UNIVERSITÀ
DI TORINO

Laurea Magistrale in Informatica

Dipartimento di Informatica

Neural Networks

Course

Neural Networks and Deep Learning

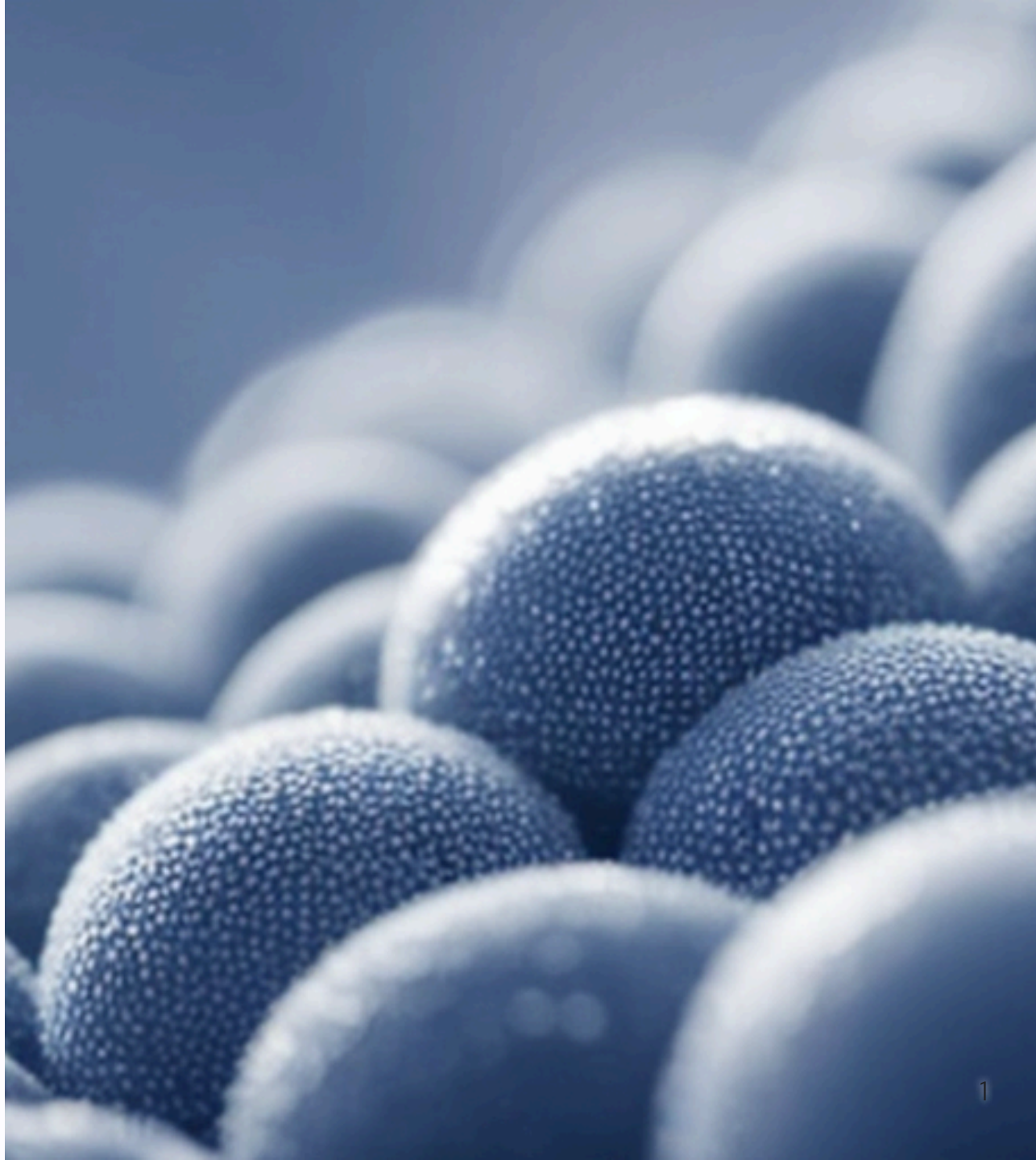
Professor

Roberto Esposito (roberto.esposito@unito.it)

Image dreamed by stable diffusion

Prompt: "nice image for the cover of a course describing the work of basi neural networks | cinematic"

Teacher: Roberto Esposito



Reference Material

These lectures are based on the book: Neural Networks and Deep Learning by Michael Nielsen.

Using Neural Networks to Recognize Handwritten Digits

While most people effortlessly recognize the digits on the right as 504192, that ease is deceptive:

- In each hemisphere of our brain, we have a primary visual cortex (a.k.a., V1) containing 140M neurons, and tens of billions of connections;
- an entire series of visual cortices – V2, V3, V4, and V5 – doing progressively more complex image processing is also present.

In fact, **recognizing handwritten digits isn't easy**.

The difficulty of visual pattern recognition becomes apparent if you attempt to write a computer program to recognize digits like those shown before.

Simple intuitions about how we recognize shapes - "a 9 has a loop at the top, and a vertical stroke in the bottom right" - turn out to be not so simple to express algorithmically.

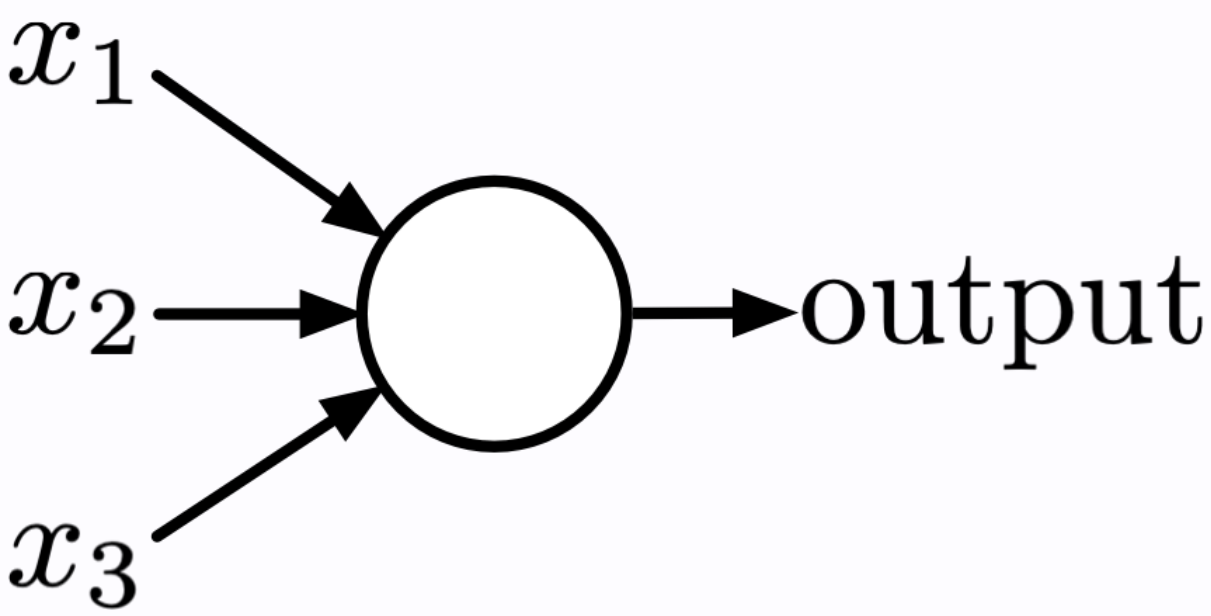
Machine Learning in general, and Neural Networks in particular, approach the problem in a different way. **The idea is to take a large number of handwritten digits and then develop a system which can learn from those training examples**.

