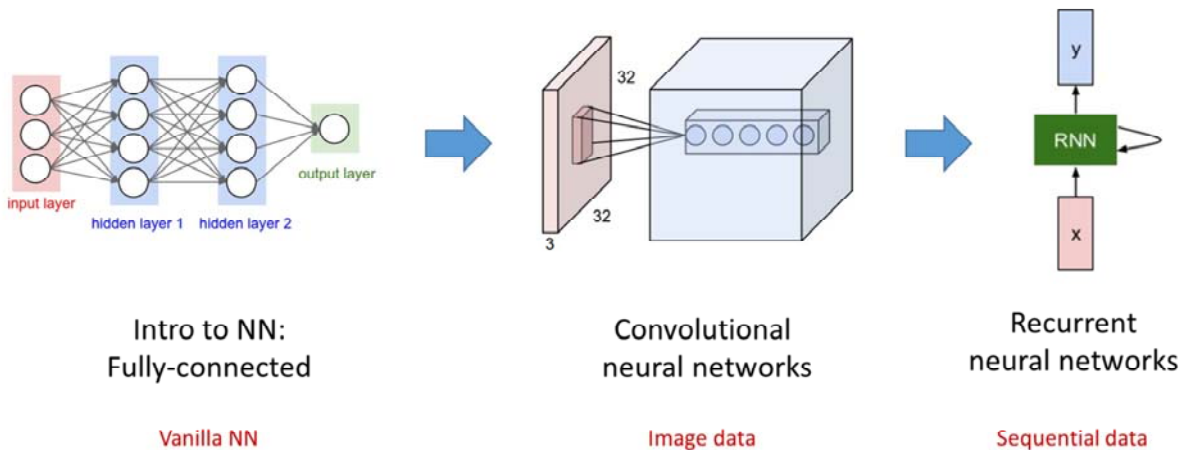


# Deep Learning crash-course

## Session 1: Intro to neural networks

Sofia Minano, BioDTP,  
Wednesday 6th November 2019

## This course



The main objective of the course is to get an overview of the current landscape in neural networks and Deep learning, and give you a starting point if you want to develop further on these models, for fun or to apply them in your research. We focused on giving a general and not too handwavy intuition of the main concepts.

We structured it in three sessions, with theory and practice parts, that focus on the three types of neural networks that are more consolidated now. In this session we will look at 'vanilla neural networks', the most simple approach but useful to get to grips with the main concepts. In the following sessions we will look at convolutional and recurrent nets, which are usually applied in image and sequential data respectively

We will also take the chance in the following sessions to delve a bit deeper on some of the concepts from the intro session

## Outline of this session



- |   |        |
|---|--------|
| <b>1. Basics</b>  | 5 min  |
| <b>2. Structure of a NN</b> <ul style="list-style-type: none"><li>• Single neuron</li><li>• Multi-layer Perceptron (MLP)</li></ul>                                      | 15 min |
| <b>3. Forward pass</b> <ul style="list-style-type: none"><li>• Transformations across layers (weights &amp; biases)</li><li>• Matrix notation</li></ul>                 | 15 min |
| <b>4. Training and backward pass</b> <ul style="list-style-type: none"><li>• Intuition</li><li>• Loss function</li><li>• Gradient descent and backpropagation</li></ul> | 20 min |
- **Practical: MLP for digit recognition with Keras**

# Basics

This section is based on chapter 1 of Deep Learning, Goodfellow et al

This part of the lecture is mostly based on Chapter 1 of the Deep Learning book, with some additions from other resources (noted in the slides)

## Basics

**Deep Learning** is

- an approach to **Artificial Intelligence**
- a type of **Machine Learning** based on artificial neural networks

**Deep Learning** achieves great power and flexibility by:

1. Gathering knowledge from **experience**
2. Being **compositional**



We are going to go through each of the statements that relate these concepts that often go together

Deep Learning is an approach to Artificial Intelligence

Deep Learning is a type of Machine Learning based on artificial neural networks

ML enables computers to improve with experience and data

Deep Learning achieves great power and flexibility by:

1. Gathering knowledge from experience; humans don't formally specify the required knowledge
2. Being compositional, **representing** the world as a nested hierarchy of concepts [many layers, "Deep"]

Check this lecture on deep learning: CBMM Deep Learning tutorial:

<https://youtu.be/RTTQctLuTVk?t=507>

# Deep Learning is an approach to AI

- What is AI?

From Wikipedia:

- Intelligence demonstrated by machines
- The study of **intelligent agents**
- AI describes machines that **mimic human cognitive functions**

- Intelligence demonstrated by machines in contrast to the natural intelligence displayed by humans
- Agents: any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals
- human cognitive functions such as "learning" and "problem solving".

From Wikipedia

- *As machines become increasingly capable, tasks considered to require "intelligence" are often removed from the definition of AI, a phenomenon known as the [AI effect](#).*
- *For instance, [optical character recognition](#) is frequently excluded from things considered to be AI, having become a routine technology.*
- Modern machine capabilities generally classified as AI include successfully [understanding human speech](#),<sup>[6]</sup> competing at the highest level in [strategic game](#) systems (such as [chess](#) and [Go](#)),<sup>[7]</sup> [autonomously operating cars](#), intelligent routing in [content delivery networks](#), and [military simulations](#).

## Deep Learning is **an approach to AI**

- Early days of AI, it rapidly tackled and solved problems that were intellectually difficult for humans, but easy for computers
- True challenge: tasks that are easy to humans to perform but hard to describe formally
- DL focuses on solving these **intuitive problems** of AI

### Early AI

<http://www2.psych.utoronto.ca/users/reingold/courses/ai/early.html>  
[https://en.wikipedia.org/wiki/Logic\\_Theorist](https://en.wikipedia.org/wiki/Logic_Theorist)

### Challenge of more intuitive tasks

The Summer Vision Project <http://people.csail.mit.edu/brooks/idocs/AIM-100.pdf>

Check Dr. Fei-Fei Li's lecture on this:

<https://youtu.be/vT1JzLTH4G4?list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&t=855>

## Deep Learning is a type of Machine Learning

- ML allows computers to tackle decision-making problems **using knowledge/data from the real world**
- Depend heavily on the **representation** of the data
- DL is a kind of representation learning, and **gathers knowledge from experience**

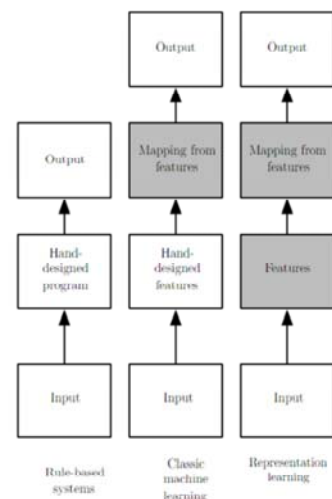


Figure from <https://www.deeplearningbook.org/>

These simple ML algorithms depend heavily on the representation of the data

We can see how representation is relevant also in everyday life: arithmetics on arabic numerals vs roman numerals

E.g. cesarean delivery problem, the algorithm takes certain pieces of information that represent the patient adequately for the problem ("features")

Logistic regression learns how these features correlate with various outcomes but it cannot influence how these features are defined

Sometimes hand-designed representations are easy to obtain, but often it is difficult to know what features to extract

E.g., detecting cars in photographs

Representation learning: part of ML, aims to learn the map from representation to output, but also the best representation

Learnt representations often perform better and make AI systems easier to adapt to new tasks

E.g., autoencoder



## Deep Learning is **compositional**

- Sometimes hand-designed representations are easy to obtain, but often it is difficult to know **what features to extract**
- DL solves this by introducing a **hierarchical** representation

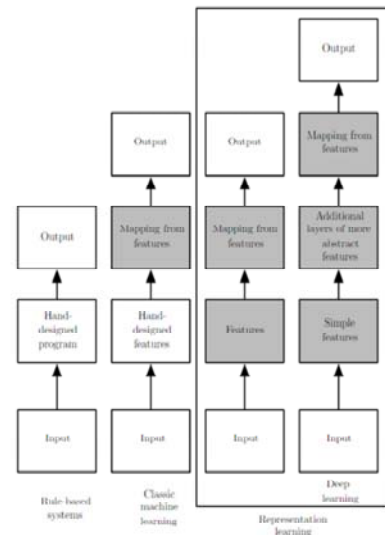


Figure from <https://www.deeplearningbook.org/>

It may also be that learning an adequate representation is as complicated as solving the original problem!

DL solves this by introducing a hierarchical representation

How does a computer go about understanding the meaning of raw sensory input data? An image is just a collection of pixel values

DL breaks down the mapping into a series of nested simple mappings, each described by a different layer of the network

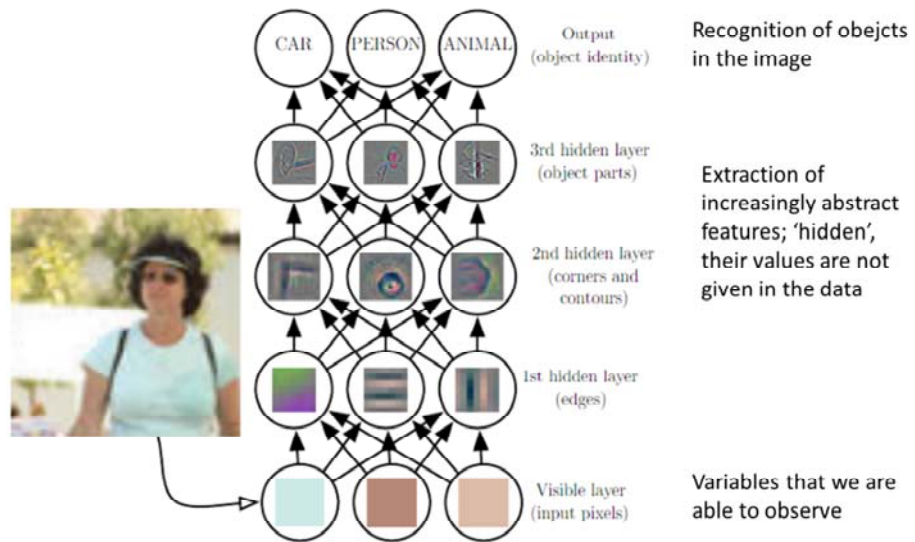


Figure from Deep Learning book [deeplearningbook.org](http://deeplearningbook.org), and there from Zeiler and Fergus (2014)

The flexibility of a hierarchical representation

A good intuition here:

