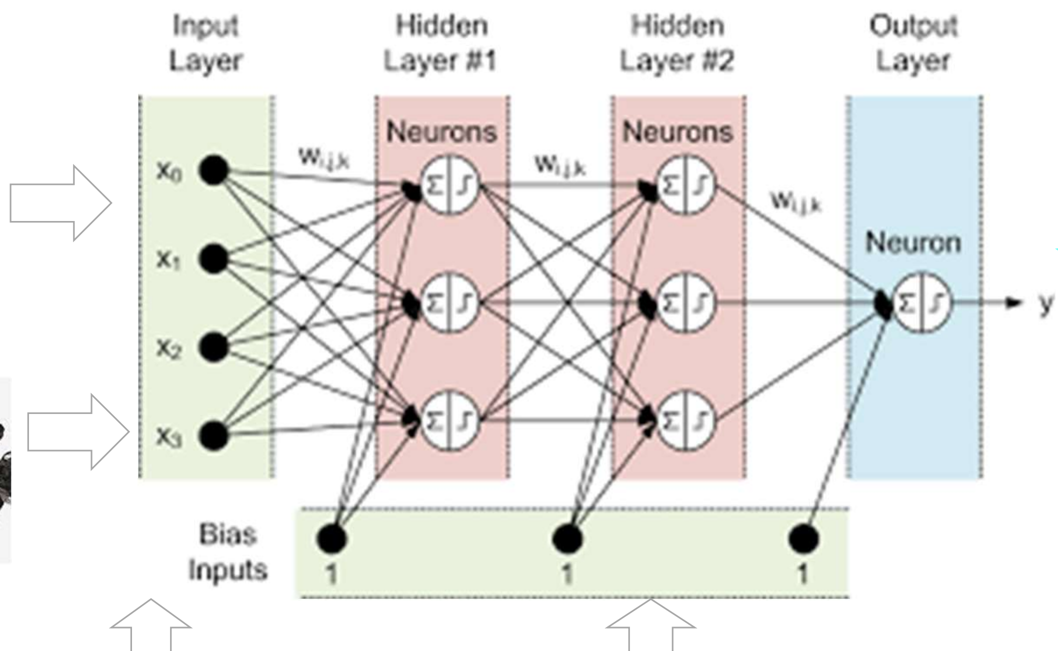
The deep learning boom (2011---)

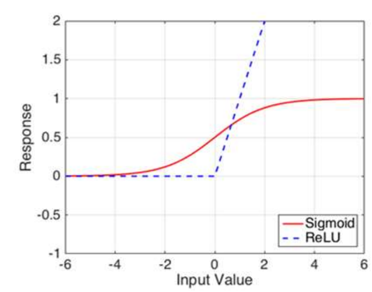
LOTS of training data



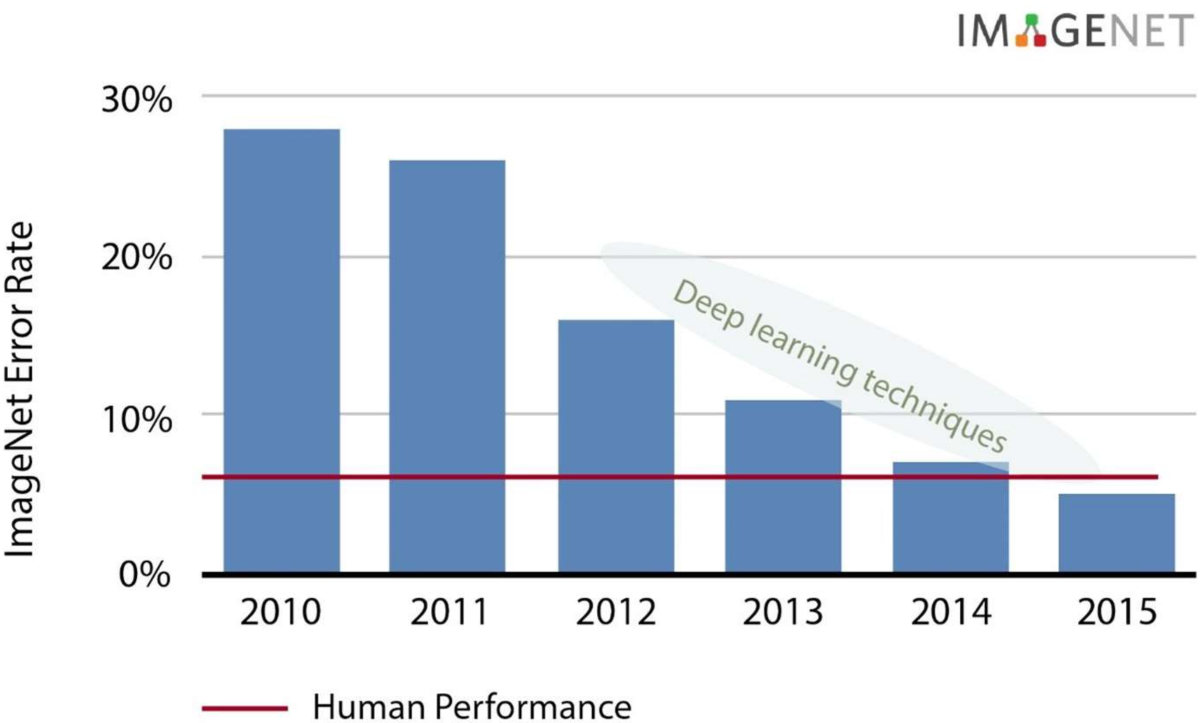
Record-breaking accuracy in some difficult tasks !