We will develop a *simple* neural network that can recognize digits with an accuracy over 96 percent.

Perceptrons

The Rosemblatt's perceptron is the simplest form of neural network.

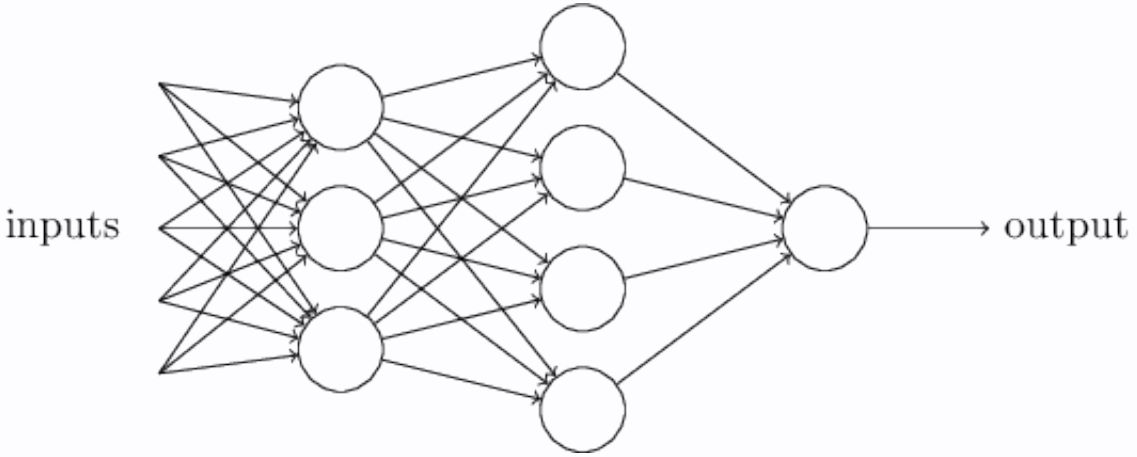
Note: in the following, we will assume that the output of the perceptron is 1 if $\mathbf{w} \cdot \mathbf{x} + b > 0$ and 0 otherwise[1].



1:This definition is slightly imprecise in that the original definition used -1 and 1 as outputs. The change is not essential, and simplifies the discussion.

A Perceptron Based Neural Network

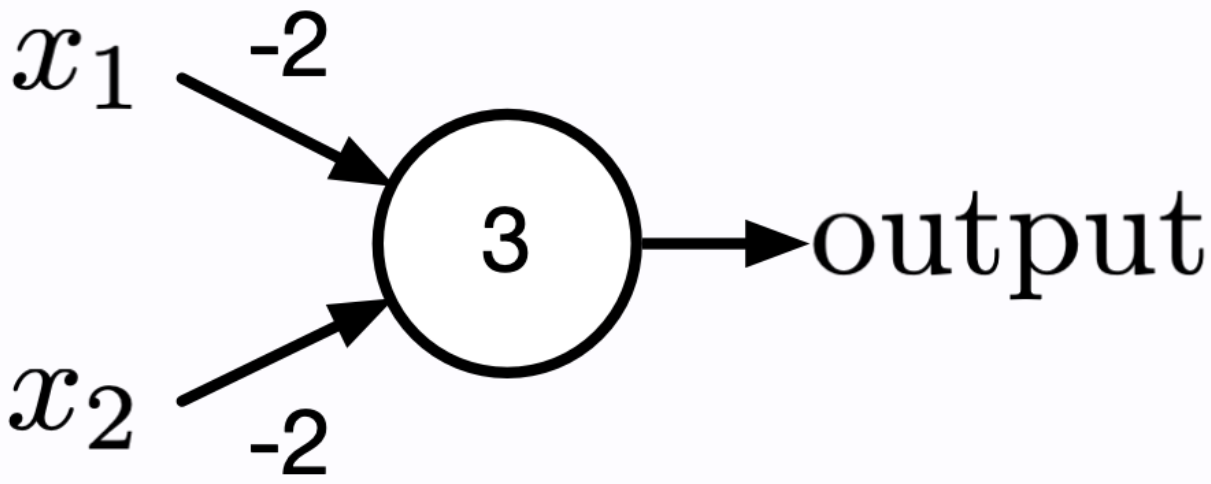
Obviously perceptron can express only very simple (linear) concepts. However it seems plausible that a complex network of perceptrons could make quite subtle decisions:



Networks as computational devices

Another way perceptrons can be used is to compute the elementary logical functions. For example, suppose we have a perceptron with two inputs, each with weight -2, and an overall bias of 3.

Q: Which boolean function does this perceptron compute?

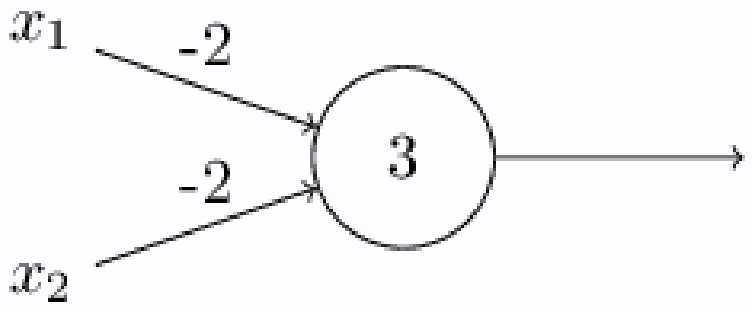


Slido 📱

Question

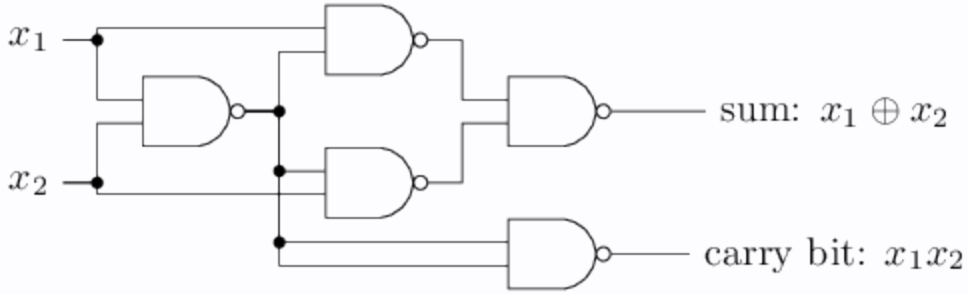
Let us analyze how it behaves:

