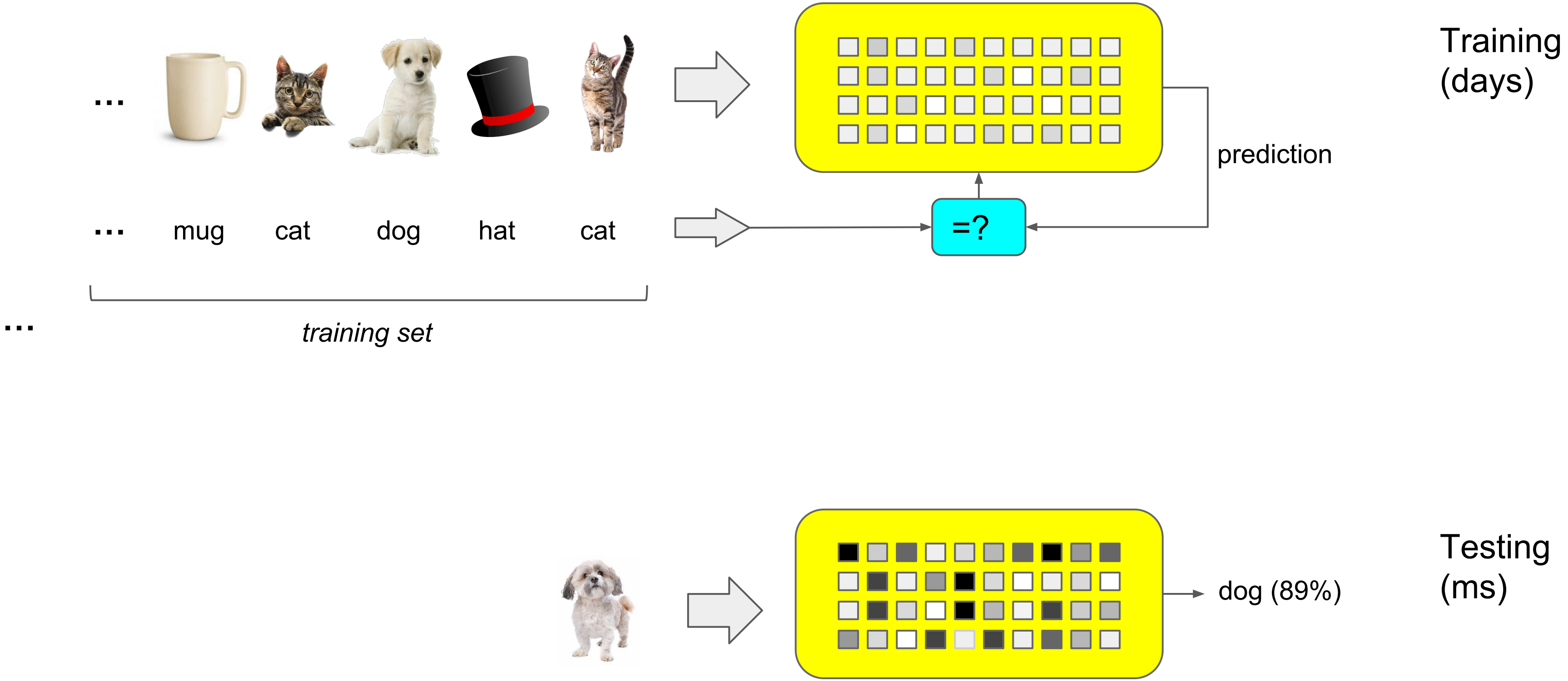
Deep Learning

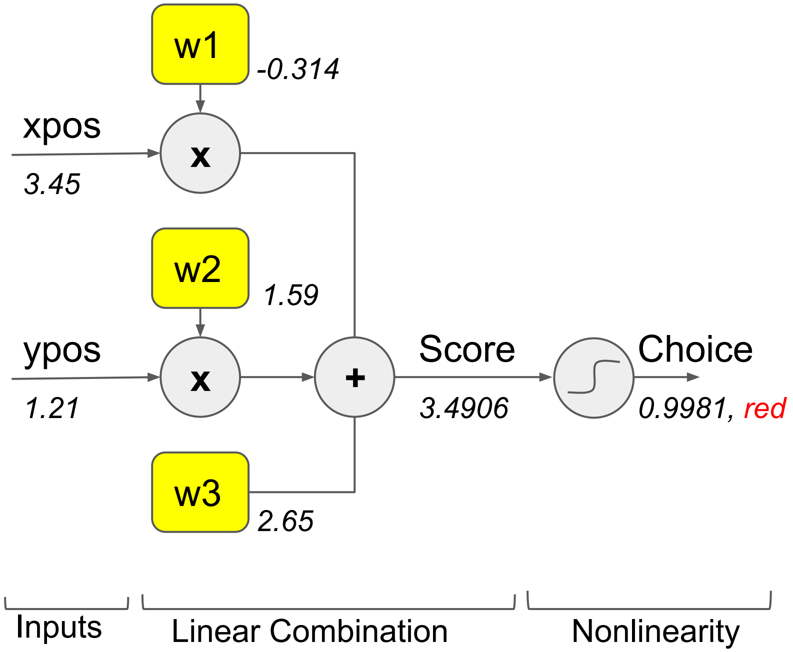
Go deep or go home

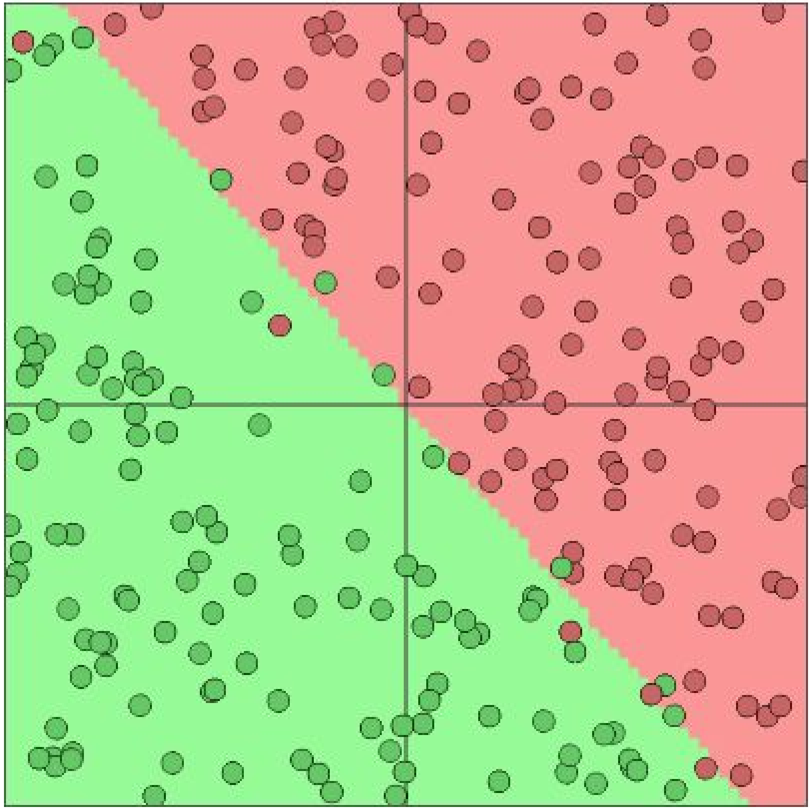
Supervised Learning



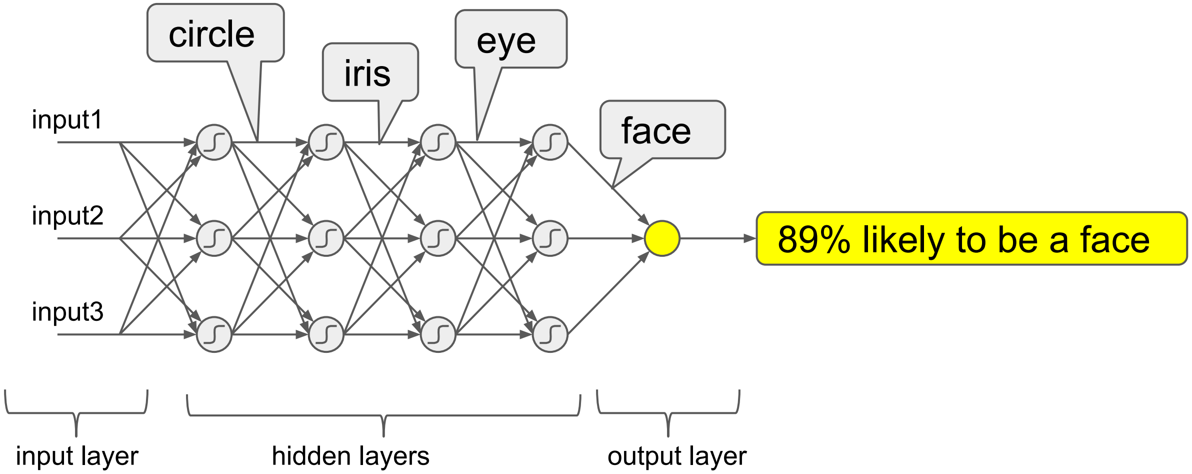
