INTRODUCTION TO DATA SCIENCE

28/01/2020

WFAiS UJ, Informatyka Stosowana
I stopień studiów

MACHINE LEARNING

- Basic terminology
- Classical approaches to prediction
- Bias-variance trade-off
- Introduction to Neural Networks

Some plots from

[4] T. Hastie, R. Tibshirani, J, Friedman, *The Elements of Statistical Learning (2nd ed.)*, Springer Series in Statistics, 2001

Basic terminology

The goal of machine learning is to predict results based on incoming data.

Features (also parameters, or variables): these are the factors for a machine to look at. E.g.: carthesian coordinates, pixel colors, a car mileage, user's gender, stock price, word frequency in the text.

- Quantitative (x = {1.02, 0.21, 0.12, 2})
- Qualitative *discrete* (x = {medium, small, large}) or *categorical* (x={red, blue, green})

Algorithms (also models): Any problem can be solved in different ways. The method you choose affects the precision, performance, and size of the final model.

- If the data is insufficient/inapproriate (e.g. statistically limited or missing important features), even the best algorithm won't help. Pay attention to the accuracy of your results only when you have a good enough dataset.

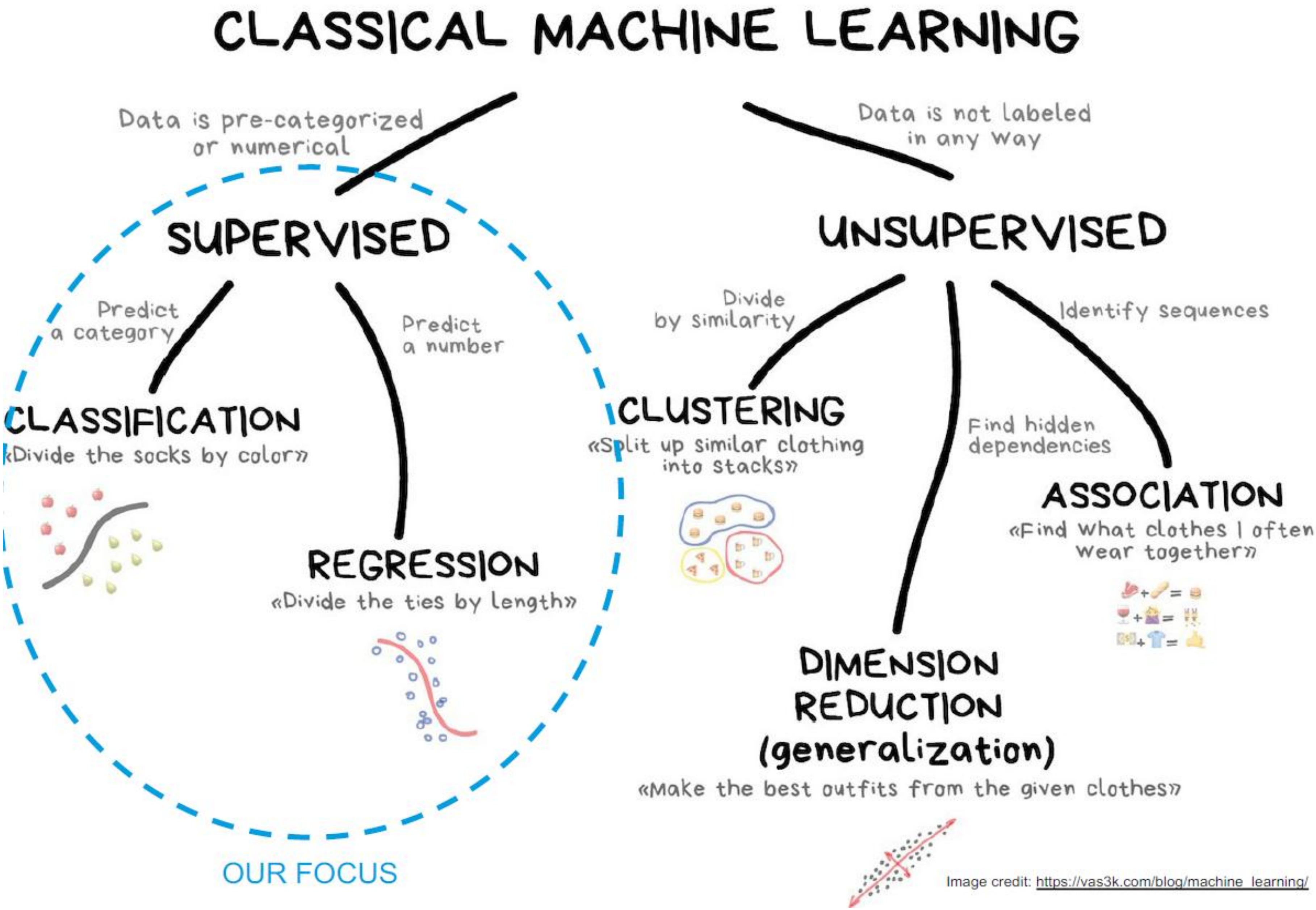


Image credit: https://vas3k.com/blog/machine_learning/

Prediction: Least squares

The linear model is one of our most important tools in statistics.

- Given a vector of inputs $X^T = (X_1, X_2, \ldots, X_p)$, we predict the output Y via

$$\hat{Y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j$$

- The term β_0 is the intercept, also known as the *bias* in machine learning

How do we fit the linear model to a set of data?

- The most popular method is the method of least squares: pick the coefficients β to minimize the residual sum of squares (RSS)

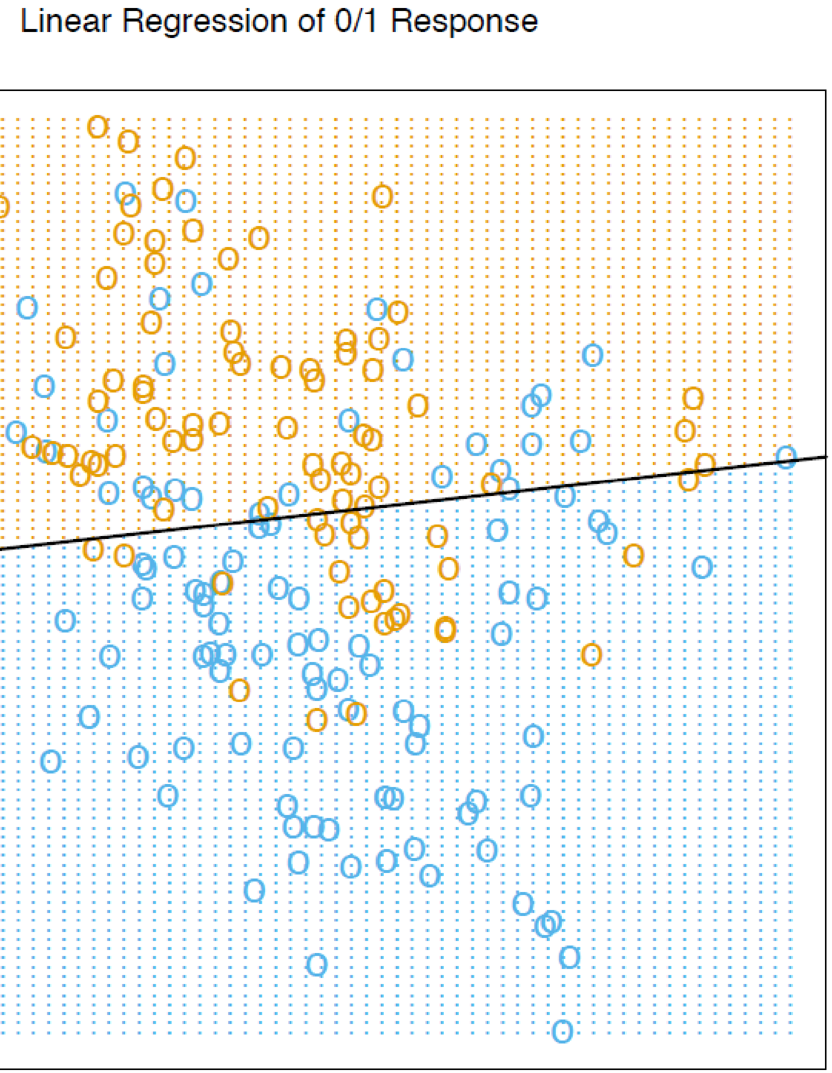
$$\mathrm{RSS}(\beta) = \sum_{i=1}^{N} (y_i - x_i^T \beta)^2$$

- $\mathrm{RSS}(\beta)$ is a quadratic function of the parameters, and hence its minimum always exists, but may not be unique.