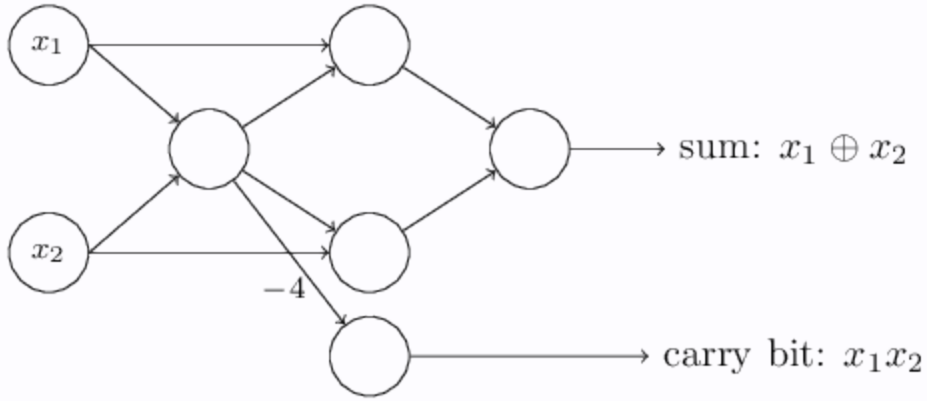
x_1	x_2	$(-2,-2)\cdot x+3$	**out**	$\neg(x_1 \wedge x_2)$
0	0	3	1	1
0	1	1	1	1
1	0	1	1	1
1	1	-1	0	0



As you may know, the NAND example is particularly interesting because it can be easily shown that **any other logic operator can be built in terms of NAND operations only**.

Example: the binary adder (with carry)





The mathematical theory of neural networks has better formalized this idea in the **Universal Approximation Theorem** which states:

Universal approximation theorem

A feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well.

"Deep Learning" by Goodfellow et al, section 6.4.1, citing **Hornik et al. 1989**

Hornik et al. 1989: *Kurt Hornik and Maxwell Stinchcombe and Halbert White* , Elsevier, Neural Networks vol. 2(5), 1989 , **Multilayer feedforward networks are universal approximators.**

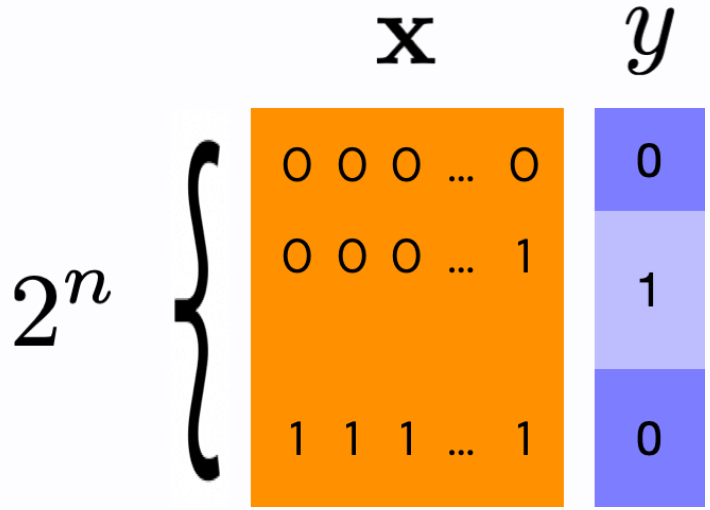
> *The concept of **Borel measurability is beyond the scope of this book**; for our purposes it suffices to say that **any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable** and therefore may be approximated by a neural network.*

"Deep Learning" by Goodfellow et al, section 6.4.1

Universal Approximation Theorem: limitations

Unfortunately, in the worst case, an exponential number of hidden units (possibly with one hidden unit corresponding to each input configuration that needs to be distinguished) may be required.

This is easiest to see in the binary case: the number of possible binary functions on vectors $\mathbf{v} \in \{0,1\}^n$ is 2^{2^n} and selecting one such function requires 2^n bits, which will in general require $O(2^n)$ degrees of freedom.



Networks as computational devices

Why is this important?

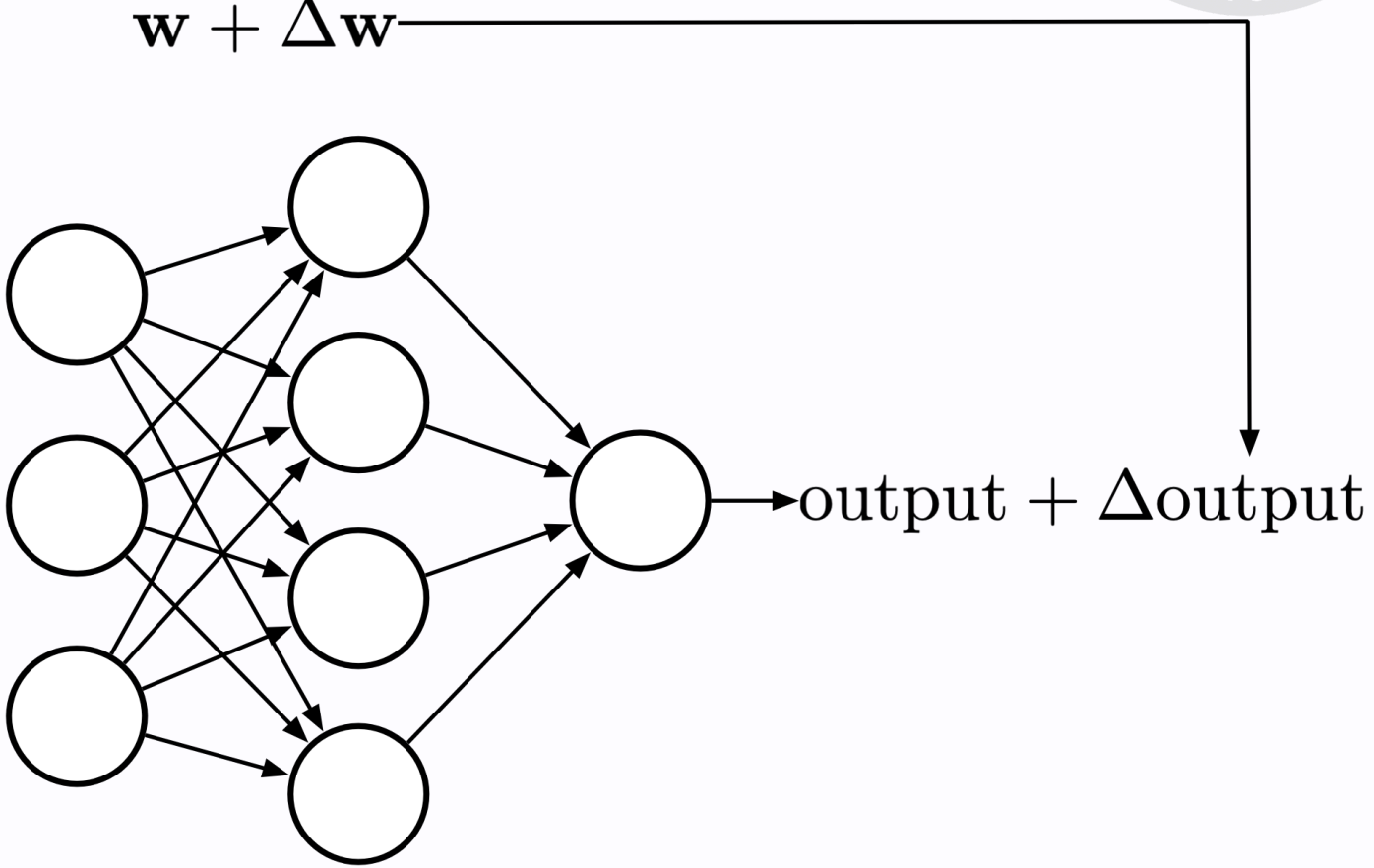
1. Perceptron networks can perform all sorts of computations;
2. **!! We can make algorithms that learns those computations !!**

Sigmoid neurons

But... it is hard to learn anything useful using the perceptrons.