Single Neuron to Neural Networks

Single Neuron





More Neurons, More Layers: Deep Learning



How to find the best values for the weights?

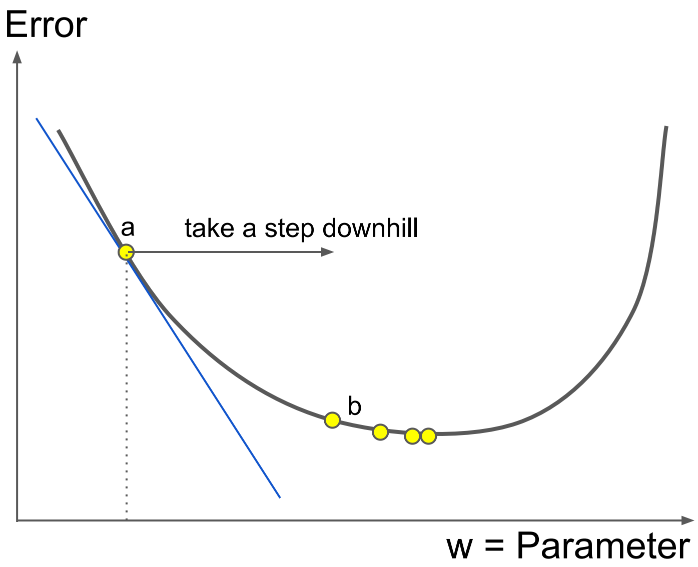
Define **error = |expected - computed|** ^{2}