Prediction: Least squares

Data were simulated with outputs being either BLUE or ORANGE.

A linear regression model was fit to the data, used here as *training dataset.*



The fitted values Y are converted in a classification according to

$$\hat{G} = \begin{cases} \texttt{ORANGE} & \text{if } \hat{Y} > 0.5 \\ \texttt{BLUE} & \text{if } \hat{Y} \leq 0.5 \end{cases}$$

Prediction: nearest neighbor classifier

An alternative algorithm for classification is the method of nearest neighbors.

Nearest-neighbor methods use those observations in the training set closest in input space to x to form Y.

The k-nearest neighbor fit for Y is defined as:

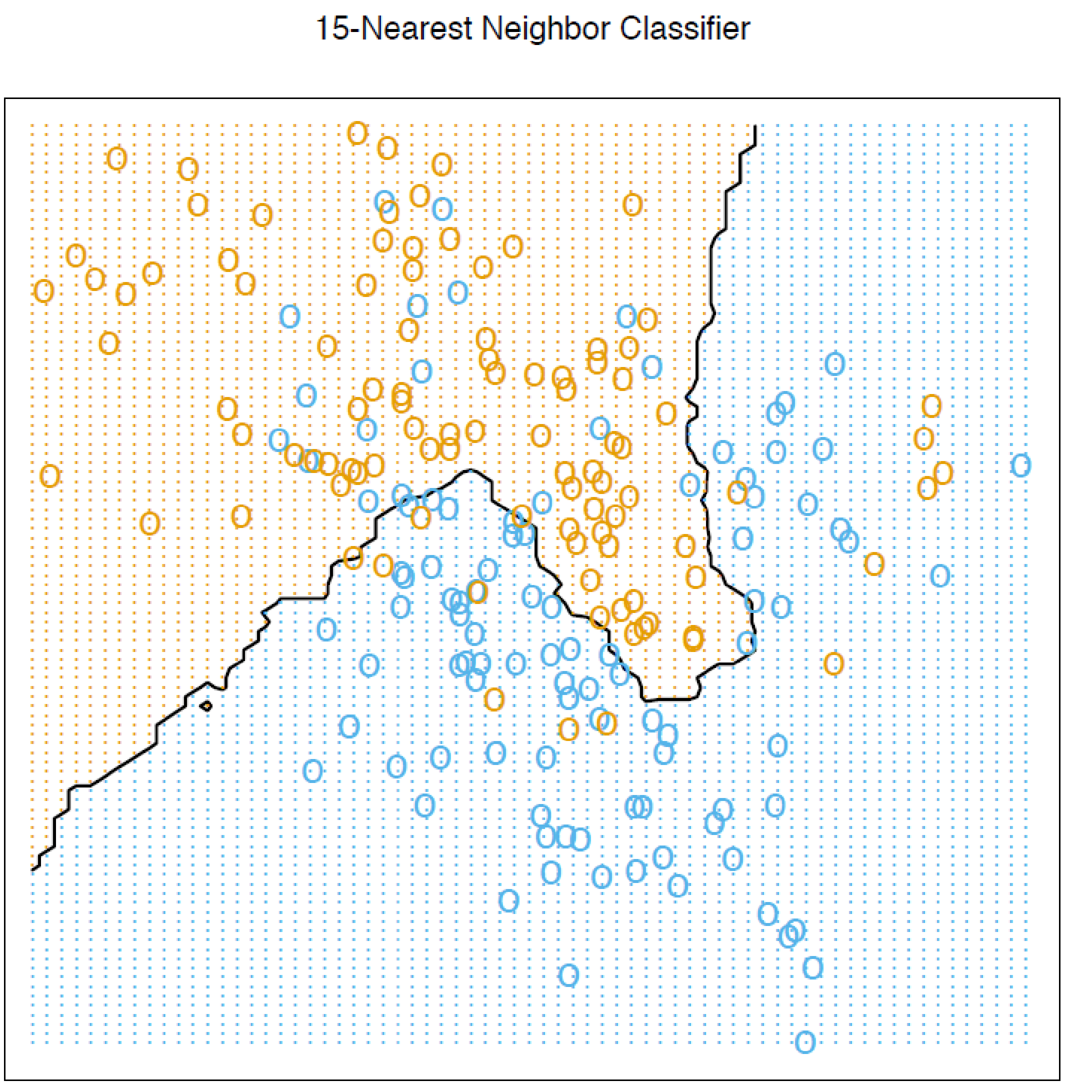
$$\hat{Y}(x) = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i$$

where $N_k(x)$ is the neighborhood of x defined by the k closest points x_i in the training sample.