Desiderata

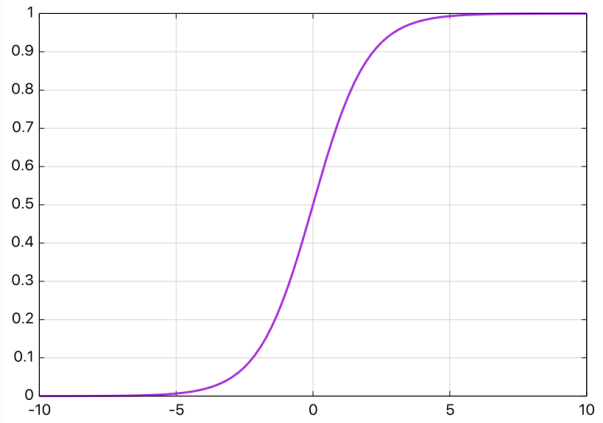
Small changes in any weight (or bias) causes a small changes in the output



Sigmoid neurons

A sigmoid neuron is defined as a neuron where the output is computed using the formula: $\sigma(w \cdot x + b)$, where

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



Note that:

- their output approximate well the step function when $w \cdot x + b$ is very large or very small, but it does not change abruptly in proximity of 0;
- Since the output is in $[0, 1]$, each one of the input of internal nodes will take values ranging in $[0, 1]$ as well.

Sigmoid neurons

To recap: it is important that the **activation function** is smooth since we would like to control how varies Δoutput.

Calculus implies that:

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b$$

We need to **require that the** output **is differentiable w.r.t. the weights and the biases**.

Note: ***it is not the exact shape of*** σ ***to be crucial, but rather its smoothness.***

But then a question arises: why this particular form for σ*? Depending on the circumstances other forms of sigma may be better.*

Exercises

Sigmoid neurons simulating perceptrons, part I

Suppose we take all the weights and biases in a network of perceptrons, and multiply them by a positive constant, $c > 0$. Show that the behavior of the network doesn't change.

Sigmoid neurons simulating perceptrons, part II

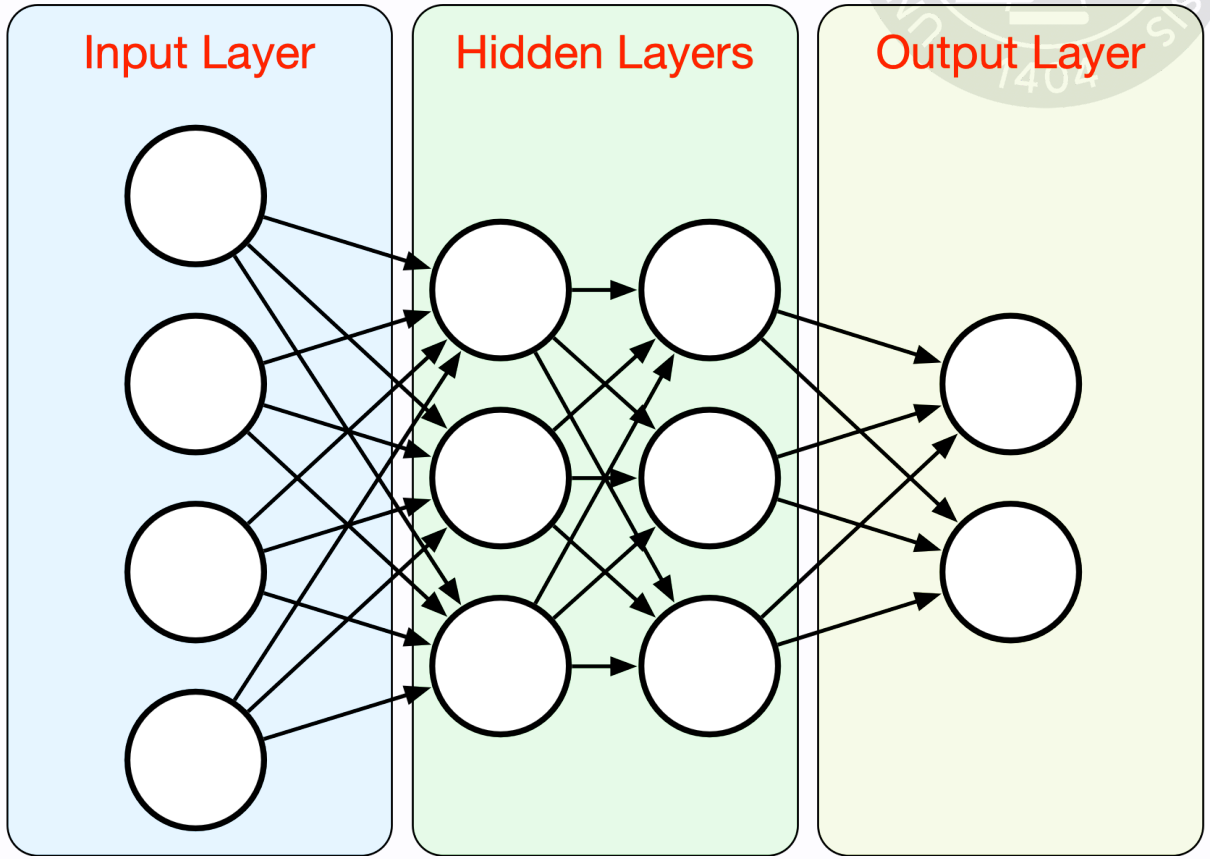
Suppose we have the same setup as the last problem - a network of perceptrons. Suppose also that the overall input to the network of perceptrons has been chosen. We won't need the actual input value, we just need the input to be fixed. Suppose the weights and biases are such that $\mathbf{w} \cdot \mathbf{x} + b \neq 0$ to any particular perceptron in the network. Now **replace all the perceptrons in the network by sigmoid neurons**, and multiply the weights and biases by a positive constant $c > 0$.

Show that in the limit as $c \to \infty$, the behavior of this network of sigmoid neurons is exactly the same as the network of perceptrons. How can this fail when $\mathbf{w} \cdot \mathbf{x} + b = 0$ for one of the perceptrons?