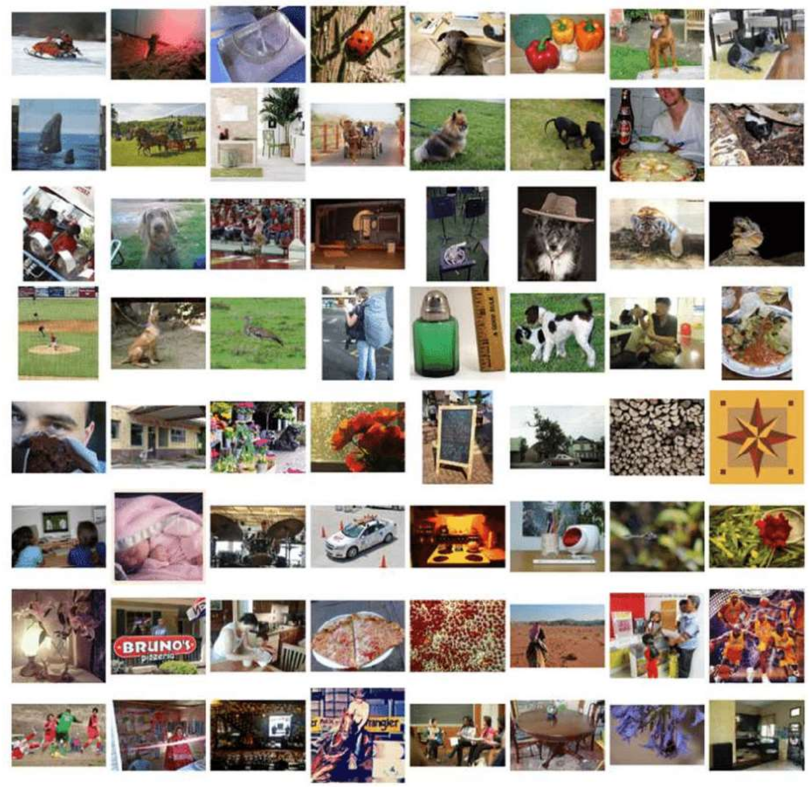
Smart optimizers: stochastic gradient descent