[https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1\\_67000Dx\\_ZCJB-3p](https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3p)

## Recap

- DL is **an approach to AI**
- DL is **a type of ML**
- Why is DL well-suited to solve these more intuitive problems?
  - Gathers knowledge from **experience**
  - Represents world **hierarchically**

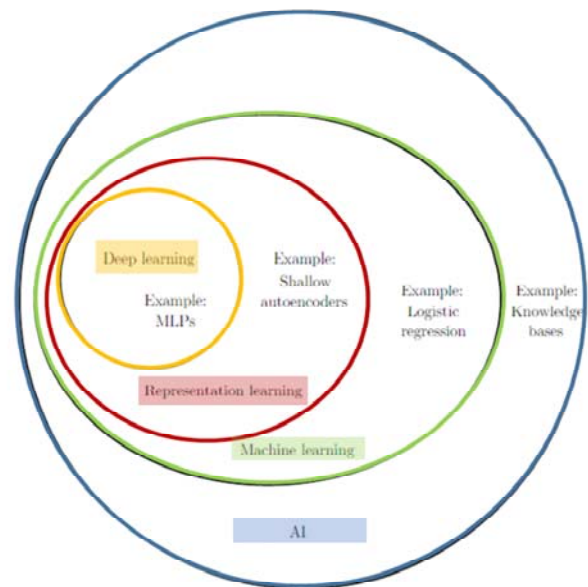


Figure from Deep Learning book, [deeplearningbook.org](http://deeplearningbook.org)

# Neural networks

Based on 3Blue1Brown

<https://www.youtube.com/watch?v=aircAruvnKk>

# What is a neural network?

- Deep learning and NN
- **Digit recognition task**
  - **Intuitive problem**, seems suitable for Deep learning
- We are going to describe how a **vanilla NN** solves this problem (and then implement it in the practical)

\*

Deep learning: a type of ML based on artificial neural networks

Introduce the idea of a NN as a stacking of linear transformations, with nonlinear layers in between

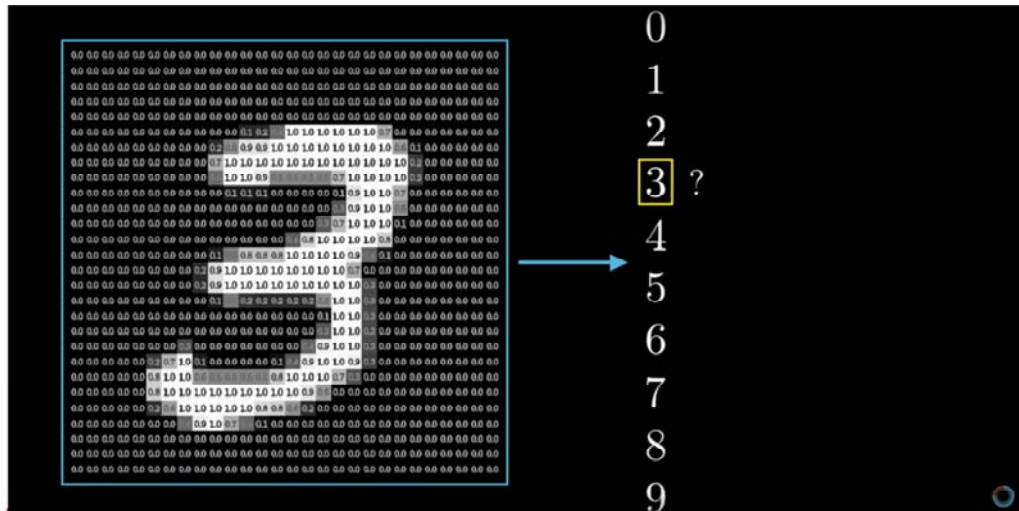
Deep learning relies on Deep NN, with many layers

For the sake of clarity we are going to focus on the problema of digit recognition, a classic example for neural network applications

Digit recognition is one of those intuitive tasks that we've talked about (easy to humans, hard to formalise)

We'll use this example to introduce the multilayer perceptron

Note that we focus on the classification task (but NN can also address regression and many other kind of tasks)



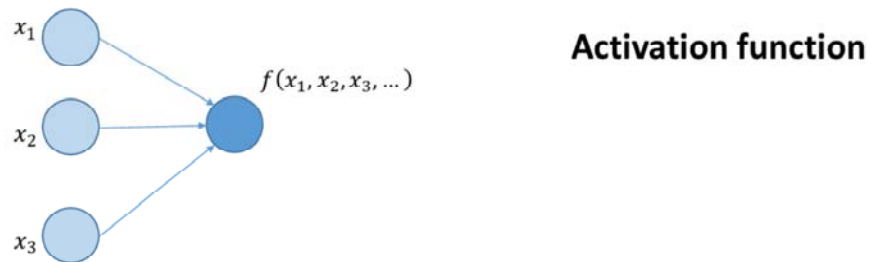
From 3Blue1Brown <https://www.youtube.com/watch?v=aircAruvnKk>

More specifically, the challenge is to write a program that takes grid 28x28 pixels and output a single number between 0 and 9

## Structure of a neural network

## Single neuron

Neurons in a network are organised in layers



### [Neurons and layers]

Neurons in a neural network are organised in layers (remember the hierarchy of concepts characteristic of DL).

- In this simple case we have an initial layer of 3 neurons and a second layer with only one neuron.

We can visualise each neuron as holding a number.

- The initial layer holds the input values.
- The neuron in the second layer holds the output of a certain function, that takes as input the output of all the neurons in the previous layer
- This function is called 'activation function'. There are different types of activation function and we will go into detail later

This simple structure called a simple perceptron.

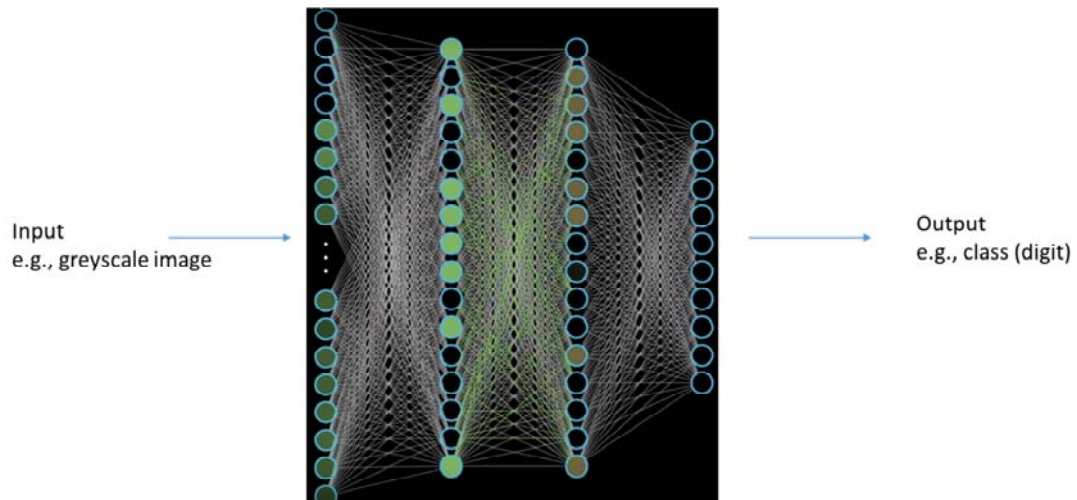
More about simple perceptron and multi-layer perceptron:

Deep Learning book sections 1.2.1 and 6.6

CBMM lecture <https://youtu.be/RTTQctLuTVk?t=136>



## Multi-layer perceptron (MLP)



### [MLP]

A multi-layer perceptron (MLP) is just an extension to more neurons and more layers

- MLPs have at least one layer between the first and the last layer, each of which with a relatively large number of neurons
- The number of layers and neurons varies per application and its often based on what has 'worked well' in the past
- However there are methods for comparing performance across different architectures to make choices

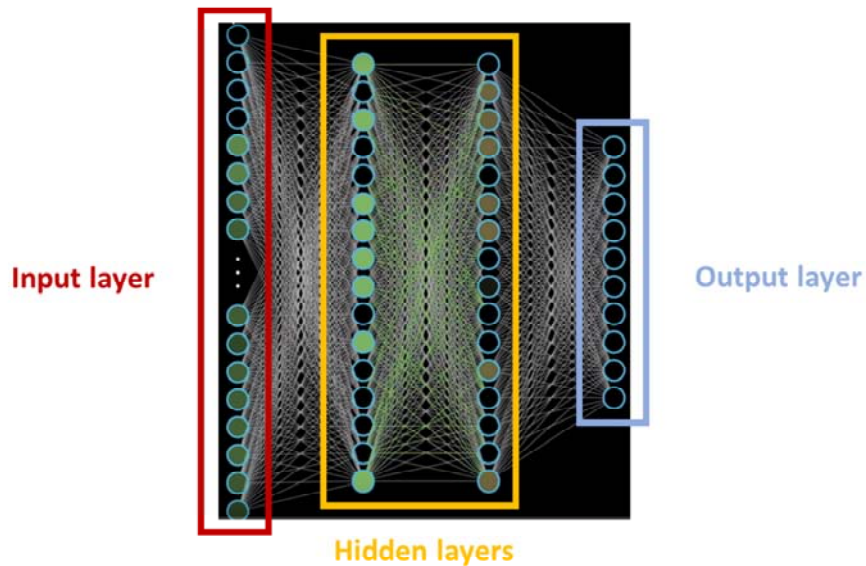
More on the network's architecture

<http://cs231n.github.io/neural-networks-1/#nn>,

Deep Learning book Ch6, 6.4 <https://www.deeplearningbook.org/contents/mlp.html>

Every neuron behaves exactly as we saw in the simpler case: at each layer after the input layer, each neuron receives as input all the neurons in the previous layer, computes a certain function, and outputs a value. This is carried out at every layer until the output layer. These networks are also called fully-connected, since all the neurons in one layer are conected to all previous ones