Architecture of Neural Networks

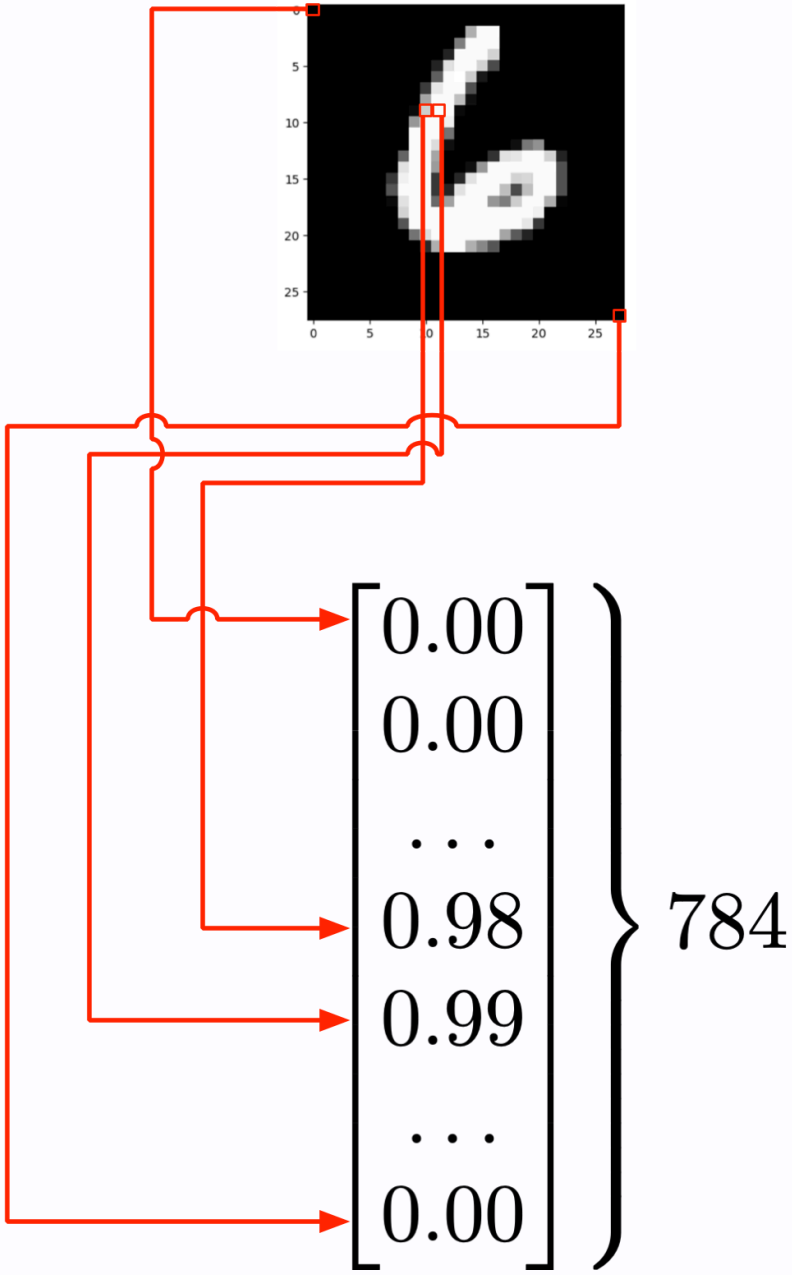
Notes:

- the input layer is not a proper "layer", it only represents the inputs;
- This kind of architecture, for historical reasons is called "Multilayer perceptrons" despite not being built out of perceptrons.



Input and Output layer design

Design of input and output layers are often straightforward. In our particular example of handwritten digits classification.

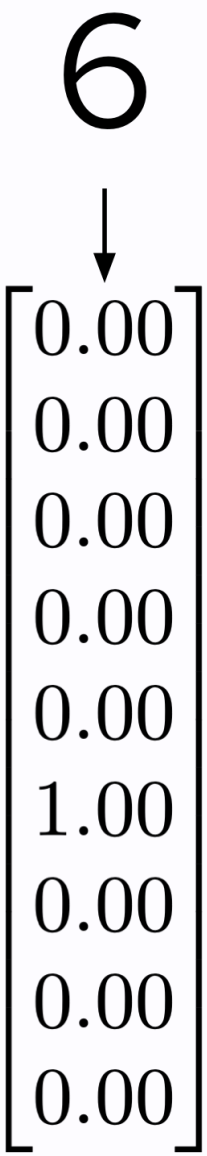


The Input layer

We will represent each training input $\mathbf{x} \in \mathbf{X}$ with a $28 \times 28 = 784$ dimensional vector. Each element of the vector is modeled as an **input neuron**.

Slido 📱

Question



The Output layer

We will represent the labels by $\mathbf{y} = \mathbf{y}(\mathbf{x})$, where $\mathbf{y}$ is a 10-dimensional vector.

For instance $\mathbf{y}(\mathbf{x}) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$ is the desired output from the network when $\mathbf{x}$ is an image depicting the digit 6.

Each element of the output vector will be modeled as an **output neuron**.

Hidden layers design

It's not possible to sum up the design process for the hidden layers with a few simple rules of thumb. Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behavior they want out of their nets.

Examples:

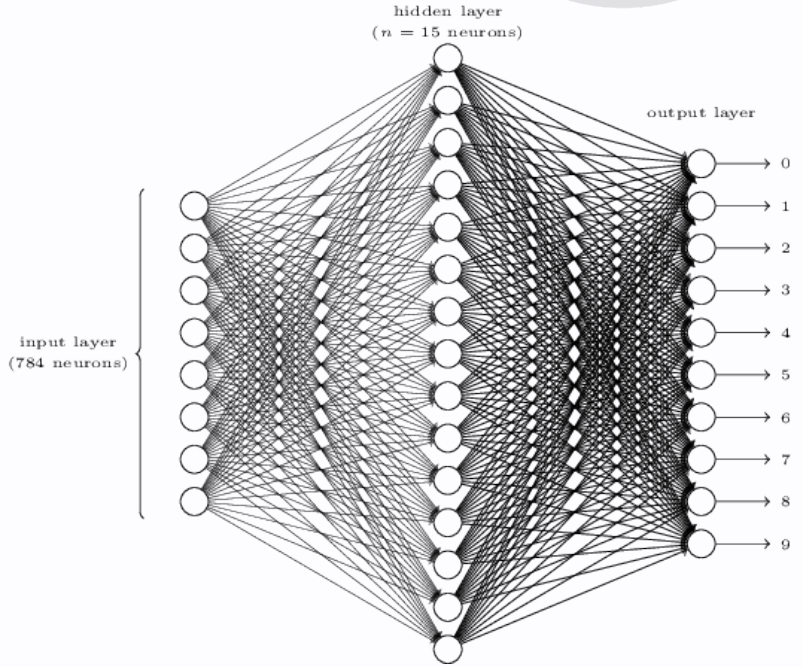
- sharing weights between neurons allows to learn a single feature detector and apply it in many different locations;
- stacking layers may allow to learn complex features from simpler ones;
- a large number of hidden units may allow to learn complex functions at the cost of overfitting or requiring more data.
- ...

A simple NN to recognize handwritten digits

To represent the labels using binary outputs it would have sufficed to use 4 units. Why do we use 10 digits instead?

