BENJAMIN VILLALONGA CORREA

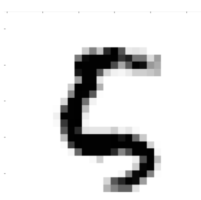
INTRODUCTION TO SUPERVISED MACHINE LEARNING

THE PROBLEM

MNIST is the typical benchmark for each supervised machine learning algorithm:

+ tags (that classify the images)

Train the *machine* with a lot of (image,tag) pairs.



The *machine* thinks

5!

LET'S FORMALIZE THE PROBLEM

Input images are vectors of a 28*28-dimensional space:

 $= (0.0, \ldots, 0.997, 0.861, \ldots, 0.0)^T = x$

And tags are basis vectors of a 10-dimensional space:

$$8 = (0, 0, 0, 0, 0, 0, 0, 0, 1, 0) = y$$

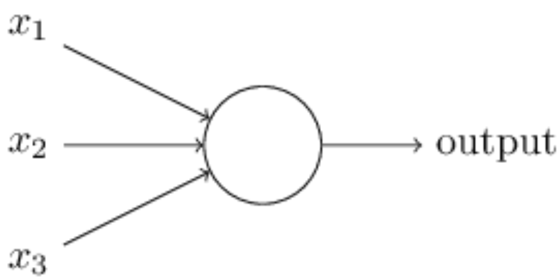
There is a function that from the space of x's to the space of y's that:

$$f(x) = \tilde{y} \approx y$$

This discrepancy will be quantified through a cost function, which we will minimize.

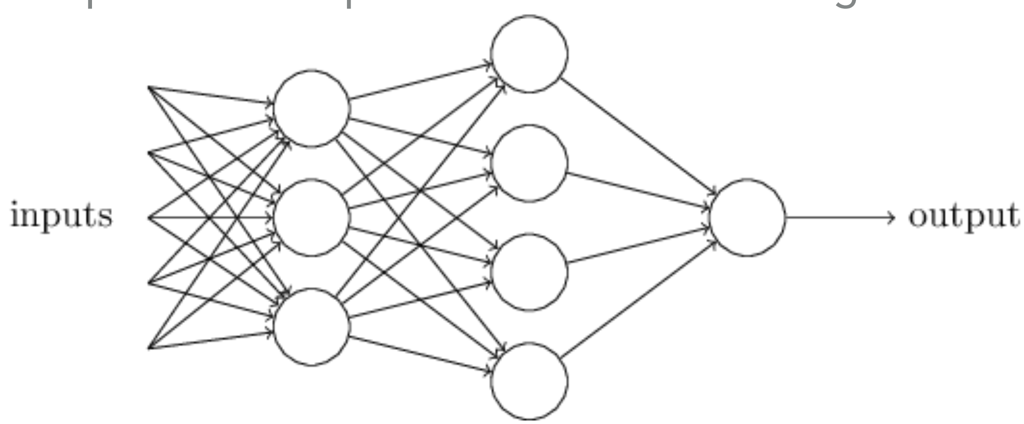
What class of functions will be easy to use and give good results?

NEURAL NETWORKS: PERCEPTRONS



$$\text{output} = \Theta\left(\sum_i w_i \cdot x_i + b\right)$$

By arranging perceptrons in complicated networks we can get nuanced decision making:

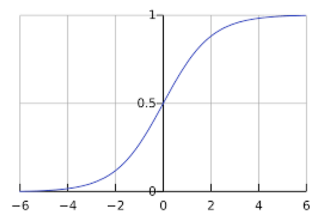


However, the output is discretized, and they are hard to optimize. Let's get a continuous version.