## Layers

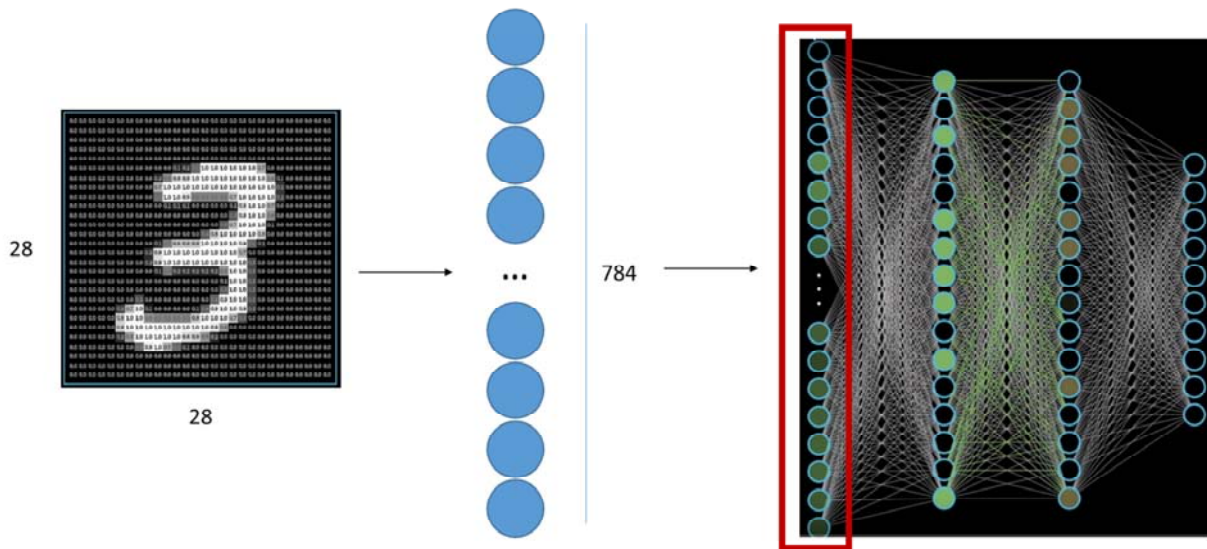


From this description of the workflow we can already see there are three types of layers

- The Input layer, that holds the input values
- Some hidden layers, where subsequent transformations occur
- Output layer, which holds the classification decision

We will address first the input and output layers before going into detail about what's going on in the hidden layers. For now, it's sufficient to know that these are transformations analogous to the one we've seen for a neuron but extended to many many neurons.

## Layers: input layer



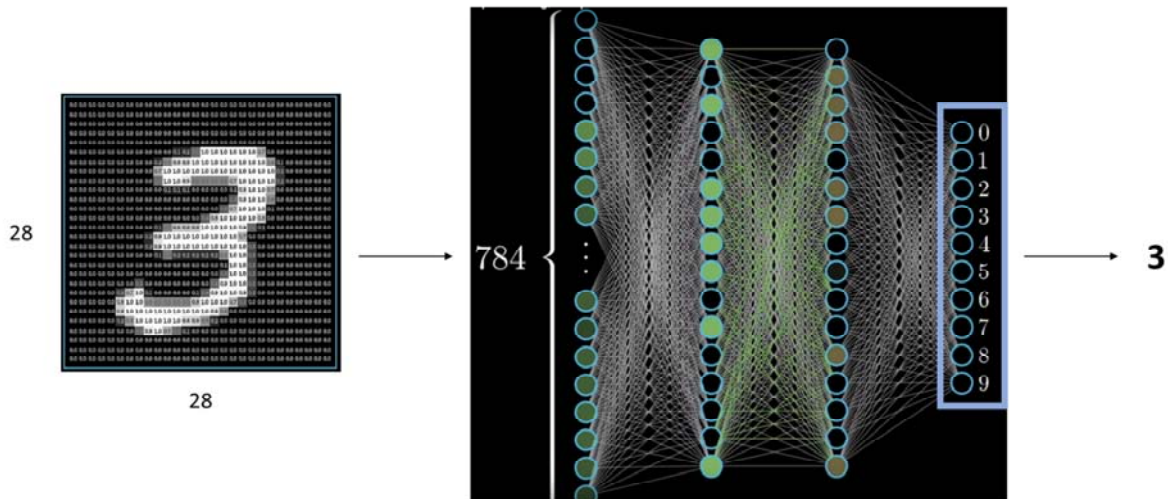
The input layer simply holds the input values.

We have seen that for our problem the input image is a grayscale image of size 28x28.

That means that we have an array of numbers organised in 28 rows and 28 columns, each of which with a value between 0 and 1, where 0 is black and 1 is White.

The first layer in our neural network will hold these values, stretched out in 784 different neurons (we reshape the 2D array as a vector)

## Layers: output layer



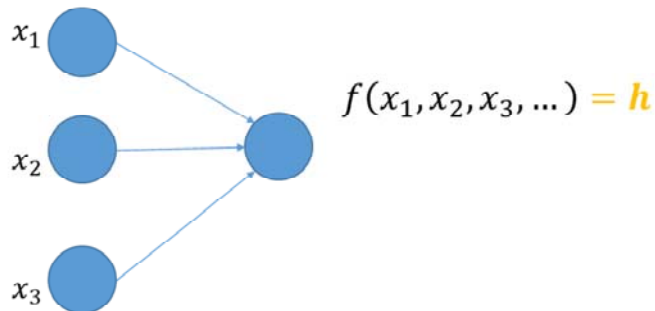
As we mentioned at the output layer we would like to have the classification decision. What digit is represented in the image?

- A convenient way to output this decision is to have as many neurons as classes we are considering.
- In this case any input image can be 1 in 10 classes, corresponding to the digits from 0 to 9.

Therefore in the output layer, each of the neurons would hold a number that represents how much the model thinks that the input image represents the corresponding class (digits in our case). [In an image classification task, it could be whether or not a cat, dog, car is present in the image]

Forward pass:  
transformations between layers

## From layer to layer: one neuron

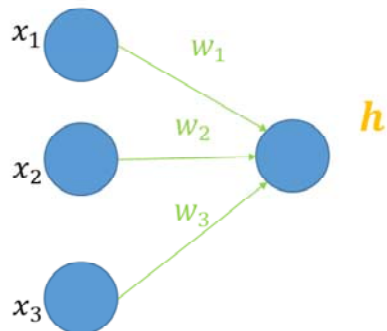


For the case of a single neuron, we have seen that each neuron holds a number that results from taking all the numbers held at the neurons in the previous layer and applying a certain function.

We consider a neuron at the first hidden layer. We represent its output by  $h$

We are now going to describe that function

## From layer to layer: one neuron



### 1. Weighted sum

$$w_1x_1 + w_2x_2 + w_3x_3 = \Sigma$$

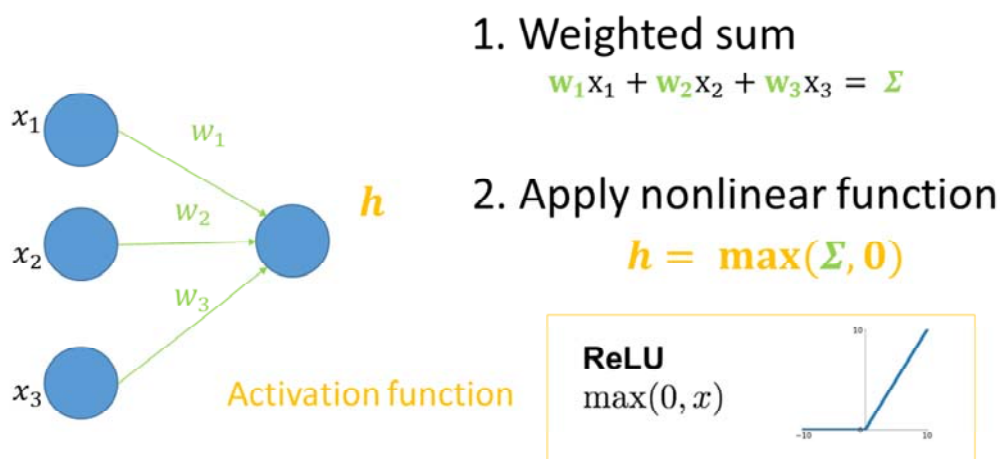
### 2. Apply nonlinear function

$$h = \max(\Sigma, 0)$$

The neuron processes the inputs as follows

- First it computes a weighted sum of the inputs. The weights represent the connections between the neurons. The weighted sum can be seen as the neuron attending to different parts of the input
- Then, we apply a nonlinear function. For example the rectified linear unit or ReLU function. This function will output the final output of the neuron

## From layer to layer: one neuron



- Choosing the relu function is not a totally arbitrary choice: the relu function is a common choice due to its properties when it comes to optimisation via gradient descent. However there are other popular options for activation functions
- Note as well that, a non linear function is required for the MLP to be a universal function approximator  
(along with a hidden layer with enough number of units/neurons)

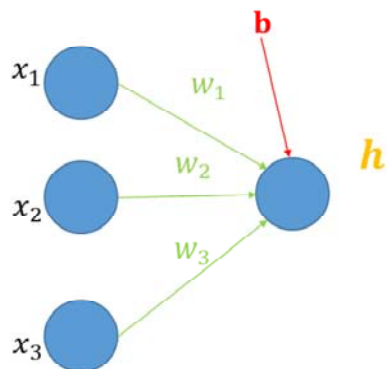
More on neural networks being universal function approximators

- Deep learning book section 6.4  
<http://www.deeplearningbook.org/contents/mlp.html>
- CS231n notes <http://cs231n.github.io/neural-networks-1/#power>
- Michael Nielsen's book <http://neuralnetworksanddeeplearning.com/chap4.html>
- The activation fn is loosely based on biological neurons;
- It represents that the artificial neuron will respond to inputs beyond a certain threshold.
- As it is now, this threshold would be zero: if the weighted sum is positive the neuron will output the result of the weighted sum, if it's negative it will output zero.