Hints:

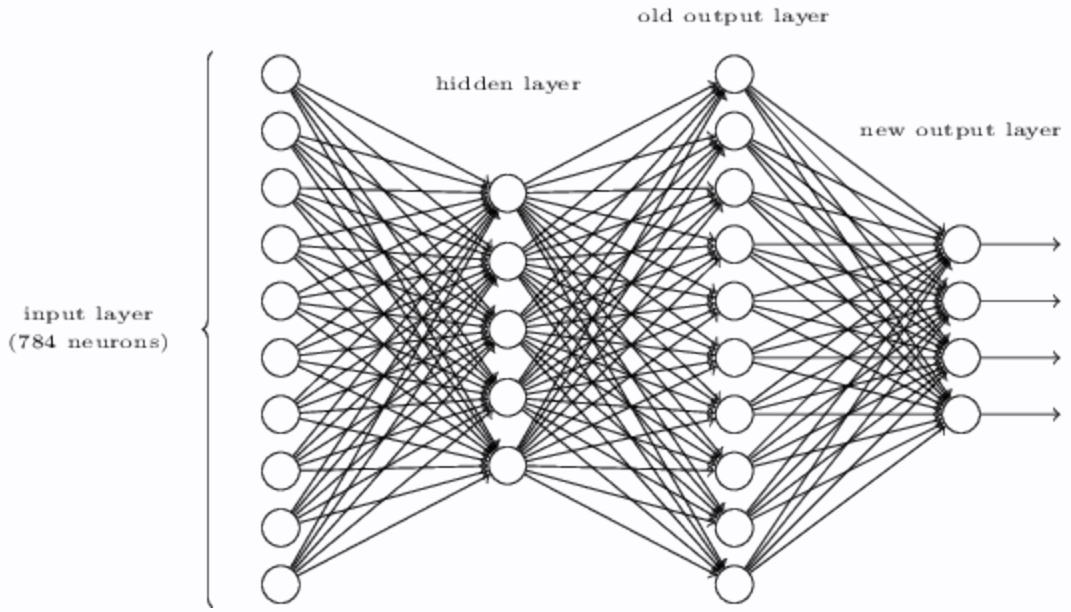
- it is a heuristic (in some other settings could have worked well);
- Distinguishing single digits based on image features is likely to be simpler than distinguish groups of them. Doesn't it?



Exercise

Find a set of weights and biases for the new output layer so that it represents the binary representation of the predicted decimal digit.

Assume that the first 3 layers of neurons are such that the correct output in the third layer (i.e., the old output layer) has activation at least 0.99, and incorrect outputs have activation less than 0.01.



Learning with Gradient Descent

We now want to devise a learning algorithm to find weights for our network.

To do that we will need to define a cost function that measures how well our network is doing, and then find a way to minimize this cost function, i.e, to solve the optimization problem:

$$\text{minimize}_{\mathbf{w},\mathbf{b}} C(\mathbf{w}, \mathbf{b})$$

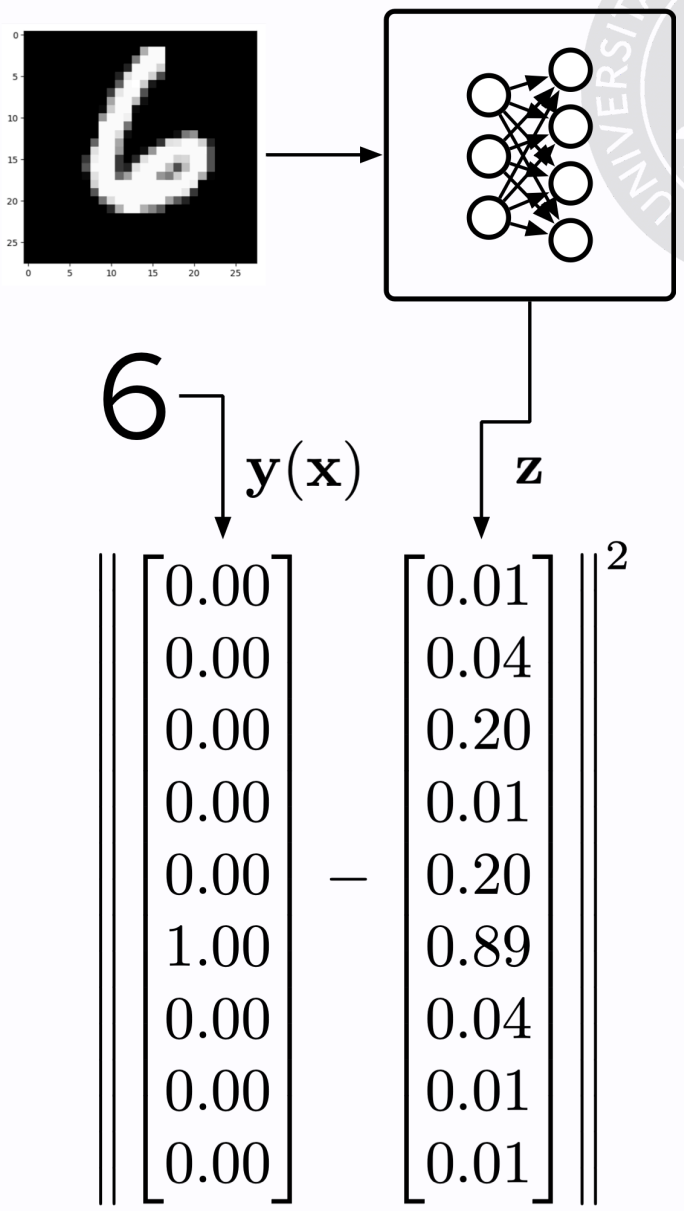
where C is a function of the weights and biases of the network, and it measures how well the network performs on the training data $(\mathbf{X}, \mathbf{y})$.

To quantify how well the current set of parameters are performing, we define a cost function:

$$C(\mathbf{w}, \mathbf{b}) \equiv \frac{1}{2n} \sum_{\mathbf{x}} \|\mathbf{y}(\mathbf{x}) - \mathbf{z}\|^2$$

where $\mathbf{w}$ denotes the collection of all weights in the network, $\mathbf{b}$ all the biases, n is the total number of training inputs, $\mathbf{z}$ is the vector of outputs from the network when $\mathbf{x}$ is input, and the sum is over all training inputs $\mathbf{x}$.

Of course $\mathbf{z}$ depends on $\mathbf{w}$, $\mathbf{b}$ and $\mathbf{x}$ and so it would have better been written as $\mathbf{z}_{\mathbf{w},\mathbf{b}}(\mathbf{x})$. It is here denoted simply as '$\mathbf{z}$' just for notational convenience.



Accuracy is perhaps the most natural evaluation measure for the quality of a classification task:

$$\mathrm{A}_{\mathbf{w},\mathbf{b}} = \frac{1}{n} \sum_{\mathbf{x}} \mathbf{I}_{\mathbf{y}(\mathbf{x})=\mathrm{step}(\mathbf{z})}$$

The indicator function $\mathbf{I}_p$ returns 1 if p is True and 0 otherwise, and $\mathbf{step}(\mathbf{z})$ is the step function applied point-wise to each element of vector $\mathbf{z}$.

Slido Question 📱

Why didn't we use the accuracy measure as our cost function?

Why didn't we use the accuracy measure as our cost function?

The reason is that accuracy is not a smooth function of weights and biases and that makes it difficult to figure out how to change the weights and biases to get improved performance.

Even among smooth functions of weights and biases, why the quadratic cost? Can't we do better?

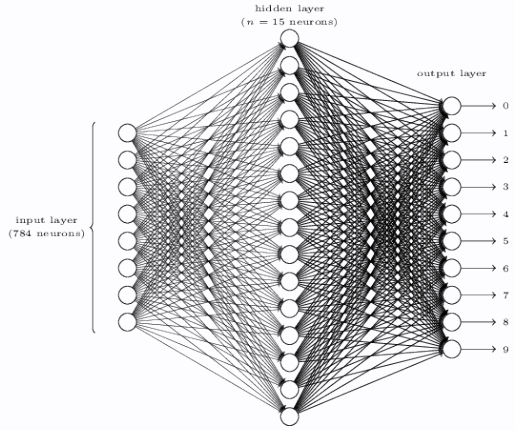
This is a valid concern, **we will see that other cost functions works better in some scenarios**. From the time being, we will use the quadratic cost because it is simpler and it works well for explaining the basics of learning in neural networks.