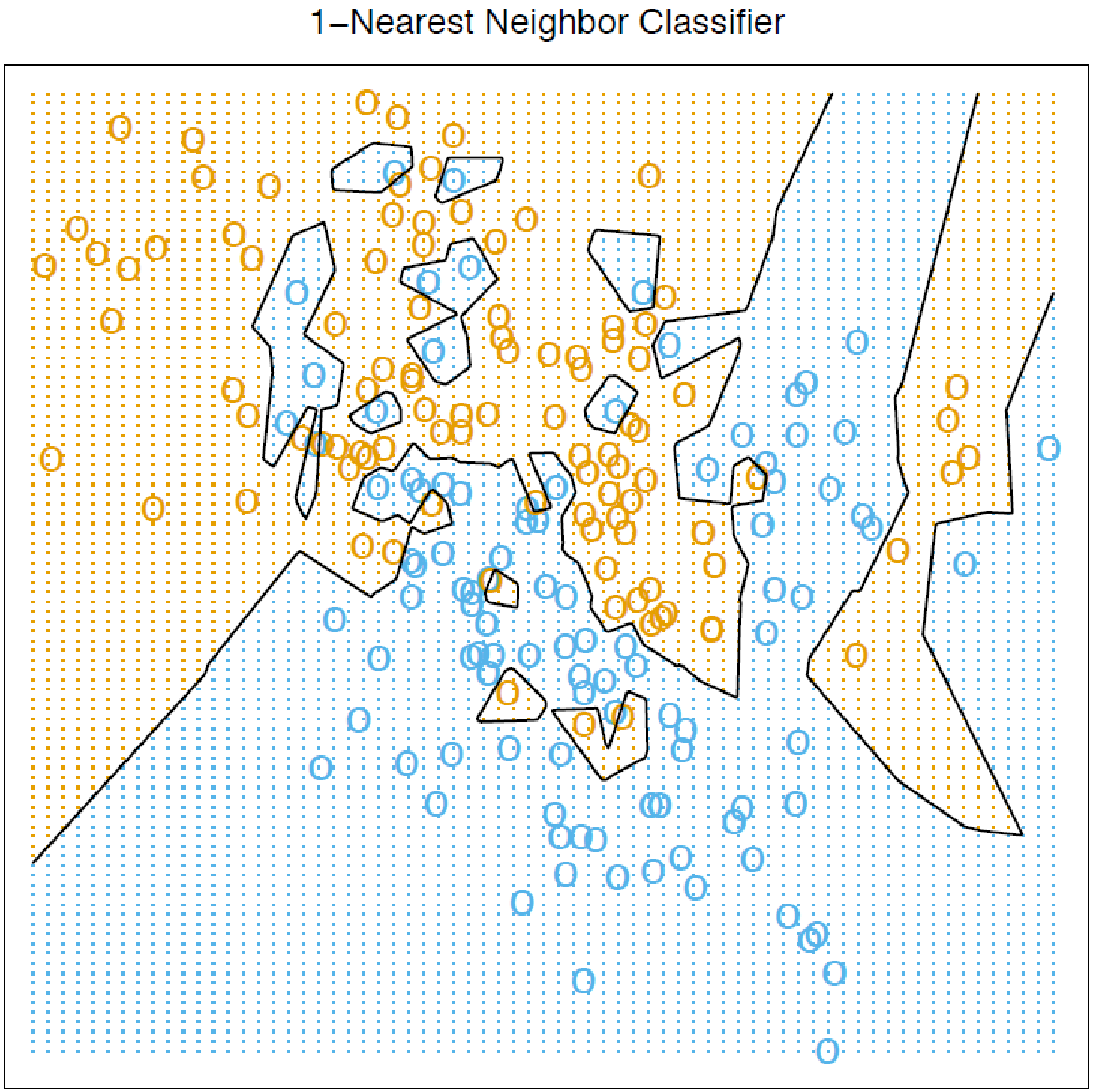
Closeness implies a metric, which in our case we assume is Euclidean distance.