- What if we want a threshold different from zero? In order to “control” this threshold with a parameter, we introduce a bias term

## From layer to layer: one neuron



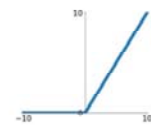
### 1. Weighted sum

$$w_1x_1 + w_2x_2 + w_3x_3 + b = \Sigma + b$$

### 2. Apply nonlinear function

$$h = \max(\Sigma + b, 0)$$

**ReLU**  
 $\max(0, x)$

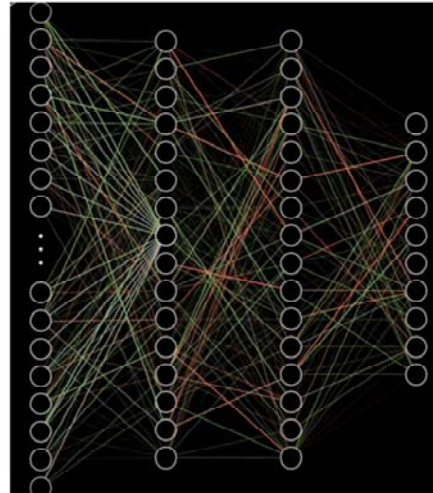


- With a bias term, the ReLU function will output the result of the weighted sum if its value is below  $b$ , and zero otherwise

Intuition: the neuron pays attention to certain inputs more than others, if they are large enough

## From layer to layer: matrix notation

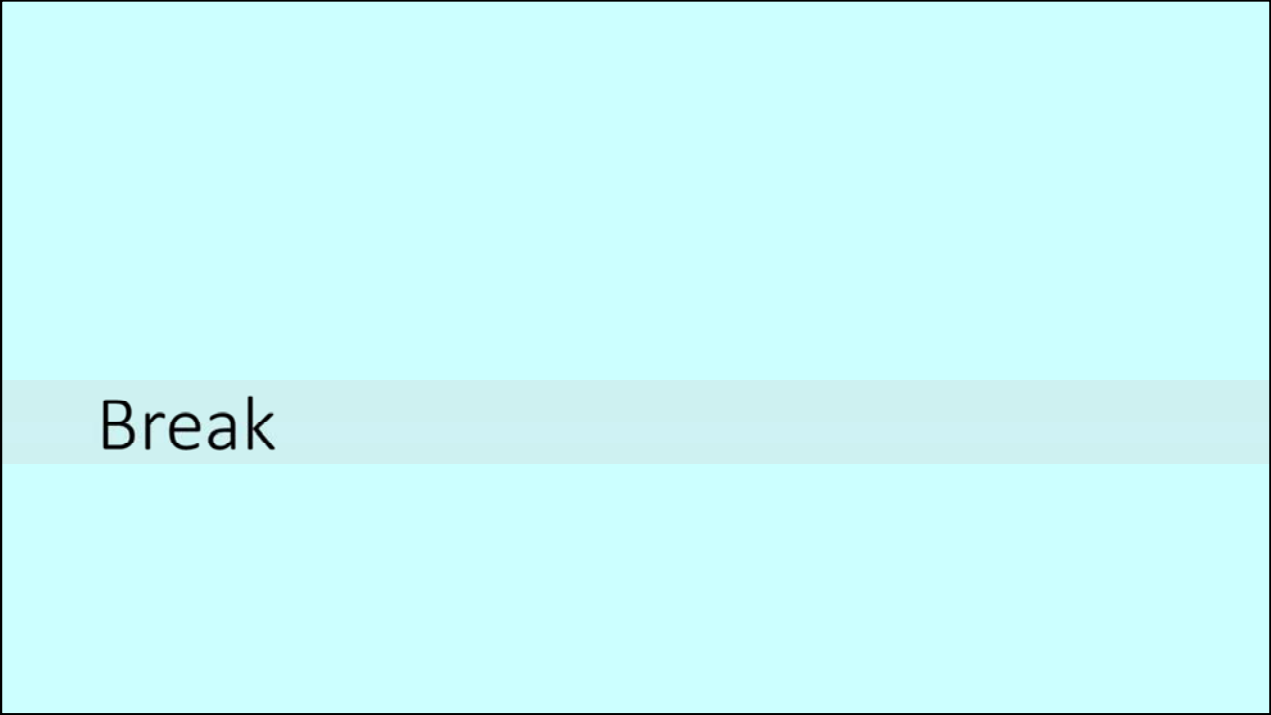
- In the network
    - Each **connection** has its own **weight**
    - Each **neuron** has its own **bias**
  - In this example MLP with
    - 784 neurons at the input layer
    - 16 neurons at the first hidden layer
    - 16 neurons at the second hidden layer
    - 10 neurons at the output
- = 13002 weights and biases!



If we extend what we've seen for one neuron to all of the neurons in our sample network, we get over 13000 parameters!

And this is just a small network....

- Can we write the transformations we've seen for one neuron for all the network, in a more compact form?
- We'll make use of matrix notation for that. If you have some knowledge of linear algebra that's excellent, if not it's just enough if you remember there is a compact way of writing these equations

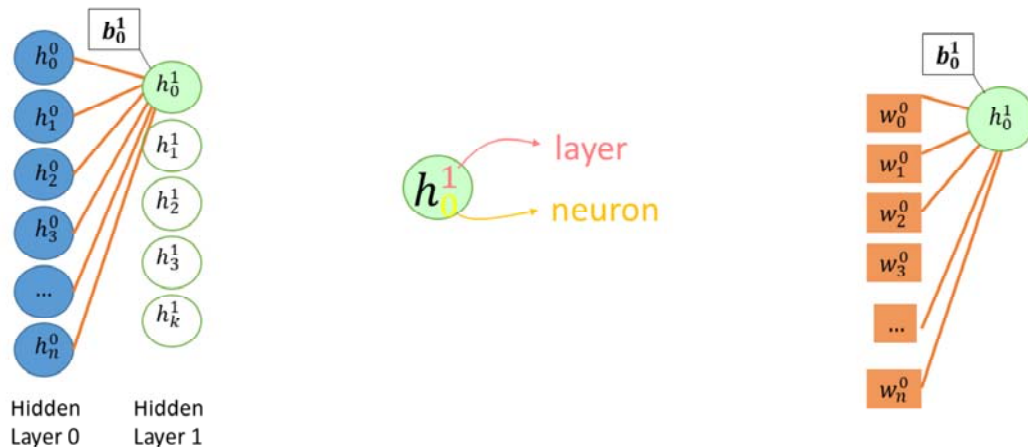


Break

30 min

## From layer to layer: matrix notation

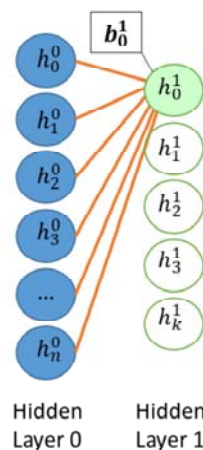
We can write the transformations that take place across the whole network in a more compact notation using matrices



Let's look again at how a neuron output is computed, this time with the neuron in its layer context.

- We consider two consecutive hidden layers
- We look at the first neuron in a certain hidden layer (layer 1) in Green:
- As we have seen already its output is computed from all the neurons in the previous layer (blue)
- Nomenclature:
- The neurons outputs are represented by  $h$  as a reference of them belonging to a hidden layer. The superscript refers to the layer the neuron belongs to, and the subscript is the index for each neuron in a layer
- The connections between all neurons in the layer 0 and the neuron in layer 1 we are examining are shown in orange; remember that for each connection we have an associated weight, which is the factor that multiplies the neuron output in the weighted sum
- We also have a bias for each neuron, which if you remember is independent of the input to that neuron

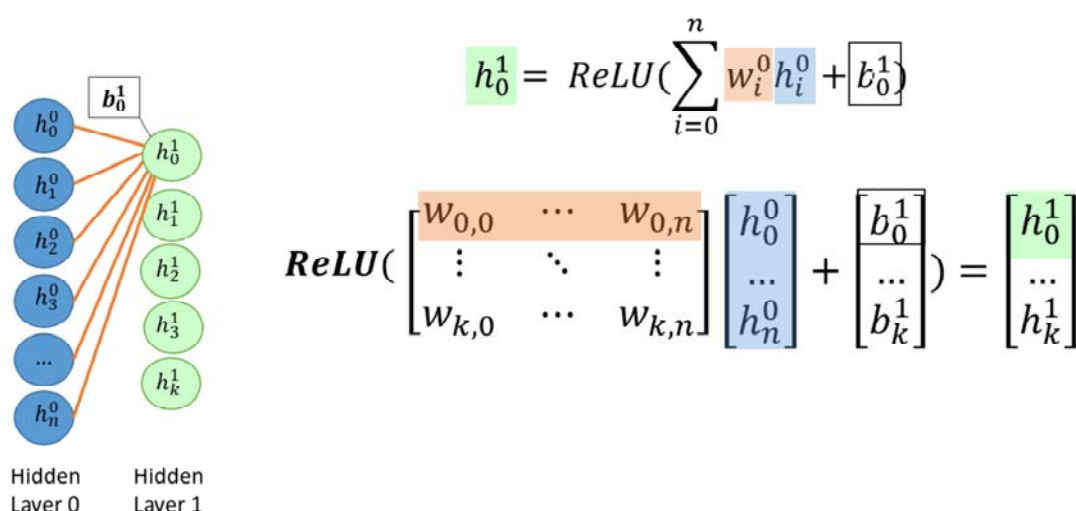
## From layer to layer: matrix notation



$$h_0^1 = \text{ReLU}\left(\sum_{i=0}^n w_i^0 h_i^0 + b_0^1\right)$$

We can express the transformation that takes place to obtain this neuron output as shown here. It just expresses the weighted sum over the neurons in layer 0, plus the neuron's own bias, and the output of which is passed through a ReLU function, that if you remember it just takes the max between 0 and the input value

## From layer to layer: matrix notation



Now we can use matrix multiplication to extend that expression to all the neurons in layer 1

If you remember matrix multiplication from linear algebra....you'll see the computation of the first neuron in layer 1 is equivalent to the one above (we multiply the first row and the first column to obtain the element at the first row and first column, etc.)

We now have a weight matrix, that collects all the weights representing the connections between all the neurons in layer 0 and all the neurons in layer 1. The first row in the weight matrix has the weights required to compute the output of the first neuron in layer 1, the second row those required to compute neuron 2, and so on until the k-th row to compute the k-th neuron. The vector that multiplies with this matrix collects the neurons in the previous layer.