Engineering: ReLU Vs Sigmoid, Dropout, Deeper, Wider, ResNets



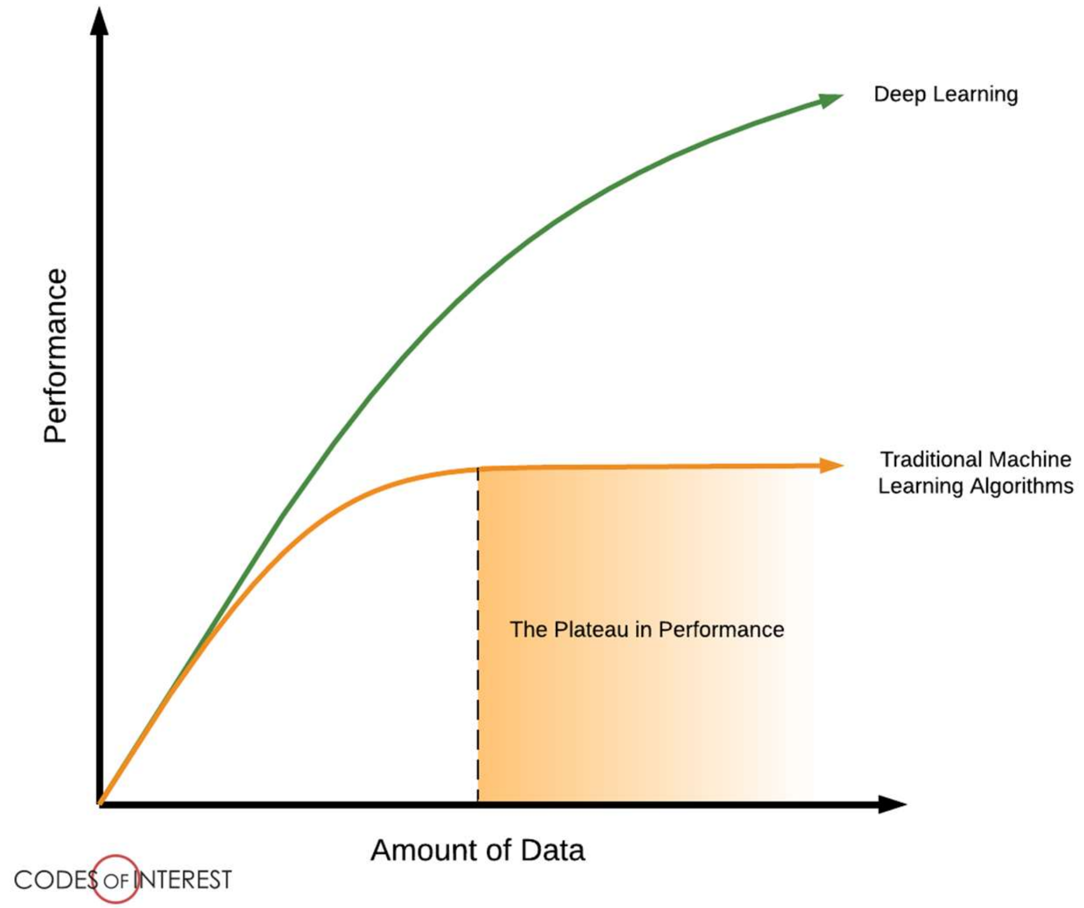
Success stories: vision





What changed with Deep Learning?

The ability to process huge amounts of data and continue to accrue accuracy gains from them.



Training a Feed-forward network

- To train a neural network, define a loss function $L(y,\hat{y})$:
  a function of the true output y and the predicted output $\hat{y}$
- $L(y,\hat{y})$ assigns a non-negative numerical score to the neural network's output, $\hat{y}$
- The parameters of the network are set to minimise L over the training examples (i.e. a sum of losses over different training samples)
- L is typically minimised using a *gradient-based method*

Recap: gradient-descent training algorithm

$$\min_{W} F(W) = \min_{W} \sum_{i=1}^{N} \frac{(y^i - \underbrace{NN_w(x^i)}_{\hat{y}^i})^2}{N} \leftarrow \text{square error}$$

$$\Leftarrow \min_{W} \sum_{i=1}^{N} L(y^i, NN_w(x^i))$$

- Choose an arbitrary initial point : $W^0 \leftarrow$ randomly.
- λ = Chosen learning rate
- Epoch t = 0
- While stopping criteria not reached /* $\|\nabla_W F(W)\| \leq \varepsilon$ */

Compute gradient:

$$\rightarrow \nabla F(W)\big|_{W=W^t}$$

Update parameters:

$$W^{t+1} \leftarrow W^t - \lambda \nabla F(W^t)$$

$$t = t+1$$

Stochastic gradient descent

- Stochastic approximation to gradient descent optimization when applied over sum of errors on several i.i.d training examples

identically and independently distributed samples.

- Typical training objective:
  - $L(w) = \frac{1}{N}\sum_{i=1}^{N} L(f(x^i; w), y_i)$
  - True gradient:

$$\nabla L(w) = \frac{1}{N}\sum_{i=1}^{N} \nabla L(f(x^i, w), y_i)$$

  - Stochastic approximation:

randomly choose an (x^i, y^i) from $D \equiv \{(x^1,y^1), \ldots, (x^N, y^N)\}$

$$\tilde{\nabla} L(w) = \nabla_w L(f(x^i, w), y^i)$$

$$w^{t+1} \leftarrow w^t - \lambda \tilde{\nabla} L(w)$$

More generally sample B example $B = \{(x^{i_1}, y^{i_1}), \ldots, (x^{i_B}, y^{i_B})\}$

- More efficient than full batch
- Empirical found to be better at optimizing non-convex functions because of noisy nature of gradients.