In words, we find the k observations with xi closest to x in input space, and average their responses.

Prediction: nearest neighbor classifier



Perfect classification?



Comparison

To compare the different algorithms, let's define a *loss* (or cost) criterion.

- Here, we can take the rate of misclassifications

In order to compare the performances, let's introduce a second, independent, dataset to evaluate the performance: the *test dataset.*

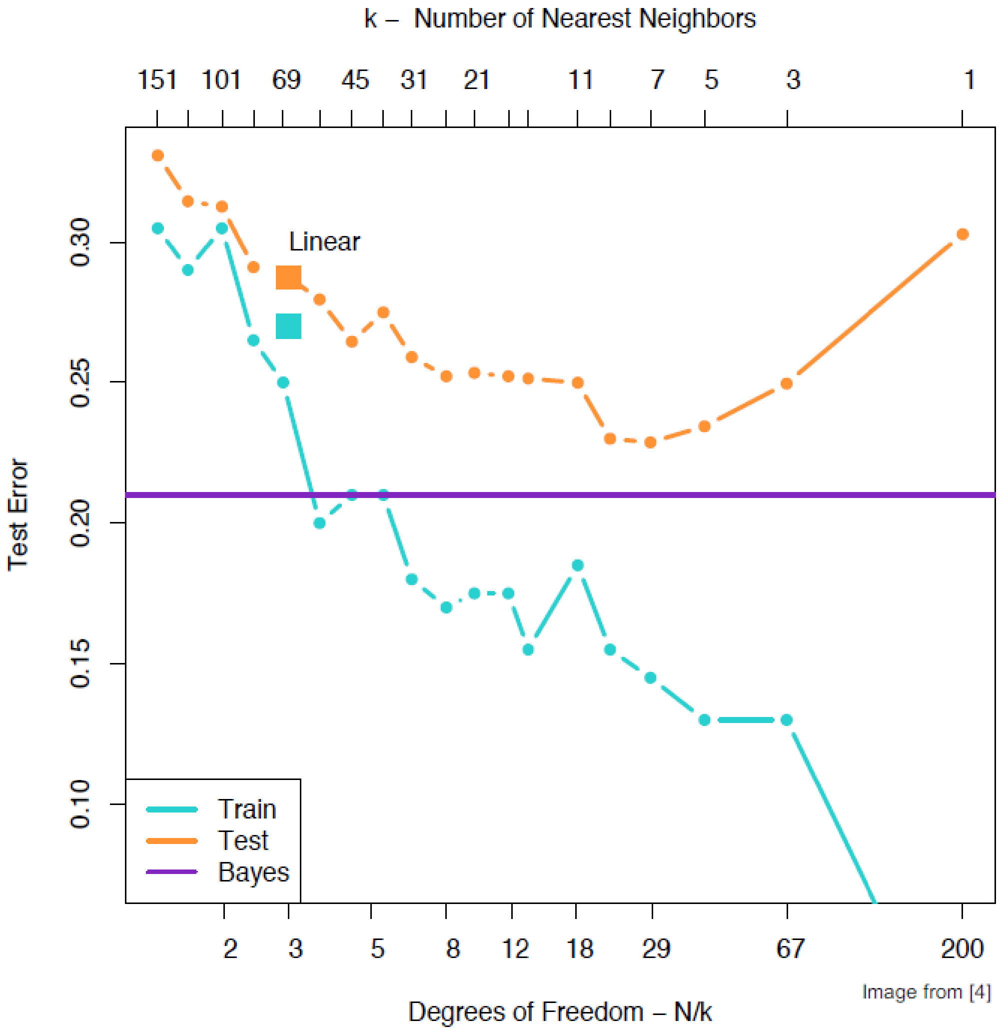


Image from [4]