NEURAL NETWORKS: SIGMOID NEURONS

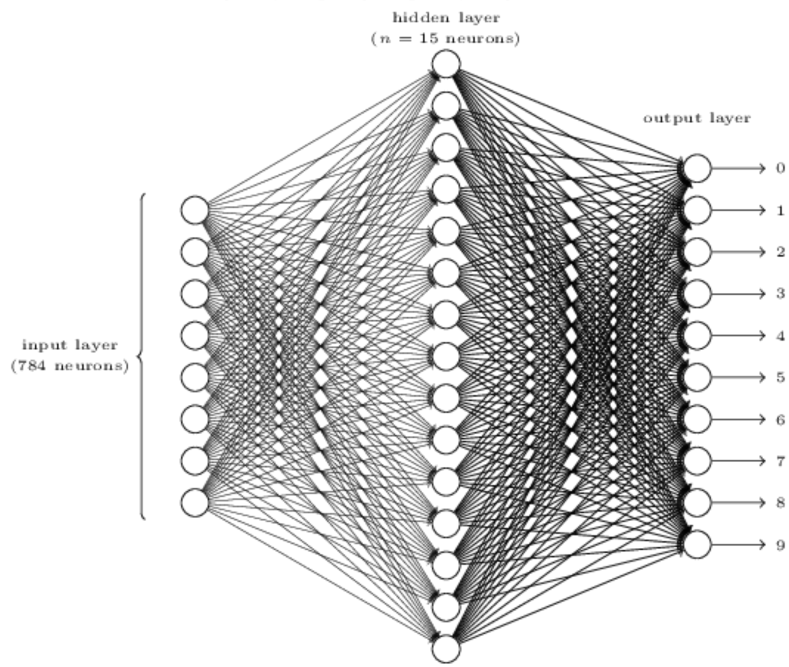
The *sigmoid function* is a continuous "version" of the step function:

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



Each neuron decides "continuously".

$$\text{input } x$$

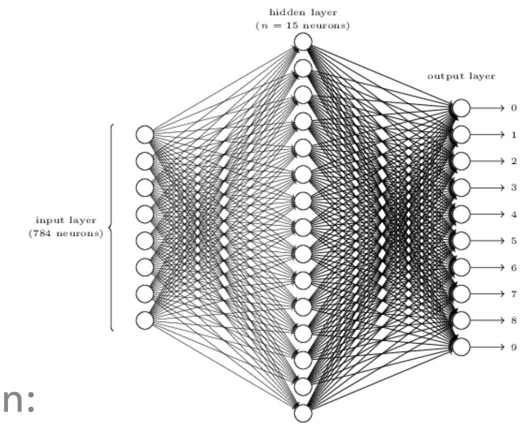


$$\text{output } \tilde{y} \approx y$$

The output is now a probability distribution of how likely it is that each number has been detected.

The exact y is also a probability distribution, but with 100% certainty of what the label is.

THE ACTION OF EACH LAYER OF NEURONS



Each neuron (j) applies a linear function from a vector space to 1-D space:

$$z_j = \sum_i w_{j,i} \cdot x_i + b_j$$

(plus the constant **b**)

then "softens" the result by the (non-linear, but monotonic) sigmoid function:

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

If every neuron in a layer is connected to every neuron in the next one, then:

$$z^{\text{layer}} = w^{\text{layer}} \cdot x^{\text{layer}} + b^{\text{layer}}$$

$$x^{\text{layer}+1} = \sigma(z^{\text{layer}})$$

We call the last output:

$$\tilde{y}$$

COST FUNCTION

There are many possible choices. One is average euclidean norm of the discrepancy over a training set:

$$C(w,b) \equiv \frac{1}{2n} \sum_x |\tilde{y} - y|^2$$

(Another good choice would be the cross-entropy between both distributions…)

Loading an entire training set of x's and y's, we minimize C through (say) an Gradient Descent.

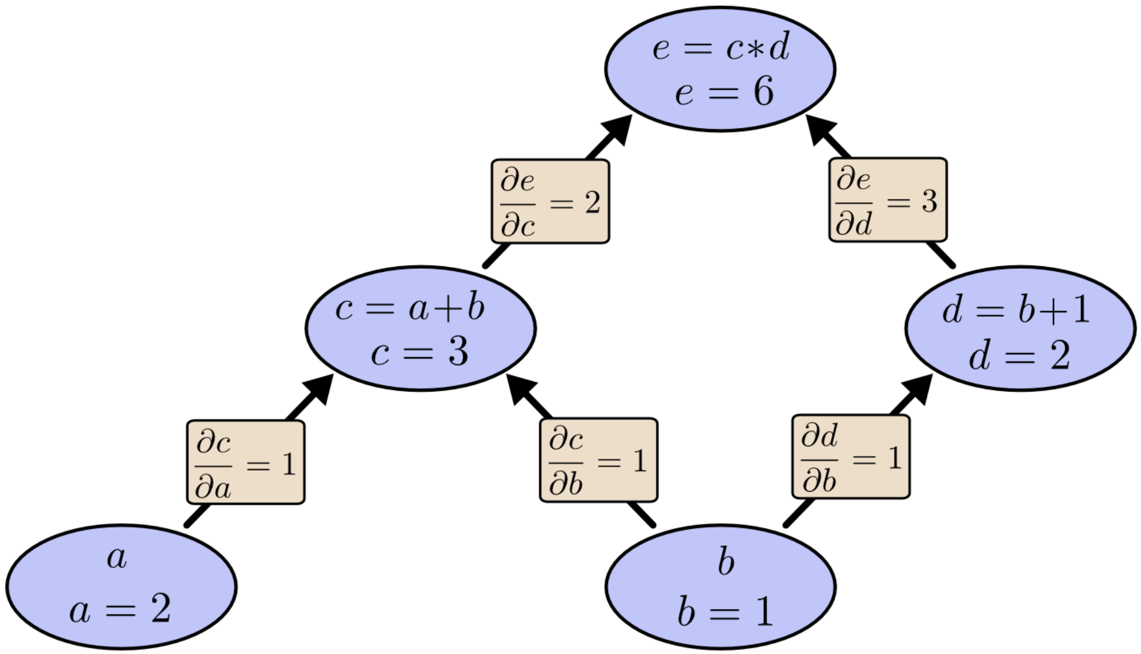
Training sets might get very large, so we usually approximate