Demo

- Difference between SGD and GD
- https://colab.research.google.com/drive/104UVC56ZKVAt0HDGQyqv5lsg_PsEewRK?usp=sharing

Stochastic Gradient Descent (SGD) for training NN

SGD Algorithm

Inputs:

Function NN(x; w), Training examples, $x_1 \dots x_n$ and outputs, $y_1 \dots y_n$ and Loss function L.

do until stopping criterion

Pick a training example x_i, y_i

Compute the loss $L(NN(x_i; w), y_i)$

Compute gradient of L, ∇L with respect to w

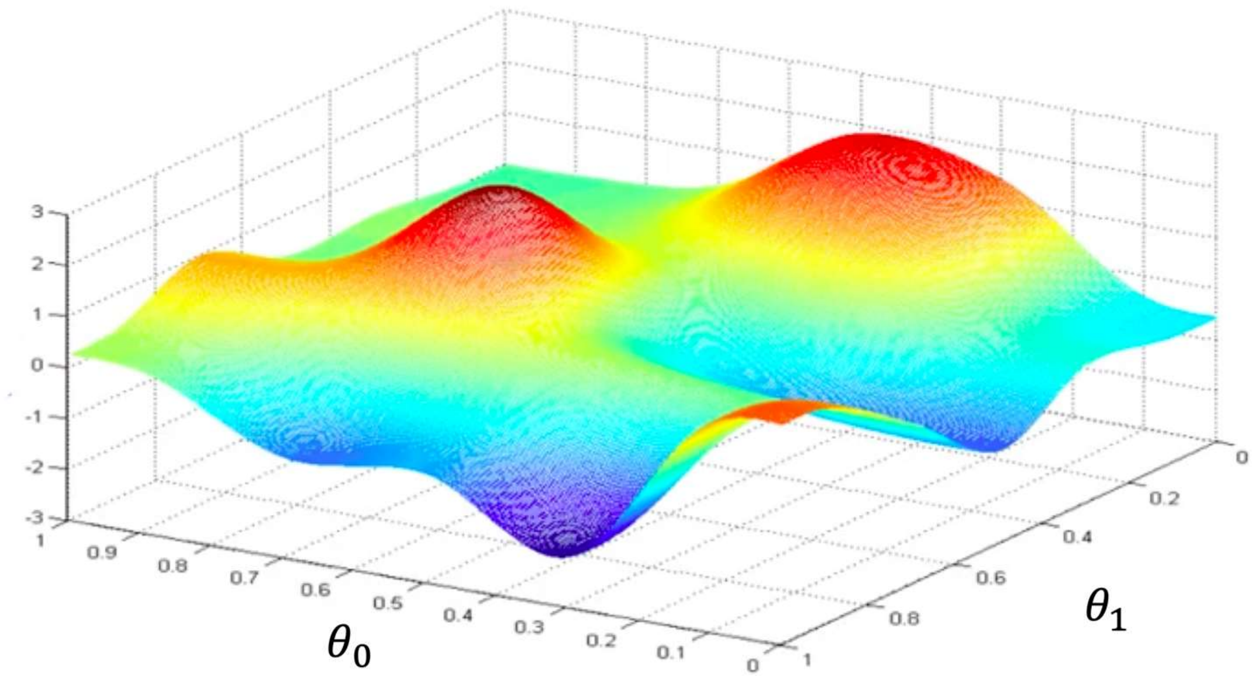
$w \leftarrow w - \eta \nabla L$

done

Return: w

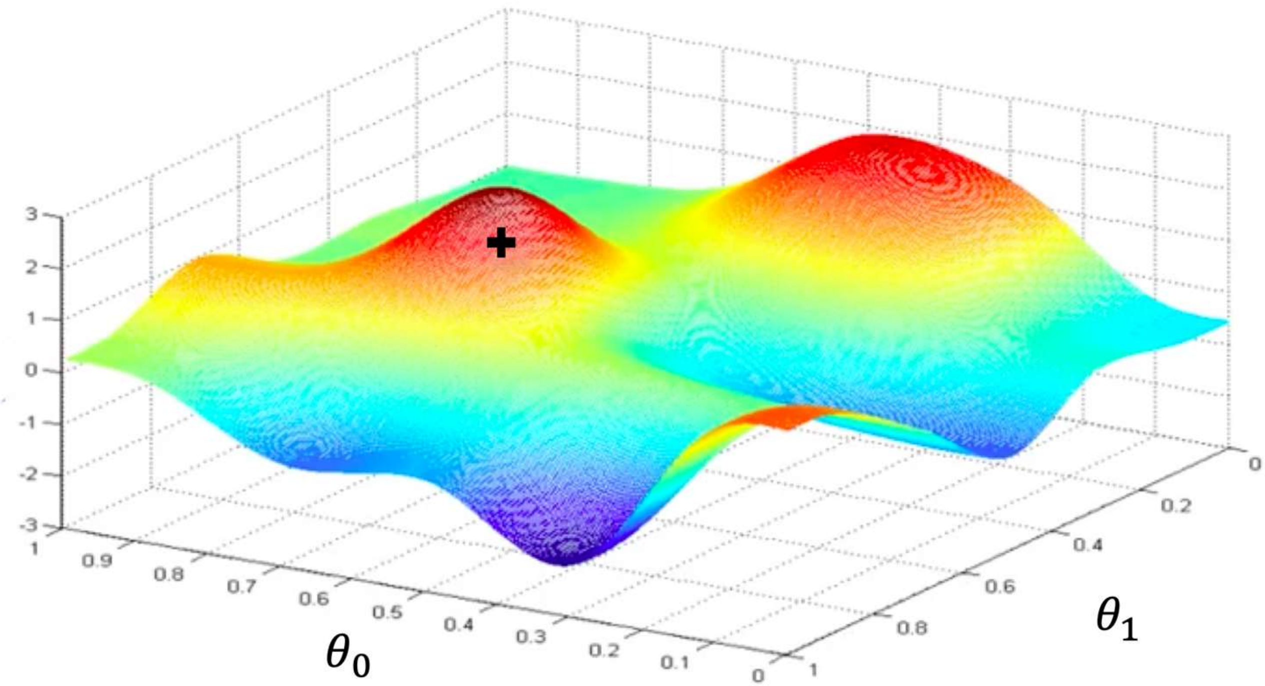
Loss Optimization

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta})$$



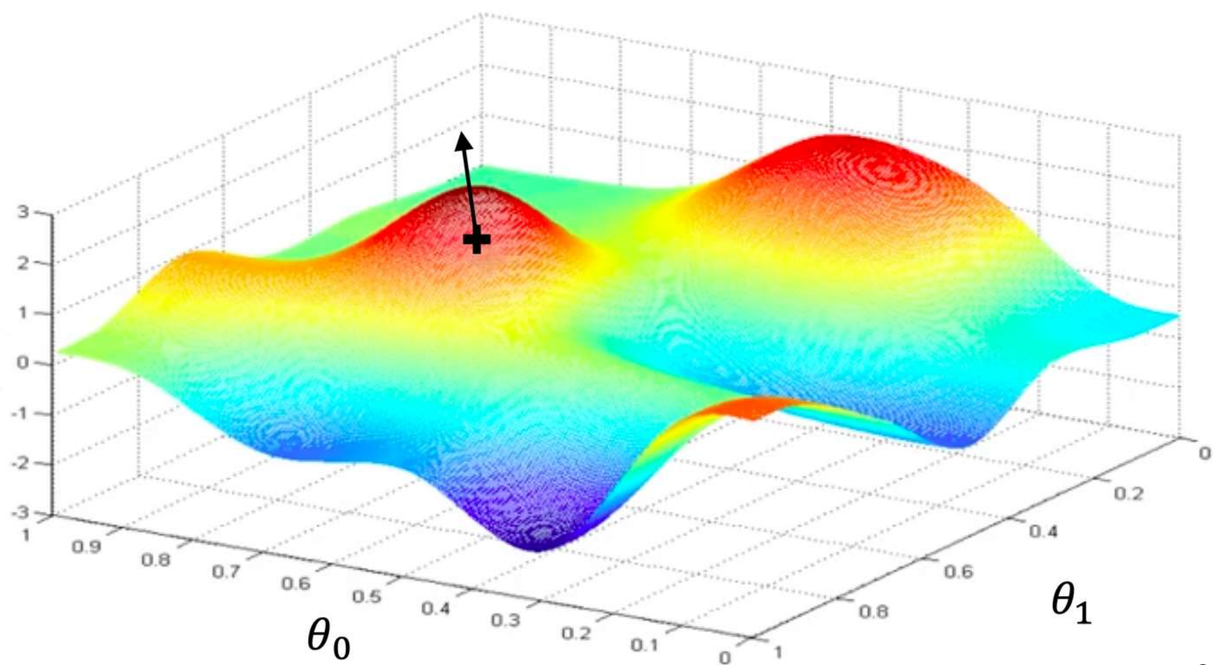
Loss Optimization

Randomly pick an initial (θ_0, θ_1)