Find parameters that **minimize** average error

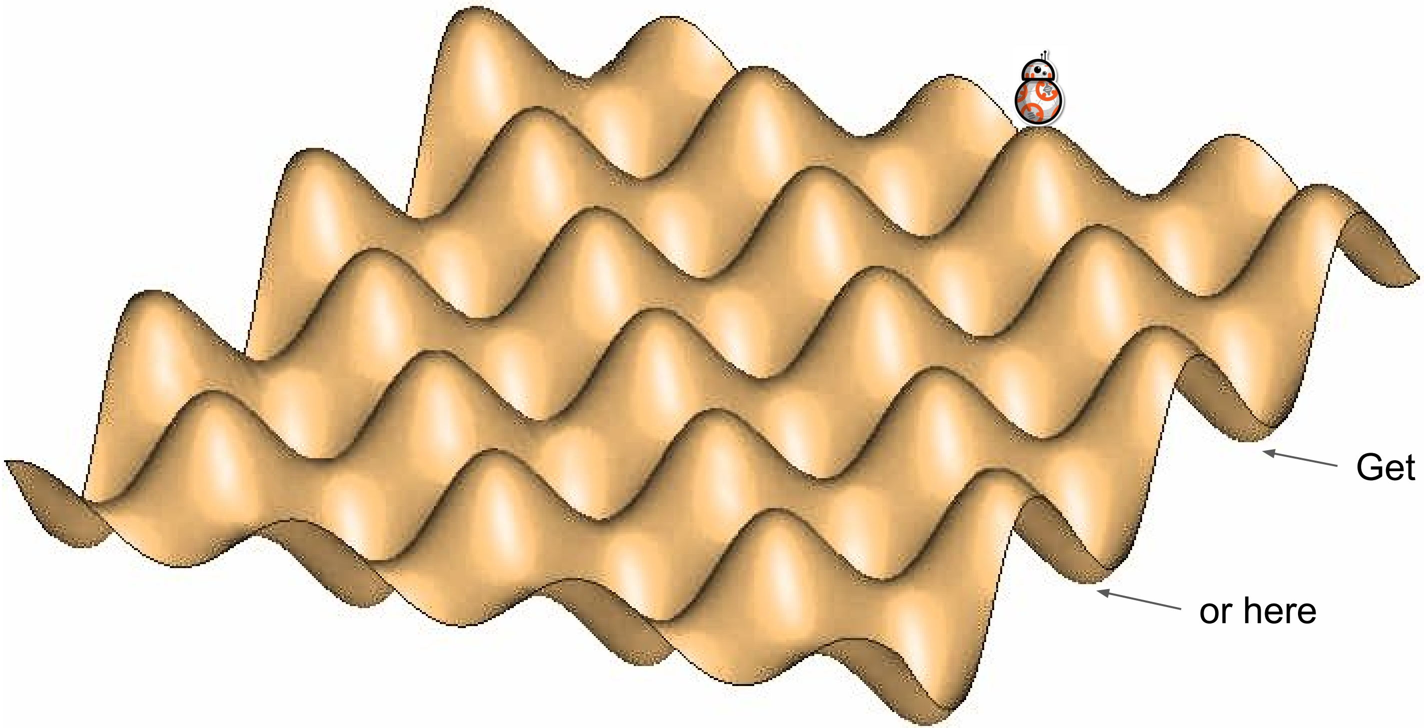
Gradient of error wrt w is

$$\frac{\partial\, error}{\partial w} = \nabla_w \, error$$

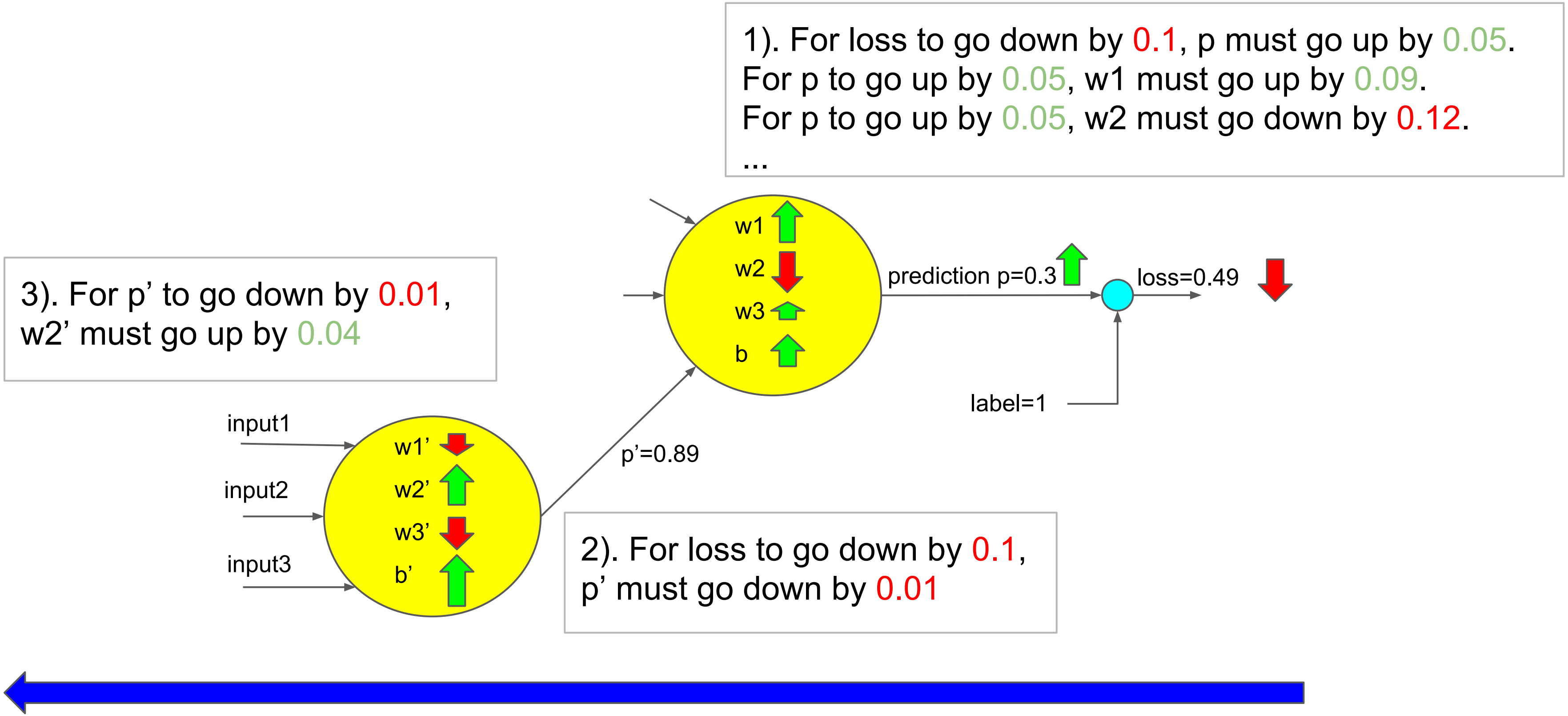
Update: $w = w - step\ .\ \nabla_w$



Egg carton in 1 million dimensions



Backpropagation: Assigning Blame



Stochastic Gradient Descent (SGD) and Minibatch

It’s too expensive to compute the gradient on all inputs to take a step.

We prefer a quick approximation.

Use a small random sample of inputs (**minibatch**) to compute the gradient.

Apply the gradient to all the weights.

Welcome to SGD, much more efficient than regular GD!

Usually things would get intense at this point...

$$
\begin{aligned}
\frac{\partial s}{\partial W_{ij}^{(1)}} &= \frac{\partial W^{(2)} a^{(2)}}{\partial W_{ij}^{(1)}} = \frac{\partial W_i^{(2)} a_i^{(2)}}{\partial W_{ij}^{(1)}} = W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial W_{ij}^{(1)}} \\
\Rightarrow W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial W_{ij}^{(1)}} &= W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} \\
&= W_i^{(2)} \frac{\sigma(z_i^{(2)})}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} \\
&= W_i^{(2)} \sigma'(z_i^{(2)}) \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} \\
&= W_i^{(2)} \sigma'(z_i^{(2)}) \frac{\partial}{\partial W_{ij}^{(1)}} (b_i^{(1)} + a_1^{(1)} W_{i1}^{(1)} + a_2^{(1)} W_{i2}^{(1)} + a_3^{(1)} W_{i3}^{(1)} + a_4^{(1)} W_{i4}^{(1)}) \\
&= W_i^{(2)} \sigma'(z_i^{(2)}) \frac{\partial}{\partial W_{ij}^{(1)}} (b_i^{(1)} + \sum_k a_k^{(1)} W_{ik}^{(1)}) \\
&= W_i^{(2)} \sigma'(z_i^{(2)}) a_j^{(1)} \\
&= \delta_i^{(2)} \cdot a_j^{(1)}
\end{aligned}
$$