Bias-variance tradeoff

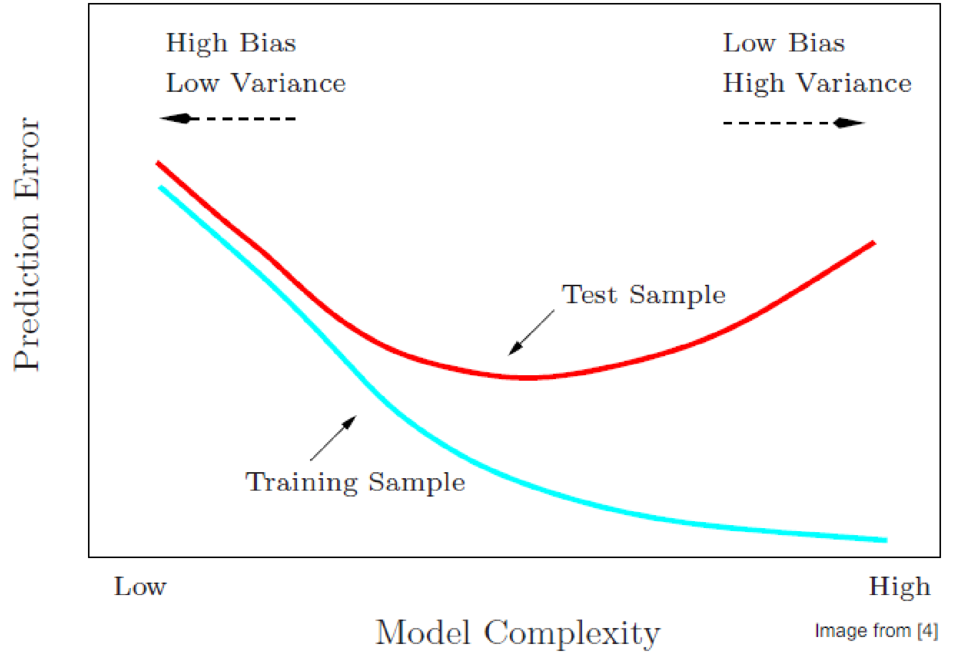


Image from [4]

The training error tends to decrease whenever we increase the model complexity, that is, whenever we fit the data harder.

- With too much fitting, the model adapts itself too closely to the training data, and will not generalize well (i.e., have large test error).
- In contrast, if the model is not complex enough, it will underfit and may have large bias, again resulting in poor generalization.

Where are the neural networks?