To the result of this multiplication we add the bias term of each neuron in layer 1 and then feed the result to a ReLU function to obtain the final number held by the neurons in layer 1

## From layer to layer: matrix notation

Further compact: bias trick

$$\begin{bmatrix} w_{0,0} & \cdots & w_{0,n} \\ \vdots & \ddots & \vdots \\ w_{k,0} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} h_0^0 \\ \cdots \\ h_n^0 \end{bmatrix} + \begin{bmatrix} b_0^1 \\ \cdots \\ b_k^1 \end{bmatrix} = \begin{bmatrix} h_0^1 \\ \cdots \\ h_k^1 \end{bmatrix}$$

Equivalent to

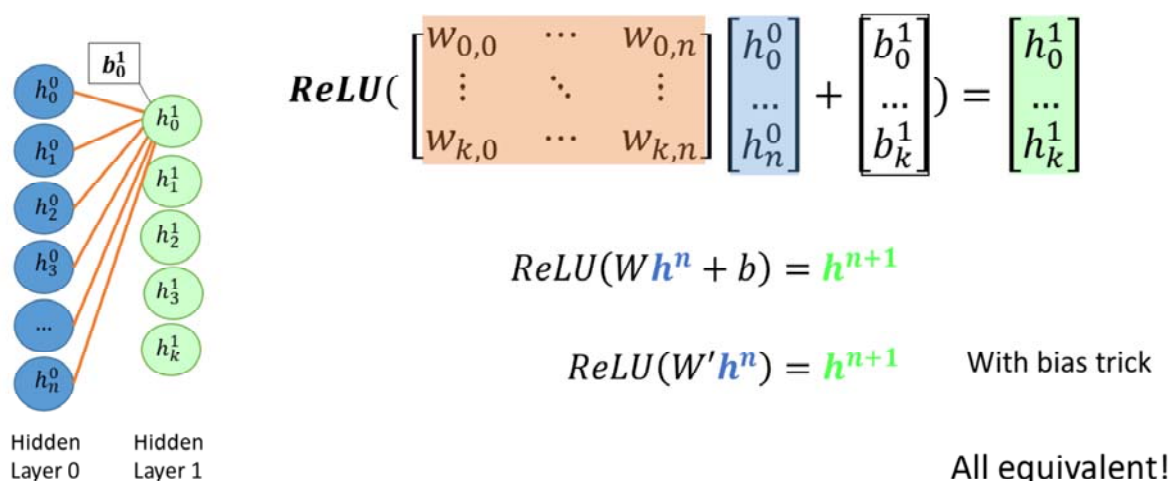
$$\begin{bmatrix} w_{0,0} & \cdots & w_{0,n} & b_0^1 \\ \vdots & \ddots & \vdots & \vdots \\ w_{k,0} & \cdots & w_{k,n} & b_k^1 \end{bmatrix} \begin{bmatrix} h_0^0 \\ \cdots \\ h_n^0 \\ 1 \end{bmatrix} = \begin{bmatrix} h_0^1 \\ \cdots \\ h_k^1 \end{bmatrix}$$

And just an additional note that often the bias vector is included in the weight matrix as an additional column for further compactness.

But note that if we adequately respect the dimensions and add a 1 as a last row in the vector of neurons in the previous layer, both expressions are equivalent



## From layer to layer: matrix notation

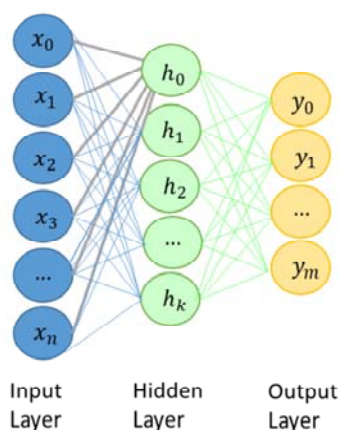


### [Recap]

Just remember that there is a compact way that is often preferred to express the transition from one layer to the next. And that often making use of this bias trick the bias term is omitted, but it's implicitly considered within the weight matrix as we've seen.

Expressing the transformations as vectorised operations is very convenient for computing efficiency (we have specialised hardware that performs this operations very fast)

## From layer to layer: matrix notation



For a **two layer** network (input  $x$  – hidden – output  $y$ )

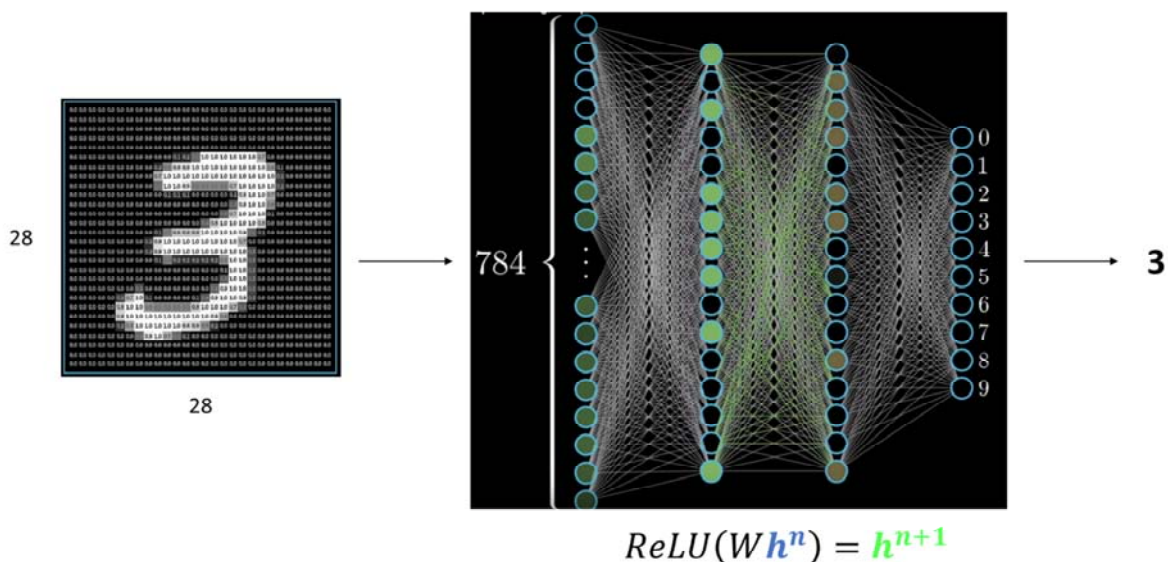
$$y = W_1 \text{ReLU}(W_0 x)$$

We will do this for as many layers we have in the network. This will result in a nested function. For example for a two layer network it would be something like in the slide.

Two things to note here:

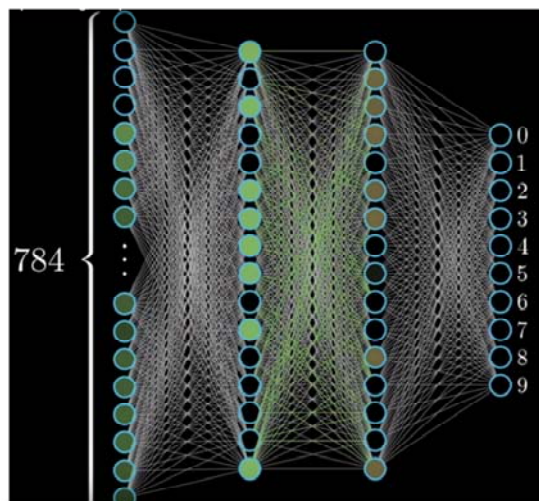
- first, when counting the number of layers in the neural network, we usually omit the input layer since it doesn't have tunable weights
- Second, Note that usually the non linear function is not applied at the last output layer (also called readout layer because it's where we read the predictions for the classes).

## From layer to layer: forward pass



Ok so we've now seen how we go from a certain input (in our case a greyscale image with a digit), across all layers applying the corresponding transformations, until the output layer, where we will obtain 'scores' representing how much the model thinks the input represents each class. This whole process is usually called forward pass, since the information flows from input to output

## How to choose the parameters?



$$\text{ReLU}(\boxed{W}h^n) = h^{n+1}$$

Network parameters

We've also seen that each layer has a collection of weights and biases that along with the activation function, define the transformation taking place across that layer. The weights and biases of all the layers constitute the network parameters, and they determine what the network "does".

With an ideal set of parameters, we would feed a certain image, for example one representing a 3, and the transformations across the layers should end up in the class "3" having the highest score in the output layer.

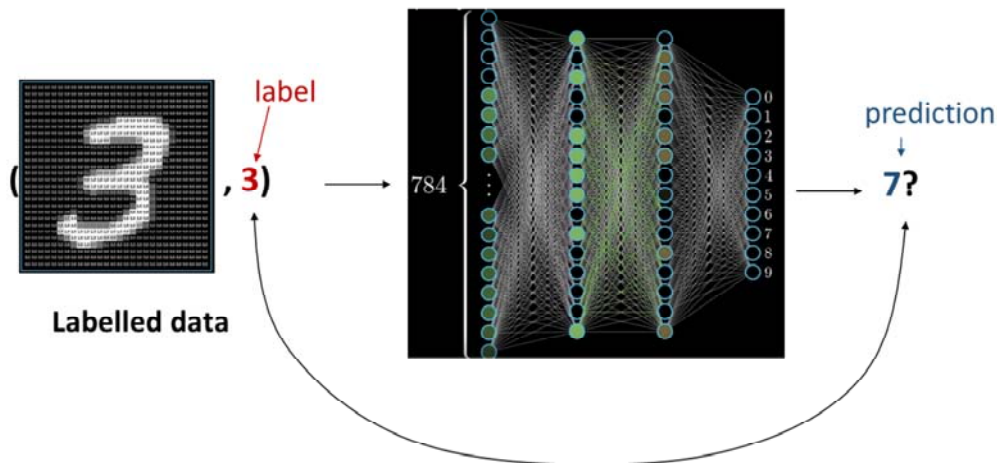
How do we choose these parameters? We would like to obtain the weights and biases that perform best in the task of recognising digits.

We can formulate this as an optimisation problem: we want to obtain the weights and biases that minimise the error when recognising digits

This is what we basically do when we train the network, which is the next part of the session

Training  
and backward pass

## Training: intuition



The main idea of training:

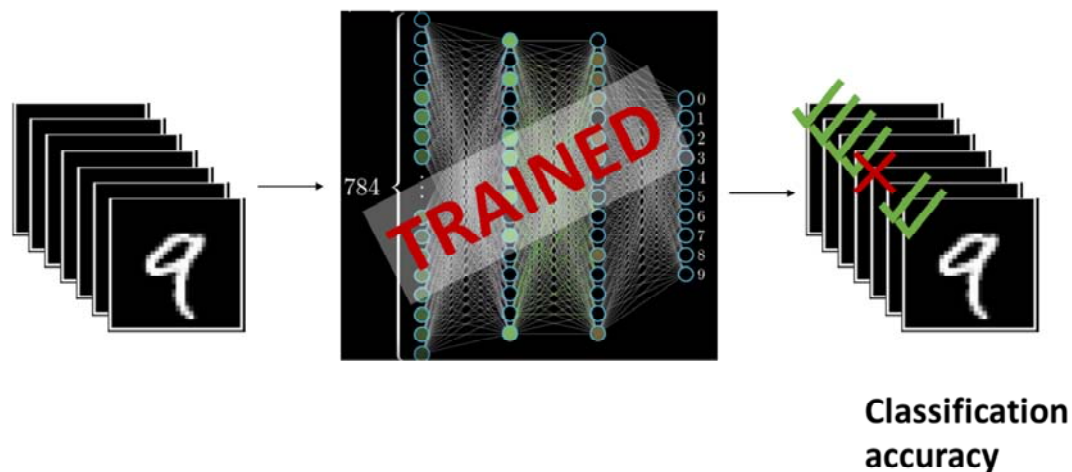
During training we will feed labelled data to the network. It is called 'labelled data' because for each image we have a label (in red) that contains the ground truth. This is basically the answer to the problem we want to solve (here, the digit it represents).