To **recap**, we want to learn how to predict the digit in an image by minimizing the cost function:

$$\mathbf{minimize}_{\mathbf{w},\mathbf{b}} \left(\frac{1}{2n} \sum_{\mathbf{x}} \|\mathbf{y}(\mathbf{x}) - \mathbf{z}\|^2 \right)$$

where the output $\mathbf{z}$ depends on:

- the network connectivity;
- the weights;
- the biases;
- the particular choices of activation units (sigmoids in our example);
- ...



This is too much complexity! **Our plan**:

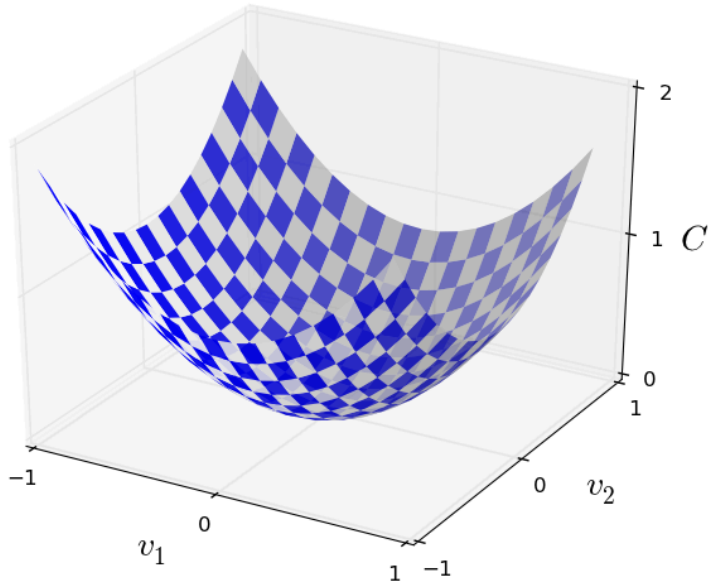
- develop a general iterative method for minimizing multivariate functions
- develop a method to apply it to the specific case of neural networks.

Consider a cost function $C(\mathbf{v})$,

$$C : \mathbb{R}^n \to \mathbb{R}.$$

To help developing our intuition let us assume for the time being that $n = 2$.

Note: in general it might be very difficult to pinpoint the global minimum.



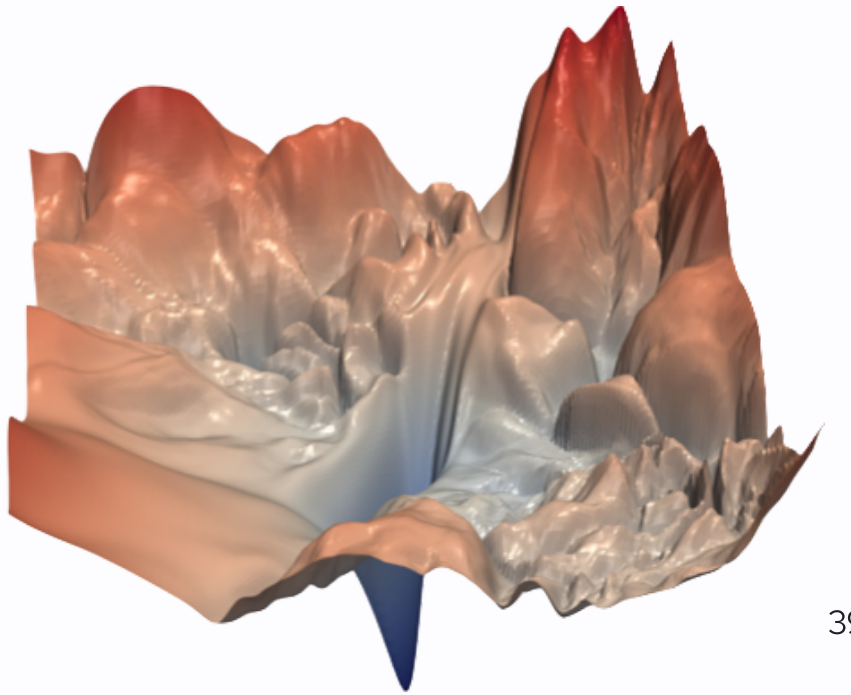


Image from Li et al. "Visualising the Loss Landscape of Neural Nets".

In real problems the number of variables range in the millions, also dependencies between variables complicate things even further.

Using calculus to find closed form solution simply won't work!

Instead imagine the problem of minimizing that function by fixing a value on the v plane and trying to guess how to update that position to lower the value of your cost function as much as possible.

To make things more precise look at what happen when we move our variables by a little amount, let's say by $\Delta\mathbf{v} = (\Delta v_1, \Delta v_2)^T$.

Calculus tell us that the vector C changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 = \nabla C \cdot \Delta\mathbf{v}$$

Slido Question

Can you find a value for $\Delta \mathbf{v}$ which ensures that ∇C decreases?

Hint: it is a function of ∇C.

Note: Write your answer using D(C) to denote ∇C.

By choosing $\Delta \mathbf{v} = -\eta \nabla C$, we are guaranteed to make the cost function to decrease (as long as the step taken is small enough to make the approximation valid).

Consider again the following expression for ΔC:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 = \nabla C \cdot \Delta \mathbf{v}.$$

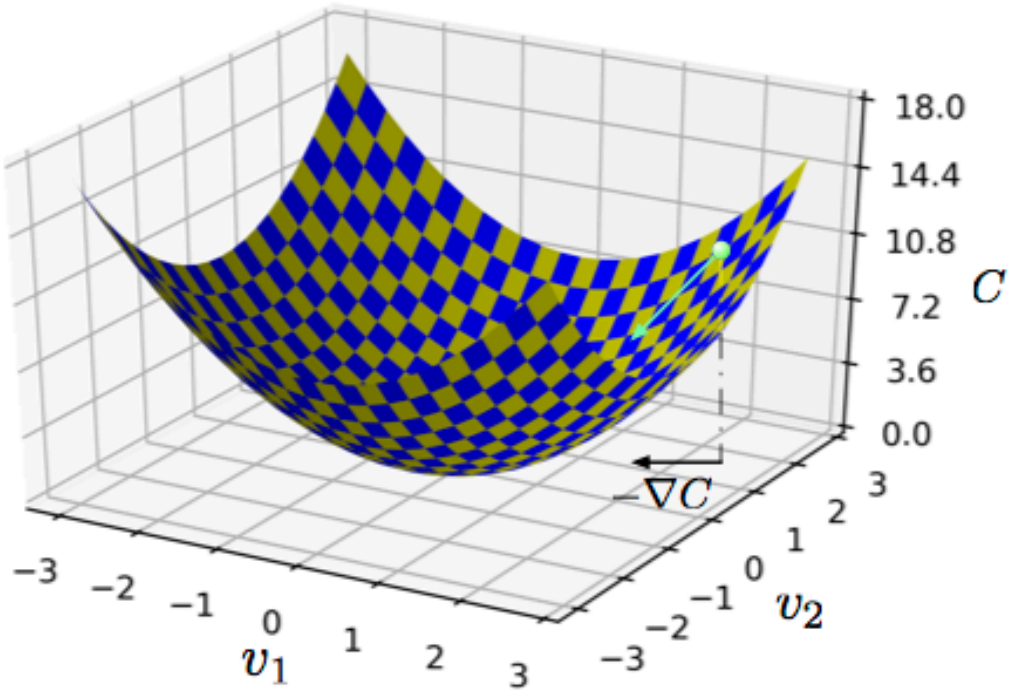
By setting $\Delta \mathbf{v} = -\eta \nabla C$, we have:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2 \leq 0.$$