Loss Optimization

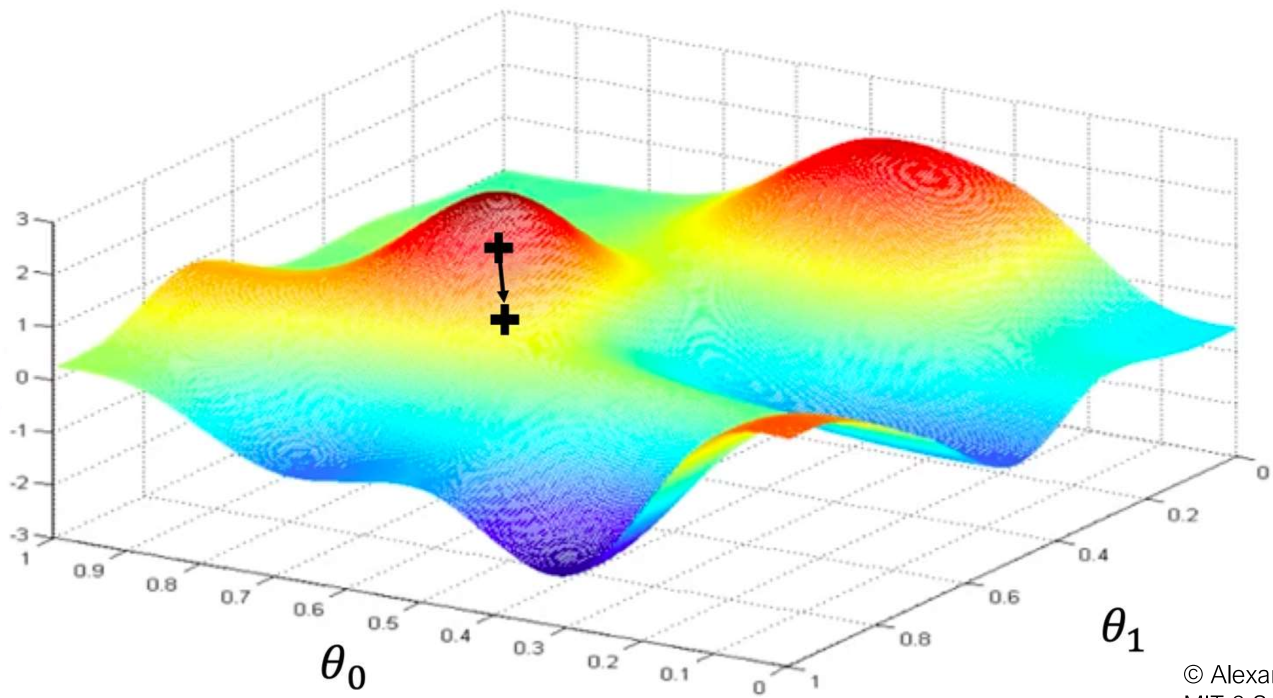
Compute gradient, $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$