Local gradient

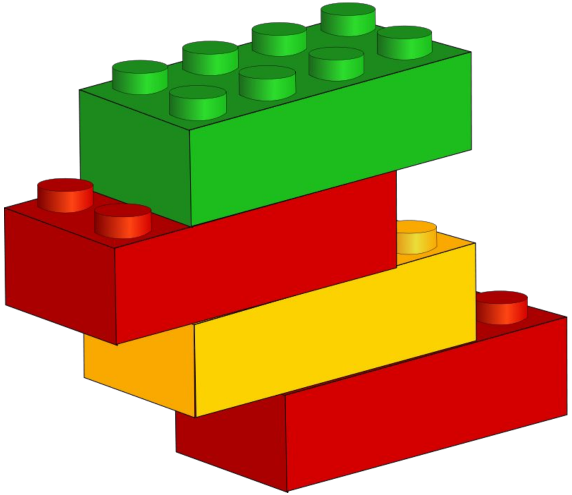
Chain rule

Jacobian matrix

Hessian

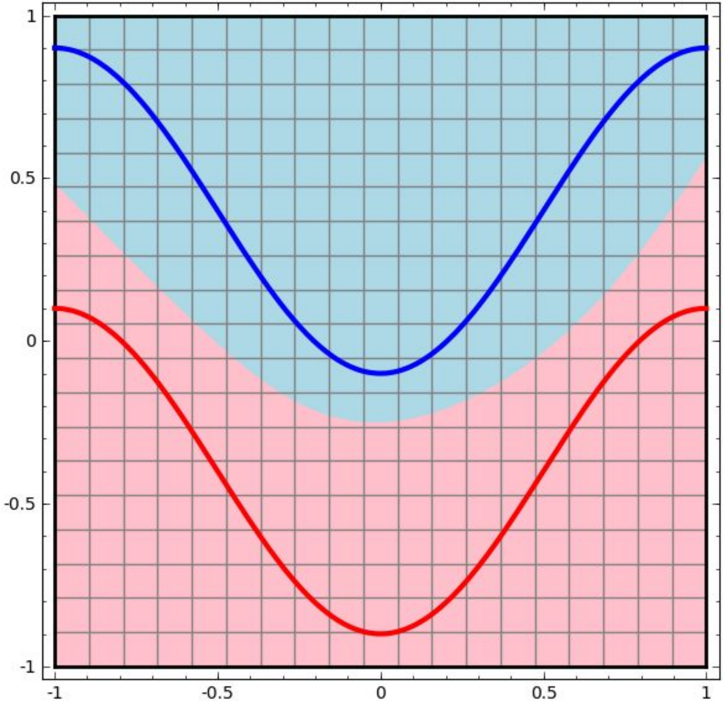
Sum over paths

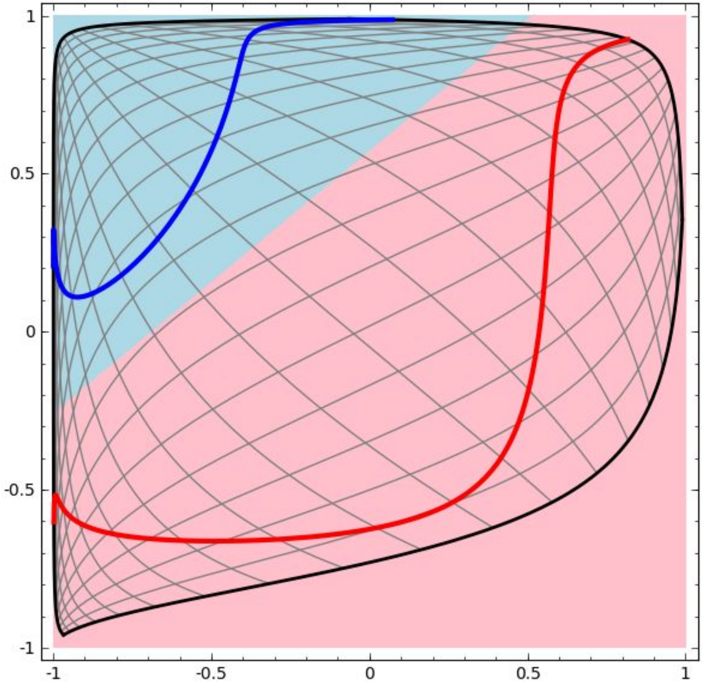
Partial derivatives



Credit: Richard Socher Stanford class cs224d

Learning a new representation

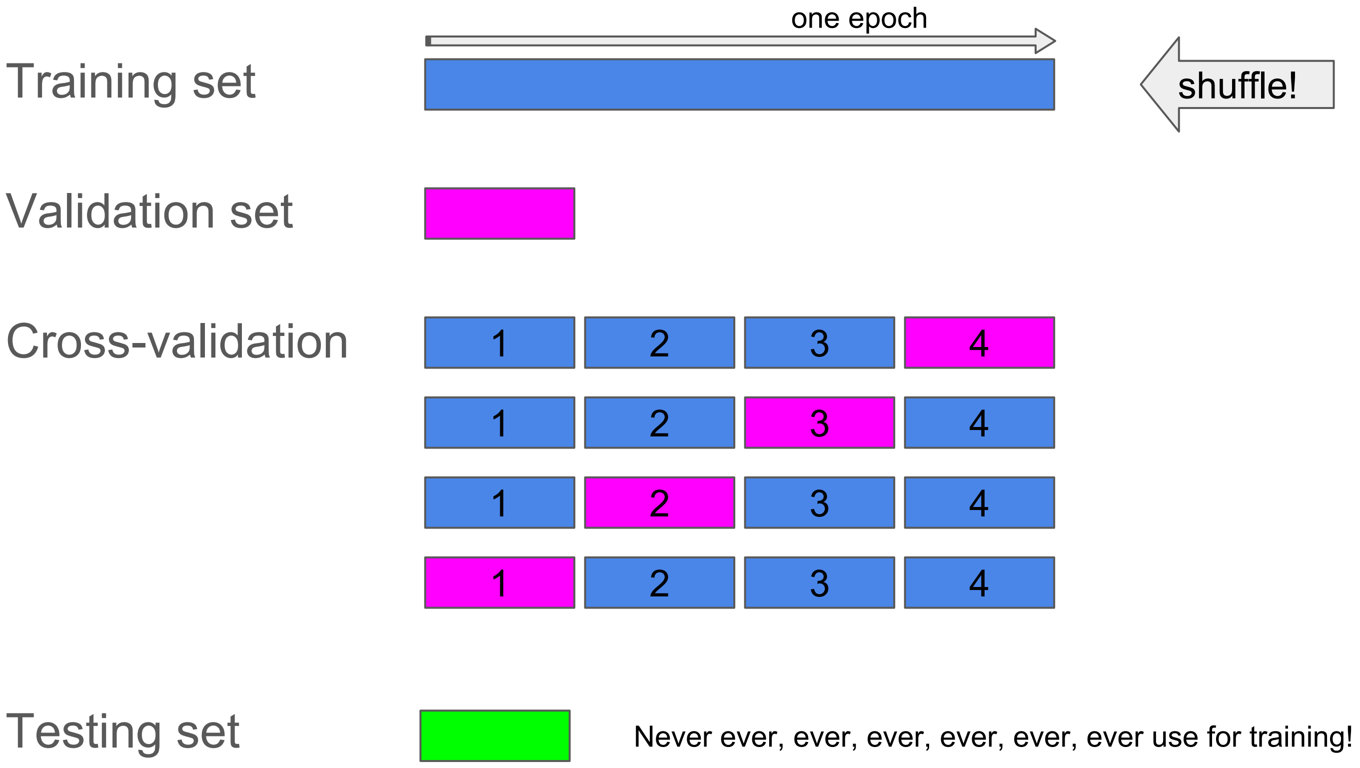




Credit: http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Training Data

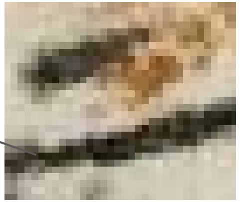
Train. Validate. Test.