- Remember that with the neural network we want to capture that mapping from inputs to labels, from images to digits.
- We are focusing on supervised learning, in which labelled data is available, but be aware that there are other approaches to learning too.

For each training sample, the network will just execute a normal forward pass, ignoring the label. After going through all the layers, the network will output a prediction for the given input. Then we will compare it to the groundtruth label. Depending on how far off the prediction is, the network will adapt its weights and biases so that it improves its performance, and its predictions become closer and closer to the groundtruth

The complete set of images that we present during training constitutes the training set. The hope is that with this layered approach of the network and its hierarchical abstraction of concepts, we may be able to generalise beyond the training set.

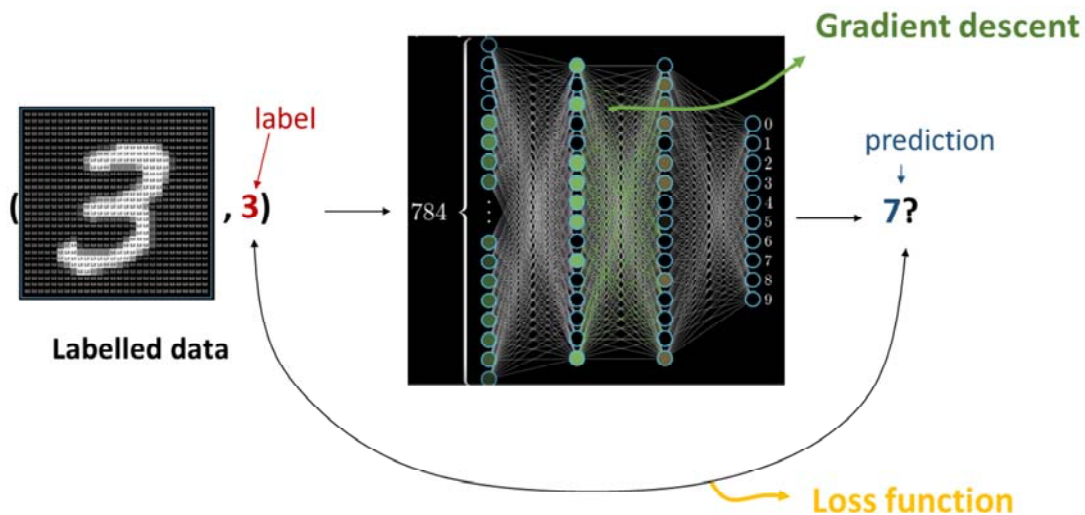
## Testing: intuition



The main idea of testing:

We test this by doing the following: after we train the network, we present it with a new set of images that the network hasn't seen during training. We call these the test set. On this test set we check how well it does by computing the accuracy of the classification (number of correct predictions over total).

## Training as an optimisation problem



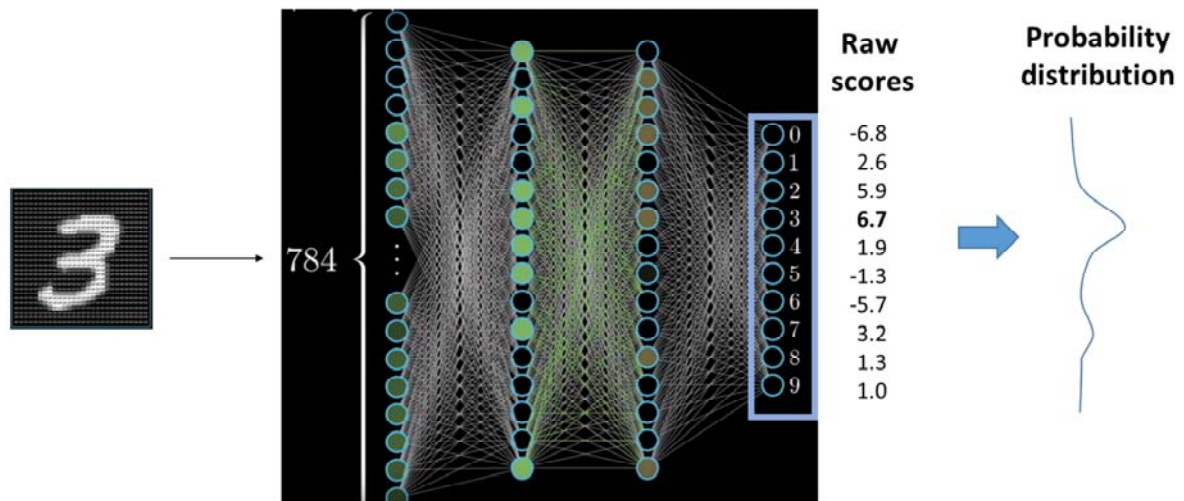
So we need to further define two aspects of this process:

- first, how is the comparison between prediction and labels carried out and how does it evaluate the performance of the network? We will capture this via the loss function
- Second, we need to specify a way of updating the values of the weights and biases based on how well or how badly the model is performing so that they perform better in future passes. We will approach this optimising their values with gradient descent

Let's start off with the loss function. In this lecture we will focus on the cross-entropy loss, which is a popular choice but there are many other alternative loss functions.



## Loss function



Ok so the loss fn describes how good a certain collection of weights and biases is. Sometimes called cost function, objective fn

To describe the loss function let's look at what happens at the end of a forward pass during training. After we go through all the layers we arrive at the output layer or readout layer, where the scores for each of the classes are collected in the neurons. We would like the score for the correct class, in this case "3", to be very high, and best highest than the other classes.

But these numbers at the output layer are just 'raw' scores. They are unnormalised (we interpret them as unnormalised log probabilities). It would be very nice for interpretability if these scores would actually represent a probability distribution across the classes, reflecting the network's prediction.

## Loss function

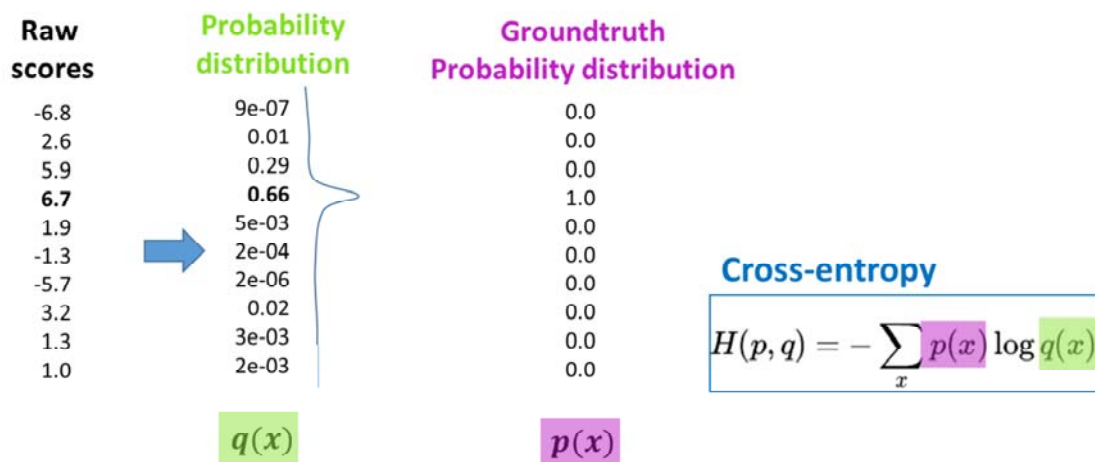
Raw scores	Probability distribution
-6.8	9e-07
2.6	0.01
5.9	0.29
<b>6.7</b>	<b>0.66</b>
1.9	5e-03
-1.3	2e-04
-5.7	2e-06
3.2	0.02
1.3	3e-03
1.0	2e-03

### Softmax

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

This is exactly what the softmax function does. It is a function that takes as inputs K real numbers and normalizes them into a prob distribution (in which each prob is prop to the exp of the score). So after the softmax each score will be now between 0 and 1 and all of them will add up to 1

## Loss function



Ok so now we have a probability distribution over the classes that represents the network's prediction. However because we have the labels, we actually know the true probability distribution, or the groundtruth. This distribution is "1" at the correct label class, and zero elsewhere.

How can we compare these two probability distributions? We can use tools from information theory to quantify how far off these distributions are. If we call the true probability distribution  $p(x)$  and the one estimated by the network  $q(x)$ , the cross entropy between them is defined as shown in the slide.

You can go into further detail about interpretation of cross-entropy but for now it's enough to know that it measures how far off the true and the estimated distributions are. This is great because we can already use this as our loss function! It indeed tells us how good or bad we are doing, which is what we were looking for.

Note that the cross-entropy function spits out a scalar (i.e., a number) for the probabilities we obtain for one input image.