MIT 6.S191: Introduction to Deep Learning
IntroToDeepLearning.com

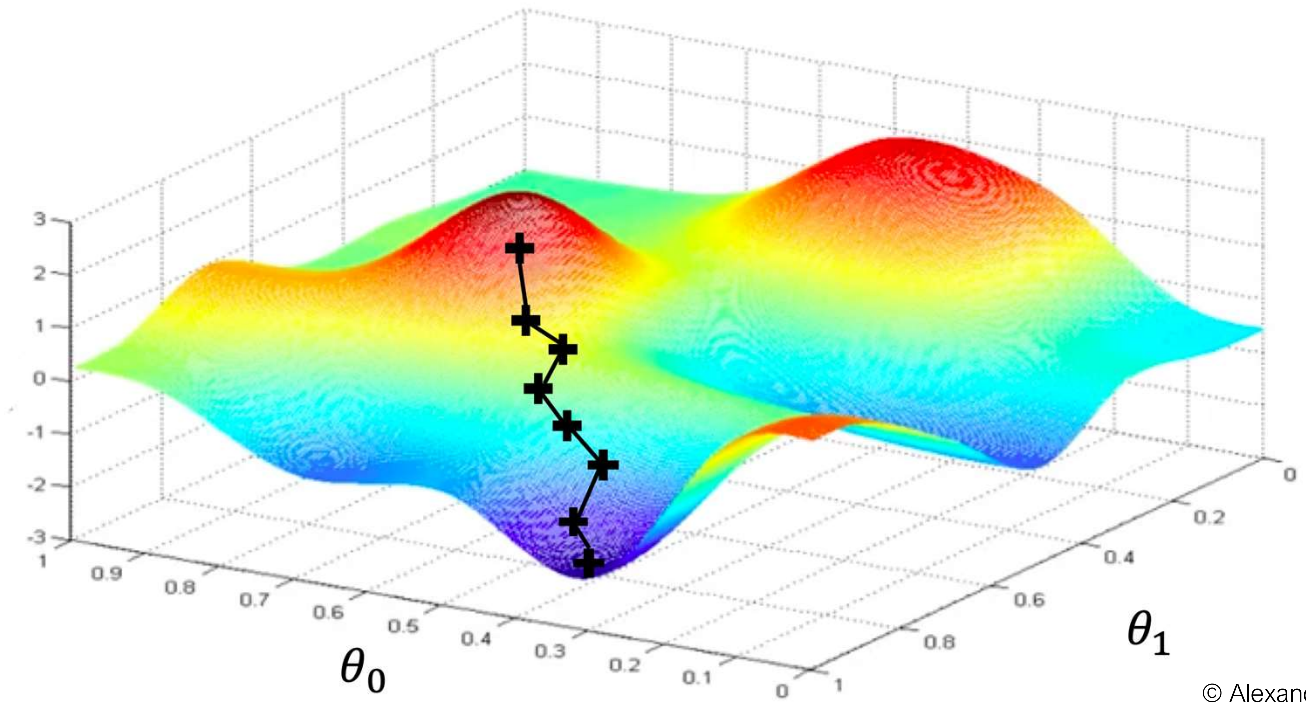
Loss Optimization

Take small step in opposite direction of gradient



Loss Optimization

Repeat until convergence



Computing gradients

Modern machine learning libraries come packaged with software for automatically computing gradients.

However, to get a deeper understanding of the working of neural networks we will study the well-known backpropagation algorithm for computing the gradients manually.

Simple one neuron network: logistic regression

Training a Neural Network

Define the Loss function to be minimised as a node L

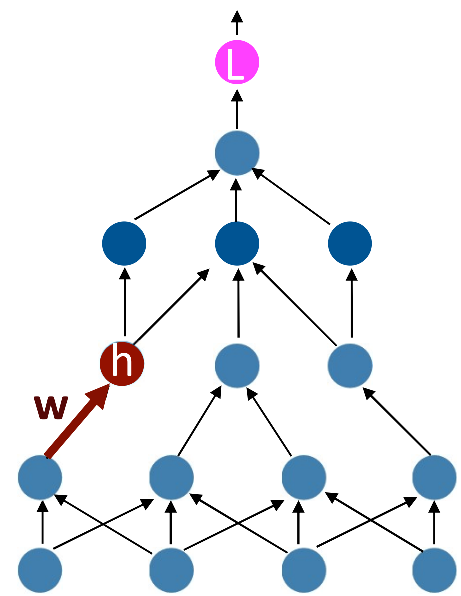
Goal: Learn weights for the neural network which minimise L

Gradient Descent: Find $\partial L/\partial w$ for every weight w, and update it as
$w \leftarrow w - \eta\ \partial L/\ \partial w$

How do we efficiently compute $\partial L/\partial w$ for all w?

Will compute $\partial L/\partial h$ for every node h in the network!