$$\frac{\sum_{j=1}^{m} \nabla C_{x_j}}{m} \approx \frac{\sum_{j=1}^{n} \nabla C_{x_j}}{n} = \nabla C$$ Stochastic Gradient Descent

Where the set of m training elements was chosen randomly among the full set of elements.

Gradient descent methods rely on computing partial derivatives. The *backpropagation* method makes it cheap.

BACKPROPAGATION (IN 2 MINS)



It is much cheaper to traverse backwards (only once).

NIELSEN'S PEDAGOGICAL CODE

- 3 layers (input, hidden and output).
- Mini-batches of 10 images.
- 30 training epochs.
- Learning rate (step of the descent) of 3.0.
- Gets about 95% accuracy over test cases.

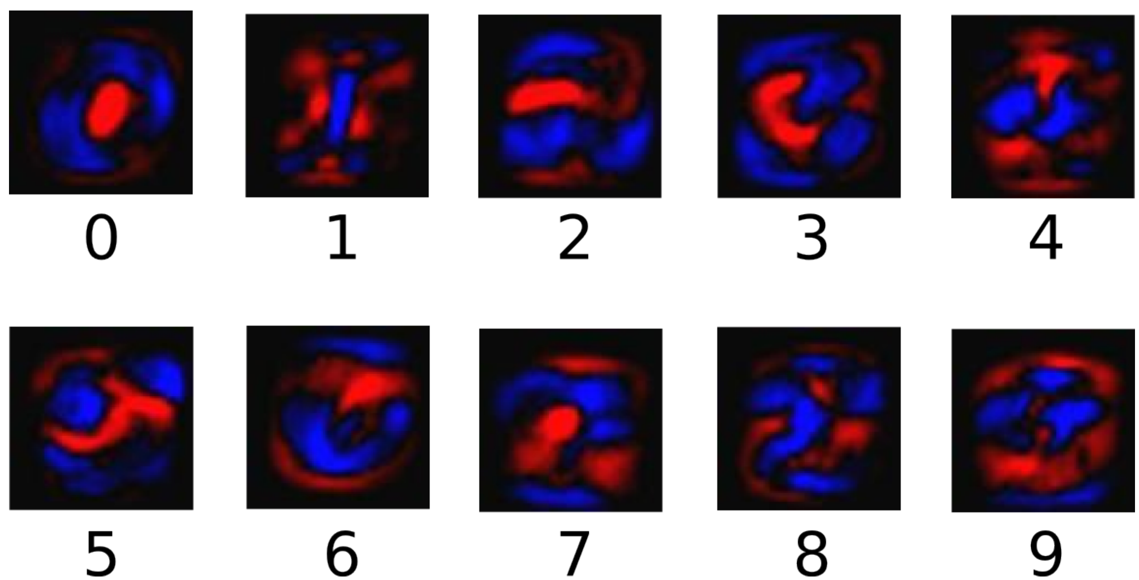
DEMONSTRATION

Demonstrating…

INTUITION OVER A LAYER'S WEIGHTS

For a (no hidden, less intuitive) layer network, TensorFlow gets the following weights (for fixed output i):

$$w_{i,j}$$



REFERENCES

- TensorFlow tutorials (tensorflow.org)
- Michael Nielsen's free online book (neuralnetworksanddeeplearning.com)
- Colah's blog (http://colah.github.io)
- Almost all figures come from Nielsen's book, except for the backpropagation graph (Colah) and the weights of that contribute to evidence (TensorFlow).