More on cross-entropy and other information theory tools in Ch3 of the Deep

learning book

<https://www.deeplearningbook.org/contents/prob.html>

## Loss function

Loss  $L_i$ ,  
for one training sample

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

Score of the correct class

**Cross-entropy loss**

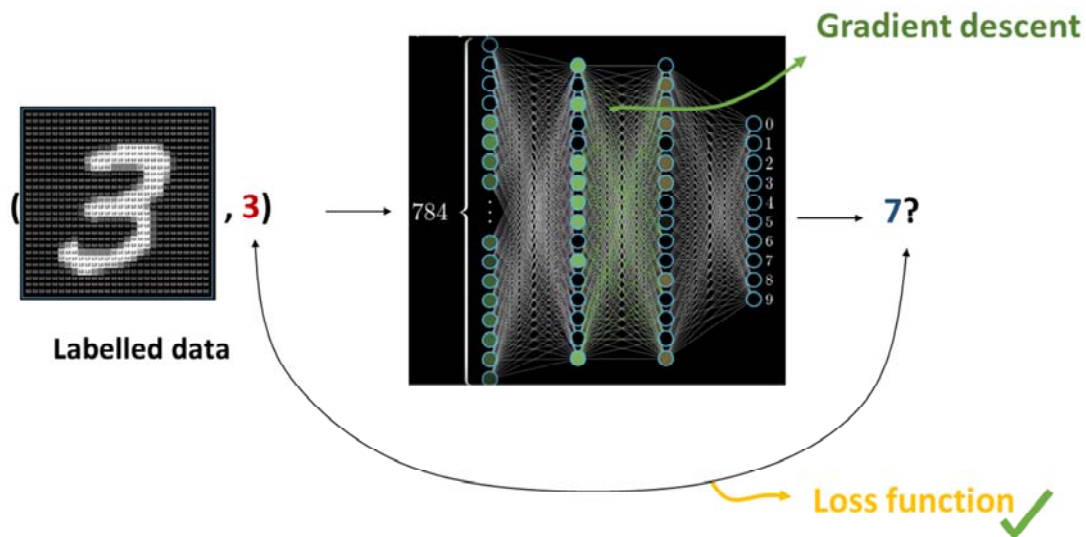
Full loss  $L$ ,  
for the complete dataset

$$L = \bar{L}_i$$

With a bit of reordering of the cross-entropy expression we can define the loss per input image as shown in the slide.

However the full loss of the complete dataset would be the average over the losses for each of the training samples. If the training samples are too many, so much that it slows down the training process, often a small portion of samples is considered, but we'll see that in more detail in a few slides.

## Training as an optimisation problem



Ok, so going back to the two aspects of training we wanted to further define:

- We have talked about the loss function and how it allows us to quantify how well/ how badly our network is doing, given a certain set of weights and a bunch of input images
- Now we are going to talk about the last aspect of the training process: the optimisation of the weights and biases, or how we go about adapting the parameters of the network to improve performance

# Optimisation: intuition

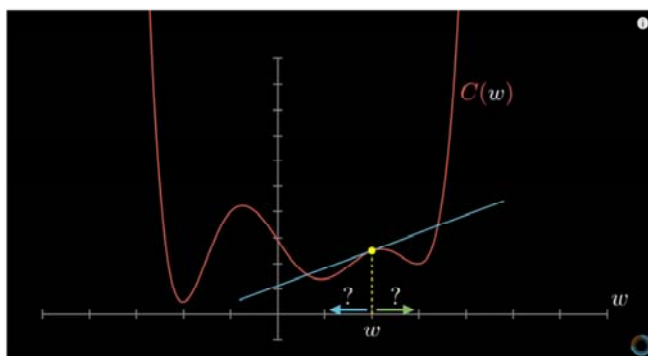
Video [https://youtu.be/HZwWFHWA-w?list=PL7HQObOWTQDNUU6R1\\_67000Dx\\_7CIB-3pi&t=275](https://youtu.be/HZwWFHWA-w?list=PL7HQObOWTQDNUU6R1_67000Dx_7CIB-3pi&t=275)

## Some things to point out ...

- Cost function = loss function
- SSE loss function

## Some things to pay attention to

- Effect of initialisation
- Iterative approach
- Step size proportional to the slope

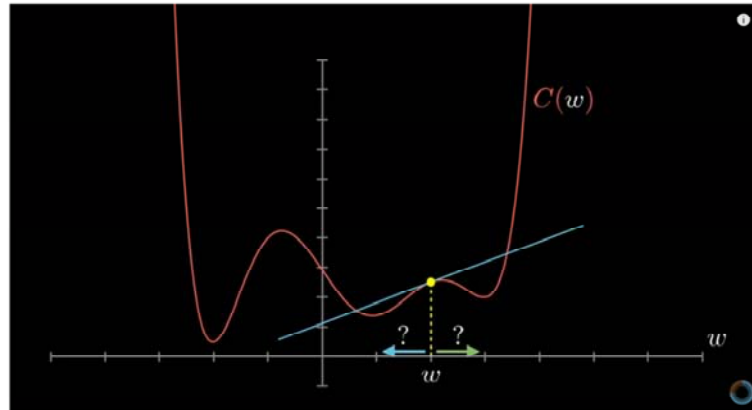


The video by 3Blue1Brown gives a nice intuition on gradient descent [until 6.55, “and that kind of helps you from overshooting”]

Before we watch the clip of the video, just some things to point it. In the video he uses the term cost function instead of loss function. And he also uses a quite simple loss function for his example (just sum of square error, not very common). We have defined the cross-entropy loss, which is far more common.

I also wanted to point out some things that are gonna come up in the video that are quite important to get the idea behind gradient descent. In the video the autor will comment on the effect of different initialisations, the iterative approach to finding the minimum of a function and the advantage of making the step size proportional to the slope. This may not make much sense now but pay attention on how he brings up these topics in his explanation

## Optimisation: intuition



- Ok so we used a simple case of a cost function with one input and one output to visualise the optimisation problem
- We've seen that we initialise in a random point, and then follow the derivative (i.e., the slope) which tells us in which direction should we move to decrease the function.
- We do that iteratively, that means that at every point we compute the slope, we take a small step following the slope in the adequate direction, and then repeat.
- We've also seen that if we move every time an amount proportional to the slope we will prevent overshooting the minimum, since as we approach the minimum the slope becomes flatter and flatter.



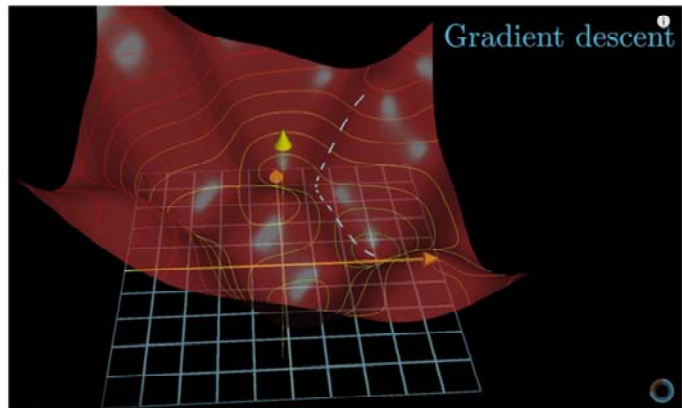
# Optimisation: intuition

## Video

[https://youtu.be/IHZwFHWa-w?list=PLZHQBOWTQDNU6R1\\_67000Dx\\_ZCJB-3pi&t=414](https://youtu.be/IHZwFHWa-w?list=PLZHQBOWTQDNU6R1_67000Dx_ZCJB-3pi&t=414)

Expanding to higher dimensions:

- Slope becomes **gradient vector**
- The gradient tells us **adjustments** in weights to minimise loss



Now we will expand this case to higher dimensions,

- first to a loss function that takes two inputs, and then to a loss function that takes many more inputs.

In higher dimensions, the gradient fulfills the role of the slope.

For those of you who remember from calculus, in a function of multiple inputs and one output, computing the gradient at a certain point tells us the direction in which the function increases most. So you may already see that for the loss function, whose inputs are the network's weights and biases, the gradient will be a vector that will tell us at each point, how much to vary the weights and biases so as to decrease more steeply the loss. But don't worry if this not clear for you yet, it is very nicely explained in the video [until 8.55, "which nudges to all of those numbers is gonna cause the most rapid decrease to the cost function"]

## 6 Main Optimisation Takeaways!

We now have a good intuition on how the optimisation problem is solved, and how we adjust the weights and biases in the network to minimise our loss. Let's consolidate what we've seen a bit more formally, in the form of 6 Main Optimisation Takeaways

## Optimisation take-aways

1. The loss function as a **high-dimensional “Surface”**, depending on the network’s weights and biases
2. We optimise the loss function with **iterative refinement**, starting off with a random set of parameters and adjusting them until the loss is below a certain threshold
3. **Gradient,  $\nabla L_W$**   
Is a Vector that at any point in that “surface” gives us the direction of steepest ascent (i.e., the direction in which the loss increases more).
4. The **negative of the gradient  $-\nabla L_W$**  gives us the direction of steepest descent (i.e., the direction in which the loss decreases more)

## Optimisation take-aways

### 5. Parameter update,

In its simplest form\*, it just follows the negative gradient direction.

$$W^{n+1} = W^n - \alpha \nabla L_W$$

New weights      Current weights      Learning rate      Gradient      (Step size proportional to the slope!)

The diagram shows the equation  $W^{n+1} = W^n - \alpha \nabla L_W$ . Below  $W^{n+1}$  is the text 'New weights' with a blue line pointing to it. Below  $W^n$  is the text 'Current weights' with a blue line pointing to it. Below  $\alpha$  is the text 'Learning rate' with a green arrow pointing to it. Below  $\nabla L_W$  is the text 'Gradient' with a blue arrow pointing to it. To the right of the equation is the text '(Step size proportional to the slope!)'.

We carry out an update everytime we complete the *full loss* (i.e across the *entire*\* training set). If costly, do over **batches** of training data.

### 6. Gradient Descent

Is the procedure of repeatedly evaluating the gradient and then performing a parameter update

Simplest form of parameter update: actually very vanilla and mostly never used, but it's the main idea behind all parameter updates. Others are momentum, rmsprop, nesterov momentum, Adagrad, Adam (see <http://cs231n.github.io/neural-networks-3/#update>)

I haven't got much into detail but note that a good initialisation is very relevant, and there are different ways of initialising networks that seem to work better than others. More on initialisation of weights: <http://cs231n.github.io/neural-networks-2/#init>



Break

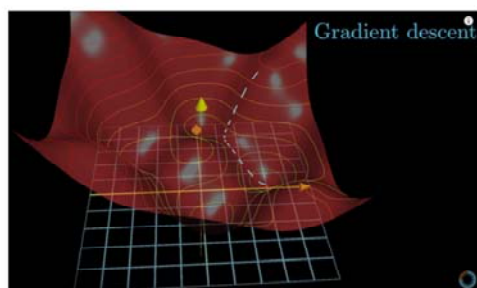
Start checking out the practical

Or have a look at <https://playground.tensorflow.org/>

# Optimisation

**How do we compute  $\nabla L_W$ ,**  
the gradient of the loss with respect to the network parameters?

- Two interpretations of  $\nabla L_W = \frac{\partial L}{\partial W}$ :



More geometric one

$$-\nabla C(\vec{W}) = \begin{bmatrix} 0.18 \\ 0.45 \\ -0.51 \\ \vdots \\ 0.40 \\ -0.32 \\ 0.82 \end{bmatrix}$$

More calculus one

Ok, so now we got a good idea of how this optimisation comes about . There is only one piece left

How do we compute the gradient of the loss with respect to the weights?