Tensors

Tensors are not scary

[r=0.45 g=0.84 b=0.76]



Number: 0D

Vector: 1D

Matrix: 2D

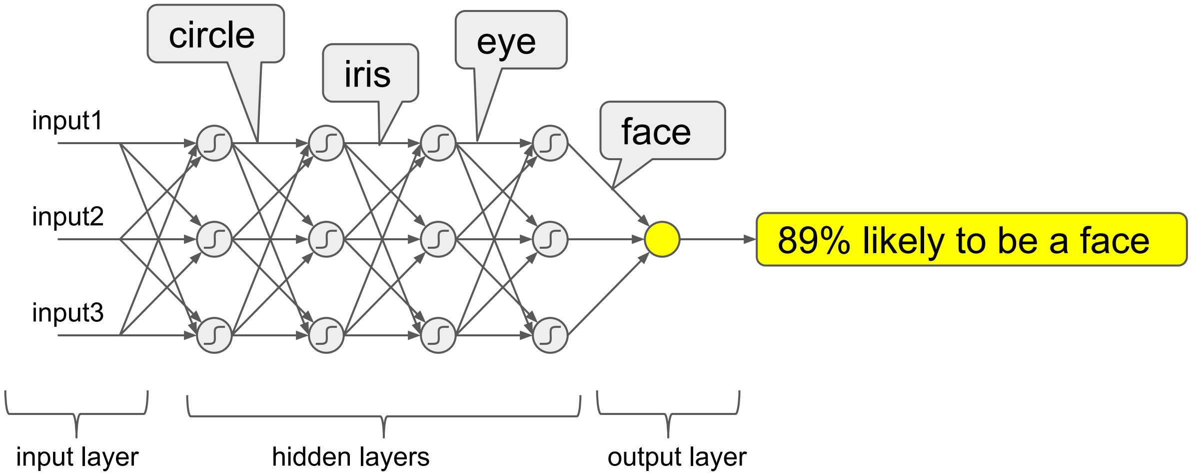
Tensor: 3D, 4D, … array of numbers

This image is a **3264 x 2448 x 3 tensor**

Different types of networks / layers

Fully-Connected Networks

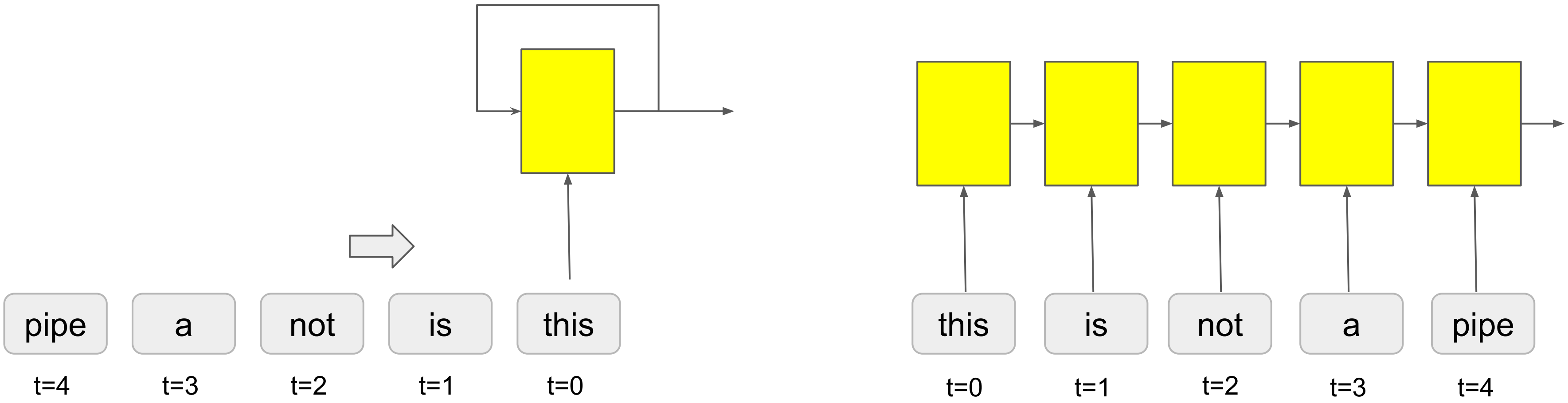
Per layer, every input connects to every neuron.



Recurrent Neural Networks (RNN)

Appropriate when inputs are sequences.

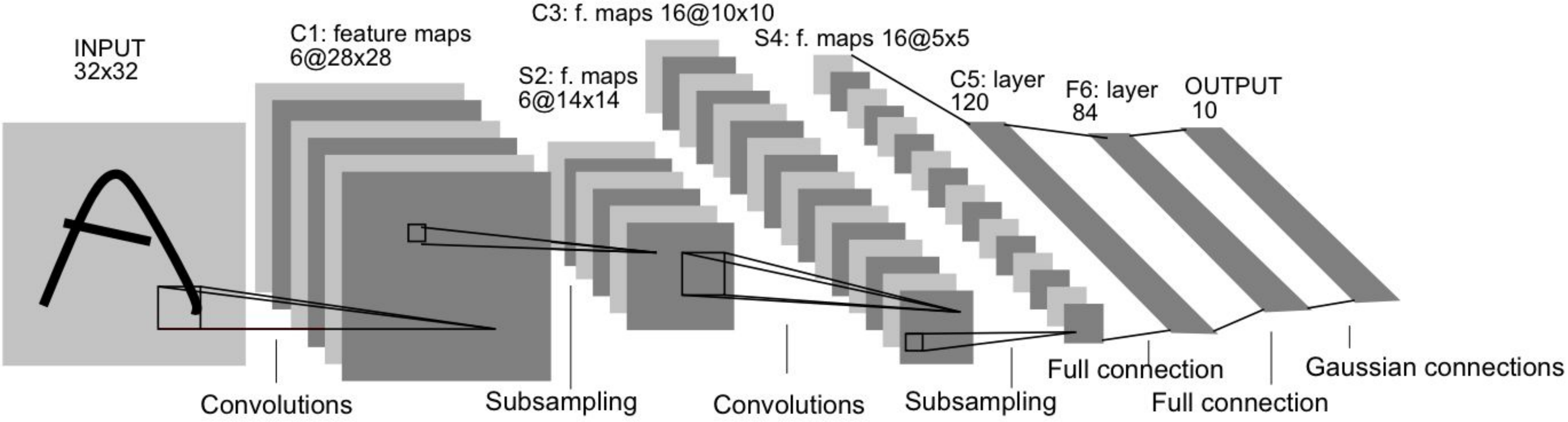
We'll cover in detail when we work on text.



Convolutional Neural Networks (ConvNets)

Appropriate for image tasks, but not limited to image tasks.

We'll cover in detail in the next session.



Hyperparameters

Activations: a zoo of nonlinear functions

Initializations: <u>Distribution</u> of initial weights. Not all zeros.