$\partial L/\partial w = \partial L/\partial h\ \cdot \partial h/\partial w$ where h is the node which uses w



Computing the gradients

New goal: compute $\partial L/\partial h$ for every node h in the network

Simple algorithm: Backpropagation

Key fact: Chain rule of differentiation

If L can be written as a function of variables $v_1,\ldots, v_n$, which in turn depend (partially) on another variable h, then

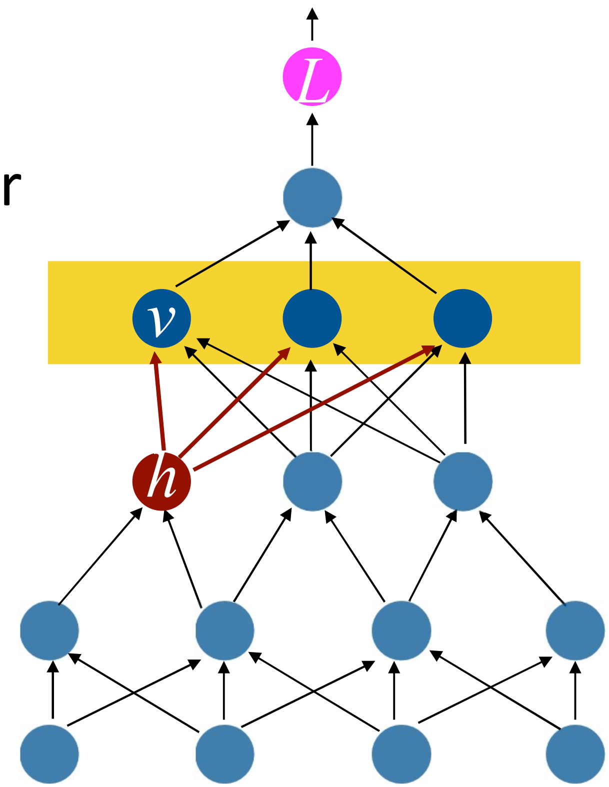
$$\partial L/\partial h = \Sigma_i \, \partial L/\partial v_i \cdot \partial v_i/\partial h$$

Backpropagation

If L can be written as a function of variables $v_1, \ldots, v_n$, which in turn depend (partially) on another variable h, then

$$\partial L/\partial h = \Sigma_i \, \partial L/\partial v_i \cdot \partial v_i/\partial h$$

Consider $v_1, \ldots, v_n$ as the layer above h, $\Gamma(h)$



Then, the chain rule gives

$$\partial L/\partial h = \Sigma_{v \in \Gamma(h)} \, \partial L/\partial v \cdot \partial v/\partial h$$

Backpropagation

$$\partial L/\partial h = \Sigma_{v \in \Gamma(h)}\ \partial L/\partial v \cdot \partial v/\partial h$$

Backpropagation

Base case: $\partial L/\partial L = 1$

For each h (top to bottom):

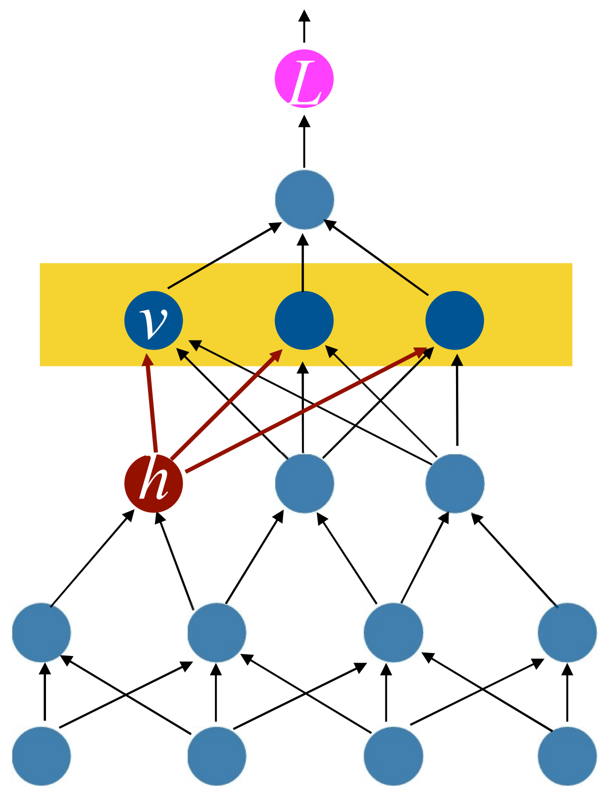
For each $v \in \Gamma(h)$:

Inductively, have computed $\partial L/\partial v$

Directly compute $\partial v/\partial h$

Compute $\partial L/\partial h$

Compute $\partial L/\partial w$
where $\partial L/\partial w = \partial L/\partial h \cdot \partial h/\partial w$



Forward Pass

First, in a forward pass, compute values of all nodes given an input
(The values of each node will be needed during backprop)

Where values computed in the forward pass are needed