Gradient descent is an iterative optimization algorithm prescribing that at each step we move our current position in the opposite direction with respect to the gradient:

$$\mathbf{v}' \leftarrow \mathbf{v} - \eta \nabla C.$$

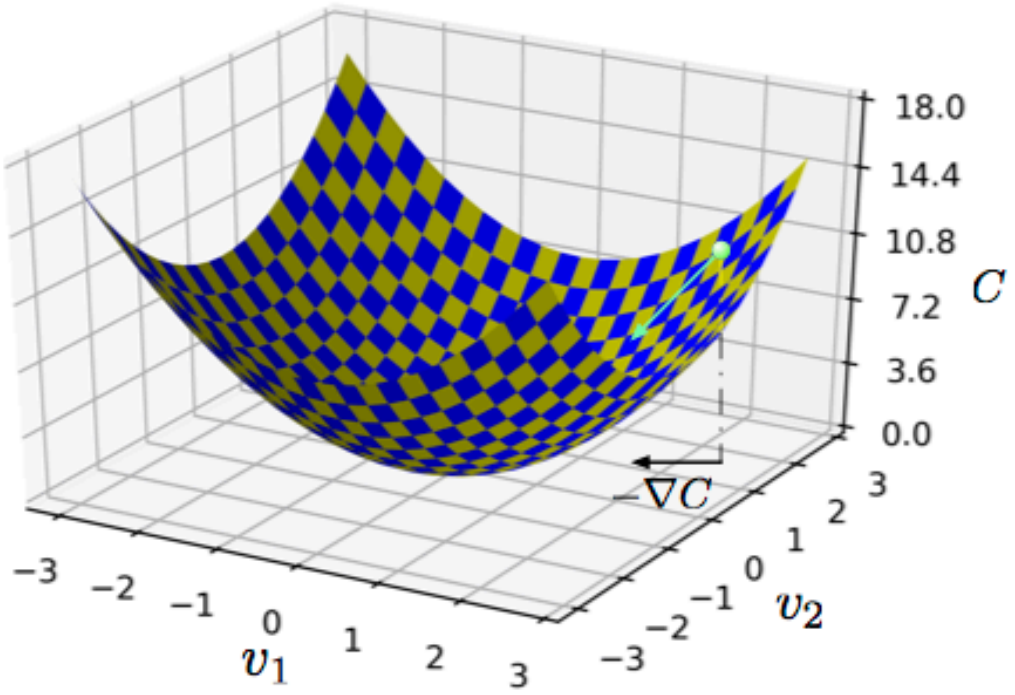
We are **descending the error function C by following the direction of the steepest descent**, hence the name.



Learning with Gradient Descent

Note that the choice of η is important:

- if it is too large then the approximation would not be correct and we may have $\Delta C > 0$;
- if it is too small the algorithm would take very little steps and become too slow.



Improving Gradient Descent

Why not repeating the whole reasoning using a **second order approximation** of ΔC?

Such methods do exists, but they become very unwieldy as soon as the number of parameters grows too large. The reason being that to compute (e.g.) the second order approximation of ΔC we need to compute all second partial derivatives of C.

Since the number of second partial derivatives grows quadratically with the number of variables, these methods become unwieldy very fast.

While techniques exist to mitigate these problems, gradient descent (and variants) are still the most used method to train neural networks.

Applying Gradient Descent in a Neural Network

By replacing $\mathbf{v}$ with our weights $\mathbf{w}$ and biases $\mathbf{b}$, we end up with a recipe for training the neural network: start from a fixed point and iterate through the data updating at each step the weights using the formulas:

$$w_k' \leftarrow w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l' \leftarrow b_l - \eta \frac{\partial C}{\partial b_l}$$

Applying Gradient Descent in a Neural Network

It is hopefully clear that many details are lacking in our current formulation of the gradient descent algorithm.

Also, **several problems needs to be addressed** before we can apply the algorithm in a real context.

Solving one of these problems is the focus of the next topic.

Stochastic Gradient Descent

Consider the cost function $C = \frac{1}{n} \sum_{\mathbf{x}} C_{\mathbf{x}}$.

The gradient of C is then $\nabla C = \frac{1}{n} \sum_{\mathbf{x}} \nabla C_{\mathbf{x}}$.

I.e., to compute the gradient ∇C we need to compute the gradients $\nabla C_{\mathbf{x}}$ separately for each training input $\mathbf{x}$ and then average them, .

Only then we are allowed to take a step in the direction suggested by the gradient.

When the dataset is very big, this becomes too slow.

Stochastic Gradient Descent is the idea of approximating the gradient over the entire dataset using a small subsample of it. Let $\mathbf{x}_1, \ldots, \mathbf{x}_m$ be m randomly chosen training inputs. If m is large enough, it should be apparent that:

$$\sum_{j=1}^{m} \frac{\nabla C_{\mathbf{x}_j}}{m} \approx \frac{\sum_{\mathbf{x}} \nabla C_{\mathbf{x}}}{n} = \nabla C$$

In stochastic gradient descent, a **mini-batch** (i.e., a subsample of our training data) is extracted and the weights are updated using the rule.

$$w_k' \leftarrow w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{\mathrm{x}_j}}{\partial w_k}$$

$$b_l' \leftarrow b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{\mathrm{x}_j}}{\partial b_l}$$

Then we pick out another randomly chosen mini-batch and train with those. When all examples have been used we say that we completed an **epoch** of training. At that point we start over with a new training epoch.

Example

The whole point of stochastic gradient descent is that it's much easier to sample a small mini-batch than it is to apply gradient descent to the full batch. For example, if we have a training set of size $n = 60,000$ as in MNIST, and choose a mini-batch size of (say) $m = 10$, this means we'll get a factor of $6,000$ speedup in estimating the gradient!

Of course, the estimate won't be perfect - there will be statistical fluctuations - but it doesn't need to be perfect: all we really care about is moving in a general direction that will help decrease C.

Recognizing Hand Written Digits

By applying stochastic gradient descent to train a 3-layers network over the MNIST dataset it is not difficult to reach accuracies that are just below 97%!

Details:

- the network is a 3 layer, fully connected, feedforward network with 784 input neurons, 100 hidden neurons, and 10 output neurons;
- the training set contains 50.000 examples (sampled from the 60.000 contained in the original MNIST training set);
- the network has been trained for ~30 epochs;
- η has been set to 3.0;
- The minibatch size has been set to 10.