Optimizers: driving the gradient descent

Objectives: comparing a prediction to the truth

Regularizers: forcing the function we learn to remain “simple”

...and many more

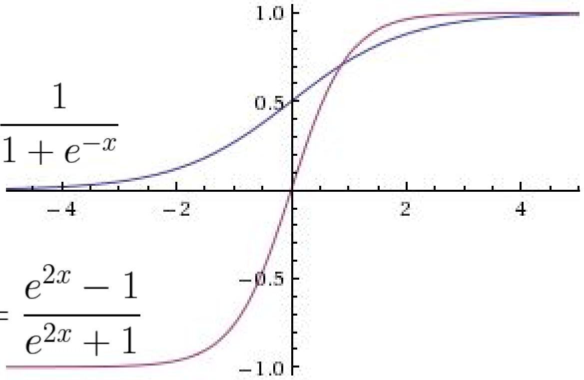
Activations

Nonlinear functions

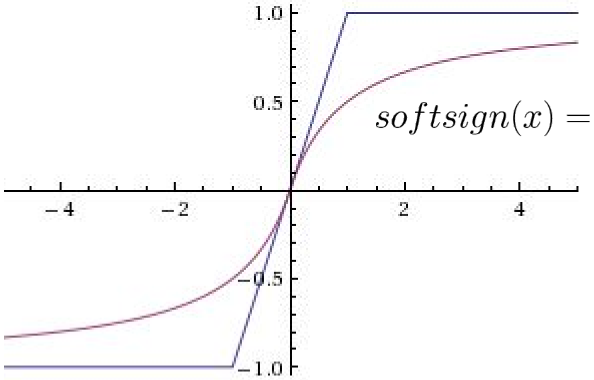
Sigmoid, tanh, hard tanh and softsign

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

$$tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

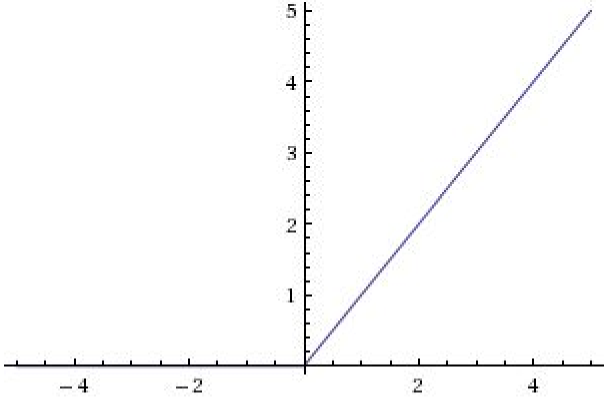


$$softsign(x) = \frac{x}{1+|x|}$$

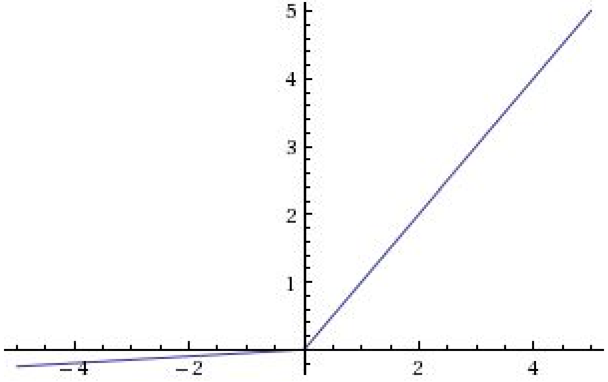


$$ht(x) = max(-1, min(x, 1))$$

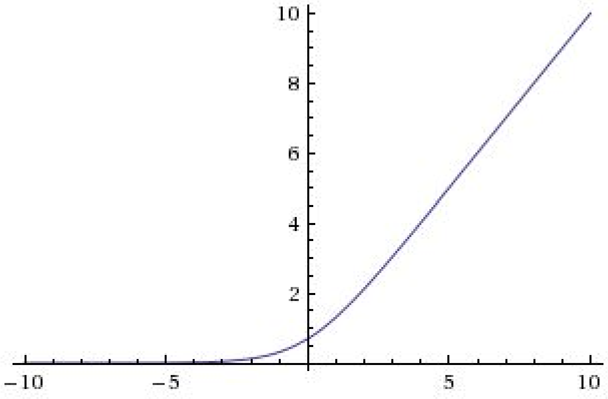
ReLU (Rectified Linear Unit) and Leaky ReLU



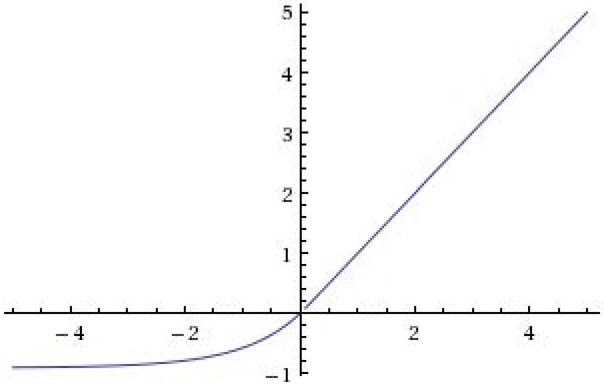
$$ReLU(x) = max(x, 0)$$



Softplus and Exponential Linear Unit (ELU)

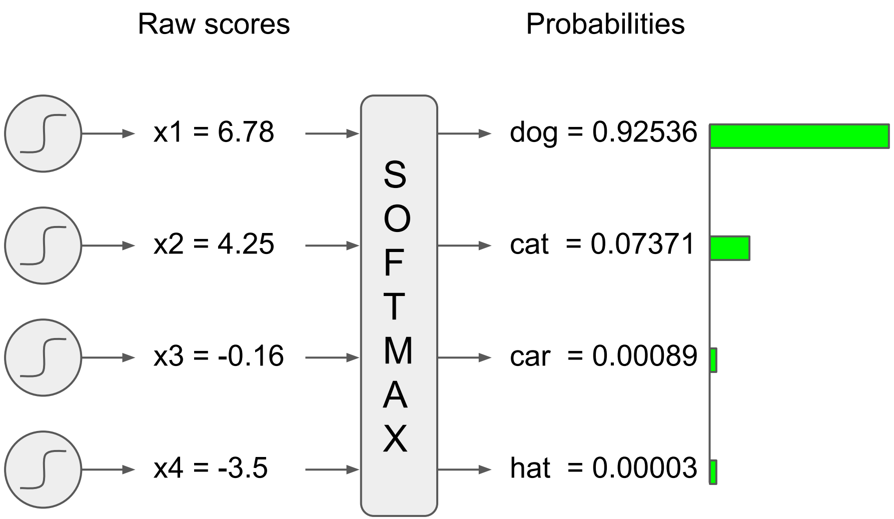


$$softplus(x) = log(1 + e^x)$$



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

Softmax



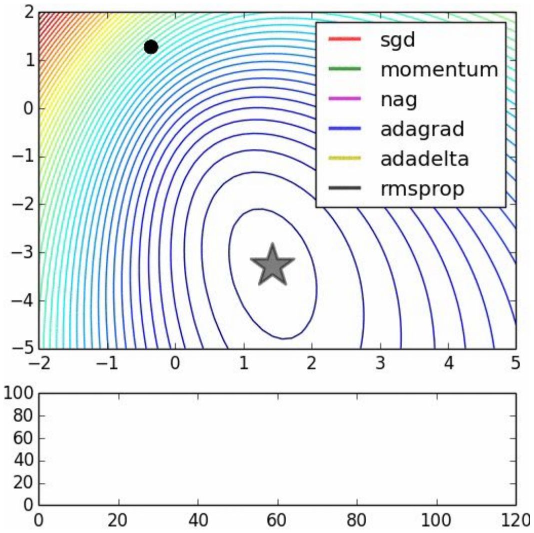
$$p_i = \frac{e^{x_i}}{e^{x_1} + e^{x_2} + e^{x_3} + e^{x_4}}$$