Note that we can interpret the gradient in two ways

- One is a more geometric approach, which is the one we've seen. In this high-dimensional "Surface" the gradient points in the direction of steepest ascent
- Another one is a more 'calculus' one. Each element of the gradient vector represents how sensitive the loss is to varying the corresponding weight. So if one element of the gradient vector is 3.2 and another is 0.1, that means that the first element will change the loss 32 times more than the second element. So basically the gradient vector tells us how much varying each weight affects the final loss.

# Optimisation





## Chain rule

derivative of composition of functions

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}.$$

## Backpropagation (or backward pass)

Algorithm to compute  $\nabla L_W$  efficiently

			3BLUE1BROWN SERIES S3 • E1 But what is a Neural Network?   Deep learning, chapter 1 3Blue1Brown
			3BLUE1BROWN SERIES S3 • E2 Gradient descent, how neural networks learn   Deep learning, chapter 2 3Blue1Brown
intuition	→		3BLUE1BROWN SERIES S3 • E3 What is backpropagation really doing?   Deep learning, chapter 3 3Blue1Brown
More math	→		3BLUE1BROWN SERIES S3 • E4 Backpropagation calculus   Deep learning, chapter 4 3Blue1Brown

This second interpretation helps us understand better how we compute the gradient of the loss function. We want to obtain how much does the loss at the very end of our network vary when we vary the weights?

The path from the weights at each layer to the loss is made up of a bunch of nested functions (more or less depending on how Deep the layer is).

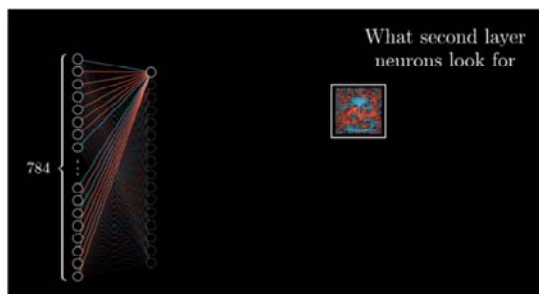
For those of you who studied calculus, we may remember we have a mathematical tool to compute the derivative of the composition of functions called the chain rule. The way we compute the gradient of the loss relative to the weights is by applying the chain rule, via an algorithm called backpropagation

I won't go into detail into backprop in this lecture, but I strongly recommend watching the videos from 3Blue1Brown as a primer, one with more intuition and one more mathsy. This is an important concept to understand especially if you'd like to build your own networks. Often the problems with training networks come from problems with gradients flowing. Bc of time and information overload I decided not to chop this video up in my explanation as before, but I very strongly recommend checking it to get a complete first view of neural networks

## A note on interpretability

Video [2 min 30s]

[https://youtu.be/IHZwWFHWa-w?list=PLZHQObOWTQDNU6R1\\_67000Dx\\_ZCIB-3pi&t=841](https://youtu.be/IHZwWFHWa-w?list=PLZHQObOWTQDNU6R1_67000Dx_ZCIB-3pi&t=841)



<https://xkcd.com/1838/>

Going back to that intuition on the layers of the network picking up on edges, corners, and building up on object elements....  
Until 16.36 ["the less intelligent it seems"]

Sometimes is not very clear what this networks are learning... that's why we often hear the term 'black box' associated to them. But there is actually plenty of research in interpretability/explainability and what the networks pay attention to

Research on explainability: <https://www.oreilly.com/learning/introduction-to-local-interpretable-model-agnostic-explanations-lime>



# Practical

Build an MLP for digit recognition



Sample images from the MNIST dataset

Keras: The Python Deep Learning library



<https://colab.research.google.com/drive/1wKkUEDkN6RLctivEXX19UF9otc8vpa15>

Based on tutorial at:

<https://victorzhou.com/blog/keras-neural-network-tutorial/> (Solved)

We are going to build a simple network like the one we've described using Keras.

Keras is a high-level neural networks API, written in Python , which runs on top of Tensorflow. It was developed aiming for fast experimentation and the idea is that it allows fast prototyping. However it becomes quite limited when you want to design more "custom" networks.

We will also use colab, which is a shareable scientific notebook, similar to jupyter but a bit more flexible in terms of sharing. It's a very popular tool in data science.

More info about how to use colab: <https://www.geeksforgeeks.org/how-to-use-google-colab/>

We will follow the tutorial by Victor Zhou for Keras beginners. I'll suggest you try filling in the gaps in the colab for a bit but ofc if you get very stuck you can always check the solution at that link

Importing in colab

[https://colab.research.google.com/notebooks/snippets/importing\\_libraries.ipynb](https://colab.research.google.com/notebooks/snippets/importing_libraries.ipynb)

# Practical



Show keyboard shortcuts:  
Ctrl/Cmd + M + H

Add cell above: Ctrl+M A

Add cell below: Ctrl+M B

Selecting and hovering over methods  
shows help!

Sections

Table of contents

- MNIST Problem
- Setup and imports
- Inspecting the dataset

Code snippets

Files

Following the excellent Keras beginners' tutorial by Victor Zhow at on <https://victorzhow.com/deep-neural-network-tutorial/>.

- MNIST Problem

We are now familiar with the digit recognition problem: given an image of a handwritten digit, we want the model to classify it as the actual digit represented in it. The name MNIST refers to the dataset, [Modified National Institute of Standards and Technology](https://en.wikipedia.org/wiki/MNIST). It is a classic machine learning problem and the type deep learning approaches excel at.

- Setup and imports

This colab runs Python 3, so we just need to make sure we install the required packages

```
[ ] !pip install -q mxnet
```

We should now be able to import the required packages

```
[ ] import mxnet as mx
import mnist
import keras
```

- Inspecting the dataset

Cells

# Practical

## Recommended:

Coding a simple neural network with Numpy

<https://victorzhou.com/blog/intro-to-neural-networks/>

```
import numpy as np

def sigmoid(x):
    # Our activation function: f(x) = 1 / (1 + e^(-x))
    return 1 / (1 + np.exp(-x))

class Neuron:
    def __init__(self, weights, bias):
        self.weights = weights
        self.bias = bias

    def feedforward(self, inputs):
        # Weight inputs, add bias, then use the activation function
        total = np.dot(self.weights, inputs) + self.bias
        return sigmoid(total)

weights = np.array([0, 1]) # w1 = 0, w2 = 1
bias = 4 # b = 4
n = Neuron(weights, bias)

x = np.array([2, 3]) # x1 = 2, x2 = 3
print(n.feedforward(x)) # 0.9998889488855994
```

If you liked the tutorial I recommend trying this one at home at your own pace. It seems daunting but it's really not that challenging. Excellent for understanding backprop!

I'd recommend writing out every line of code yourself (as in not copy-pasting directly), so that every line makes more sense and slowly sinks in

## Further reading

**Recommended exercise:** coding a simple neural network with numpy

<https://victorzhou.com/blog/intro-to-neural-networks/>

Assignment #1 from <http://cs231n.github.io/>

**Recommended read** (more advanced)

Yes you should understand backprop, Andrej Karpathy

<https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b>

## Further reading (in no particular order....)

- Cross validation: <http://cs231n.github.io/classification/>
- Regularisation; <http://cs231n.github.io/neural-networks-2/>
  - Occam's razor: [http://neuralnetworksanddeeplearning.com/chap3.html#overfitting\\_and\\_regularization](http://neuralnetworksanddeeplearning.com/chap3.html#overfitting_and_regularization) [excellent]
  - Dropout <http://cs231n.github.io/neural-networks-2/#reg>
- Bias – variance tradeoff: Deep Learning book Ch5, 5.4.4 <https://www.deeplearningbook.org/contents/ml.html>
- Batch normalisation <http://cs231n.github.io/neural-networks-2/>
- Stochastic Gradient Descent:
  - Andrew Ng [excellent] <https://www.youtube.com/watch?v=-4Zi8fCZO4>
- Babysitting the training process <http://cs231n.github.io/neural-networks-3/>
- Initialisation <http://cs231n.github.io/neural-networks-2/>
- Optimizers/parameter updates <http://cs231n.github.io/neural-networks-3/#update>
- Hyperparameter tuning <http://cs231n.github.io/neural-networks-3/#hyper>
- Gradient checks <http://cs231n.github.io/neural-networks-3/#gradcheck>
- Dimension analysis <http://cs231n.github.io/optimization-2/#mat>
- Overfitting and generalising: Deep Learning book Ch5
- Computational graphs
- Network architecture: <http://cs231n.github.io/neural-networks-1/#nn>, Deep Learning book Ch6, 6.4 <https://www.deeplearningbook.org/contents/mlp.html>

Further reading: some topics I haven't covered in detail in this session and some resources

# Useful resources

## Main resources for this lecture

- 3Blue1Brown  
[https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1\\_67000Dx\\_ZCJB-3p](https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3p)
- CS231n Convolutional neural networks **[excellent]**  
<http://cs231n.stanford.edu/>
- Deep Learning book Ch 1 <https://www.deeplearningbook.org/contents/intro.html>
- Center for Brains, Minds and Machines CBMM  
<https://www.youtube.com/watch?v=RTTQctLuTVk>
- **For the practical**  
<https://victorzhou.com/blog/keras-neural-network-tutorial/>

# Useful resources

## Other useful/fun resources

- *Neural Networks and Deep Learning*, Michael Nielsen <http://neuralnetworksanddeeplearning.com/>
- A Recipe for Training Neural Networks <http://karpathy.github.io/2019/04/25/recipe/>
- Chris Olah's blog <https://colah.github.io/>
- Distil <https://distill.pub/>
- Tensorflow and deep learning - without a PhD by Martin Görner Devovx 2016  
<https://www.youtube.com/watch?v=qyvt7kiQol&feature=youtu.be>
- AIMS CDT courses
- AlphaGo movie in Netflix
- Neural networks as Software 2.0 <https://medium.com/@karpathy/software-2-0-a64152b37c35>
- Research on explainability: <https://www.oreilly.com/learning/introduction-to-local-interpretable-model-agnostic-explanations-lime>
- Interpretability: <https://towardsdatascience.com/interpretability-in-machine-learning-70c30694a05f>

## Feedback forms

<https://docs.google.com/forms/d/e/1FAIpQLSfXHSNrEB2Oan4P4F3HtW92BvLOUYfbKaxHB-MRdV3sHydHCQ/formResponse>



See you next week!