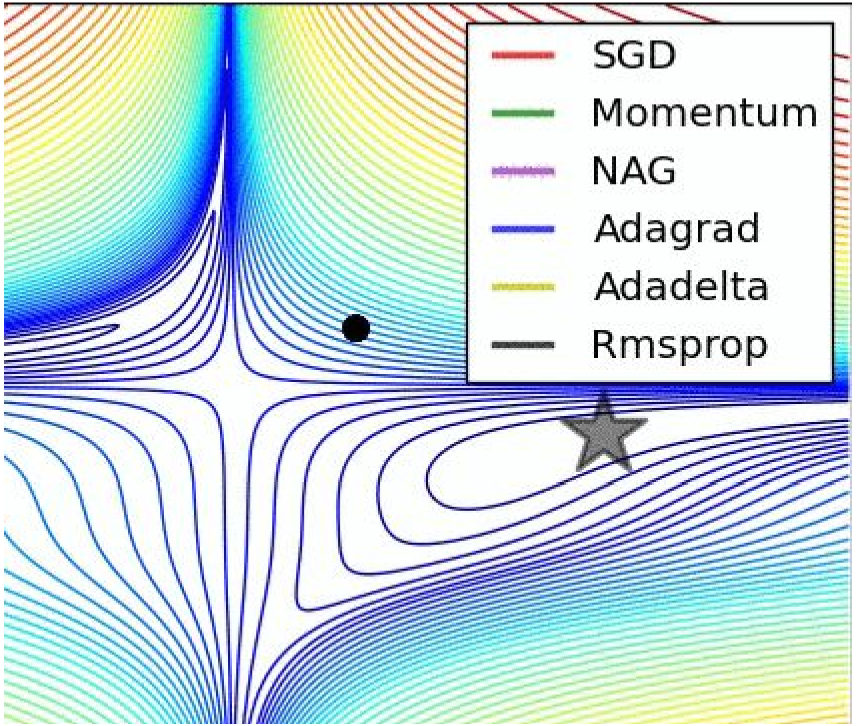
Optimizers

Various algorithms for driving Gradient Descent

Tricks to speed up SGD.

Learn the learning rate.



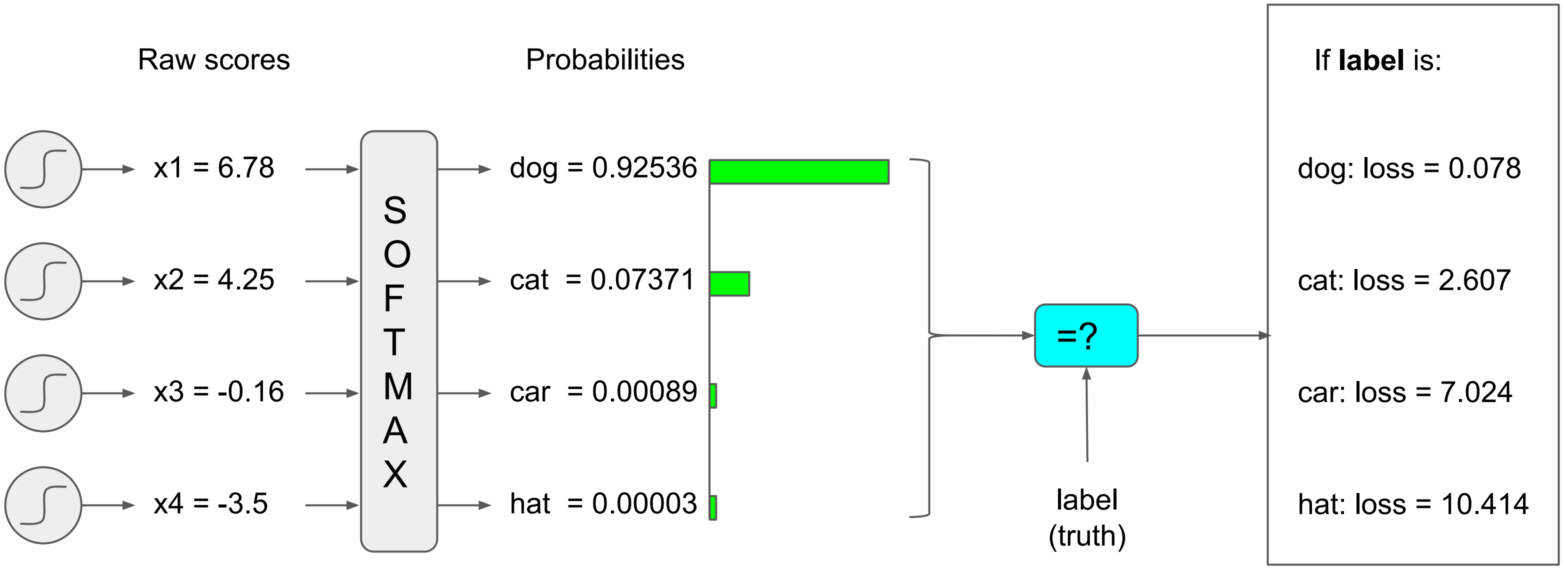


Credit: Alex Radford

Cost/Loss/Objective Functions

=?

Cross-entropy Loss

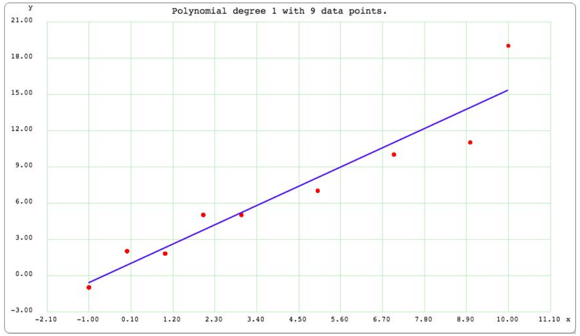


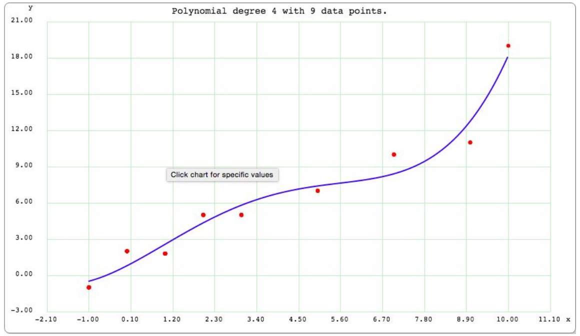
$$p_i = \frac{e^{x_i}}{e^{x_1} + e^{x_2} + e^{x_3} + e^{x_4}}$$

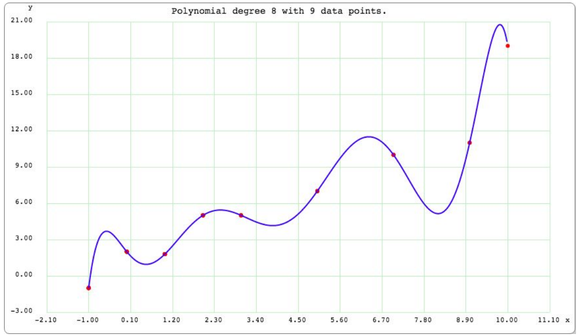
$$loss(label\ is\ i) = -log(p_i)$$

Regularization
Preventing Overfitting

Overfitting







Very simple.
Might not predict well.

Just right?

Overfitting.
Rote memorization.
Will not generalize well.

Regularization: Avoiding Overfitting

Idea: keep the functions "simple" by constraining the weights.

Loss = Error(prediction, truth) + L(keep solution simple)

L2 makes all parameters medium-size

$$L_2 = \frac{\lambda}{2} \sum_i w_i^2$$

L1 kills many parameters.

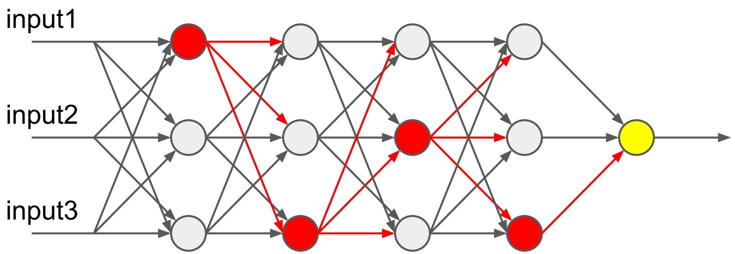
$$L_1 = \lambda \sum_i |w_i|$$

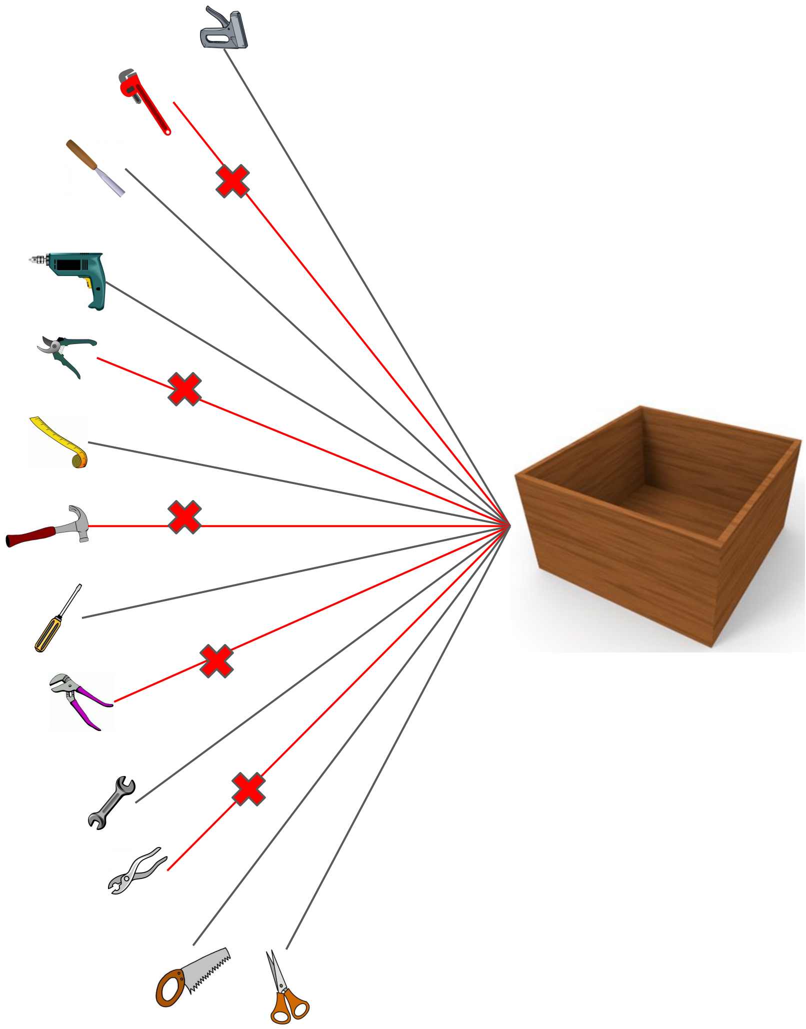
L1+L2 sometimes used.

Regularization: Dropout

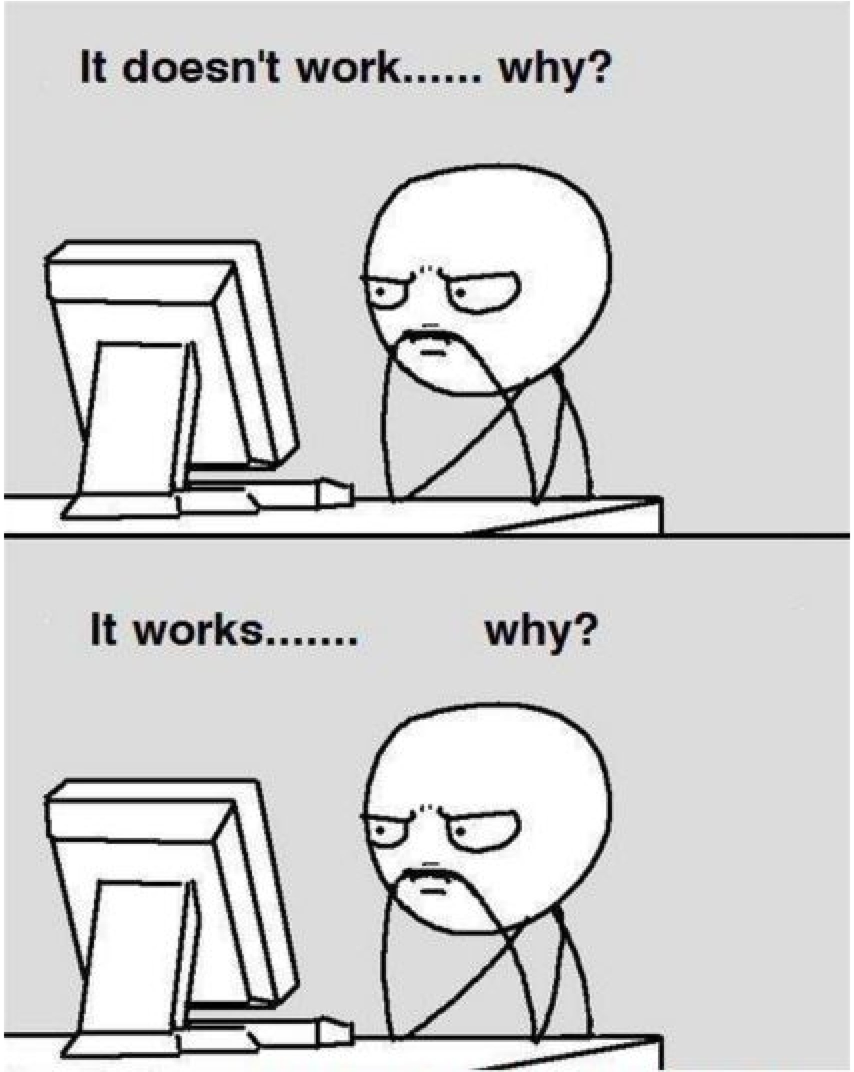
At every step, during training, ignore the output of a fraction p of neurons.

p=0.5 is a good default.





One Last Bit of Wisdom


It doesn't work...... why?
It works.......
why?

THE END