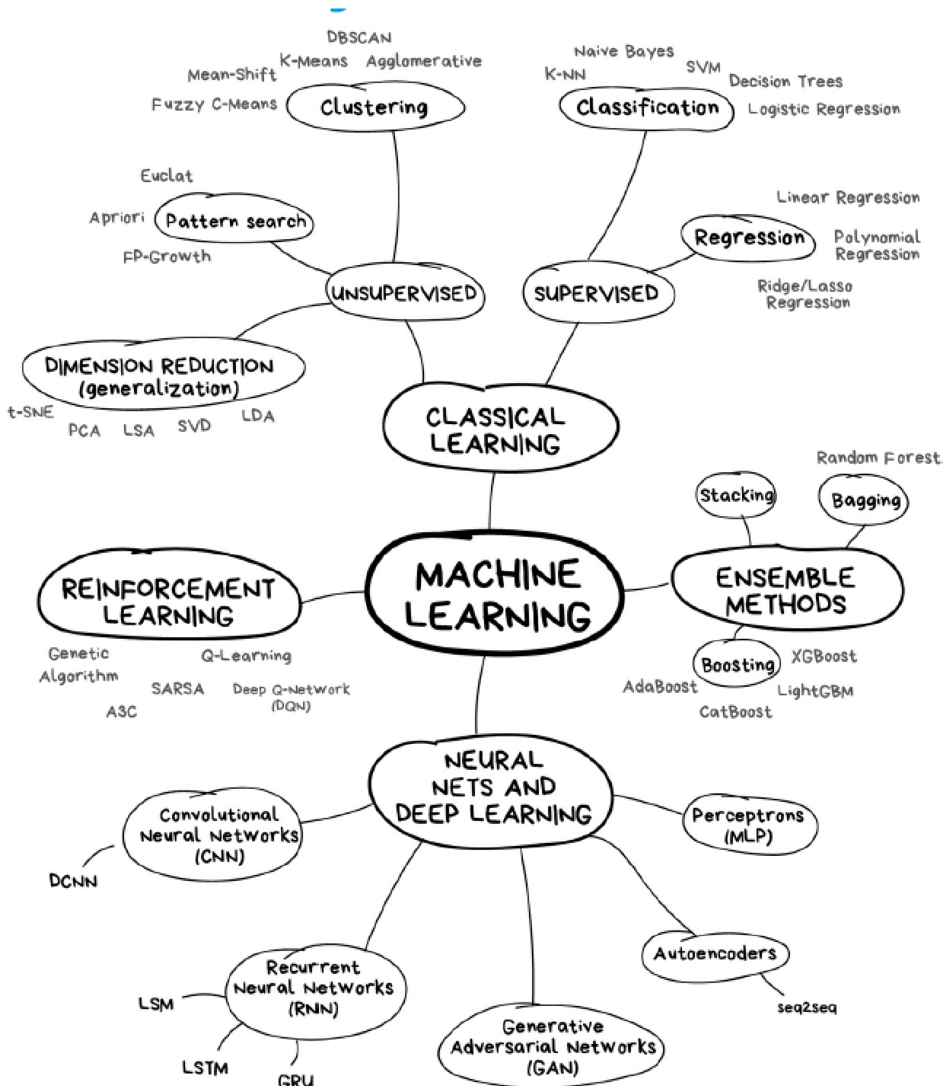


Image credit: https://vas3k.com/blog/machine_learning/

Neural networks

Any neural network is a collection of **neurons** and **connections** between them.

Neuron is a function with a set of inputs and one output. Its task is to take all numbers from its input, apply a function on them and send the result to the output.

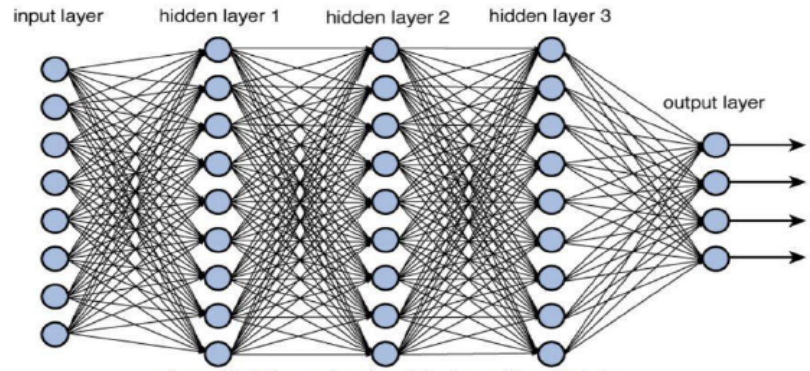
- Example: sum up all numbers from the inputs and if that sum is bigger than N give 1 as a result. Otherwise return zero.

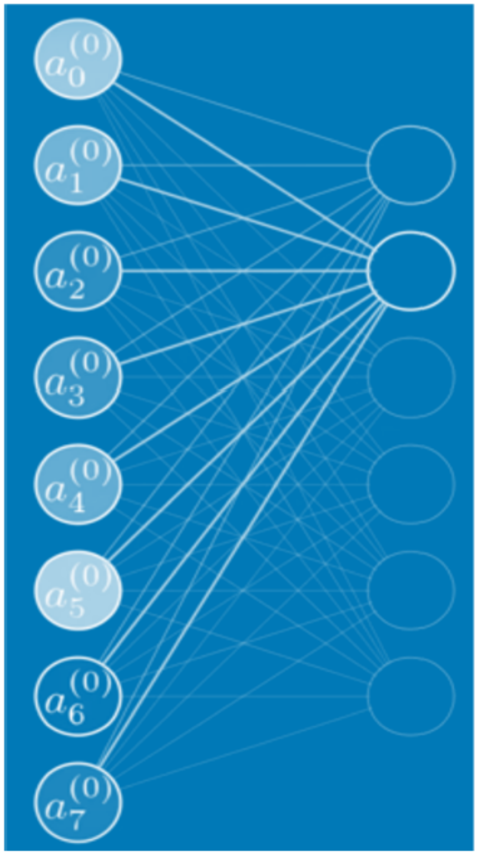
Connections are like channels between neurons. They connect outputs of one neuron with the inputs of another so they can send digits to each other. Each connection has only one parameter the *weight*.

- These weights tell the neuron to respond more to one input and less to another. Weights are adjusted when training — that's how the network learns.

How do NNs work?

How do NNs work?





layer

$$a_0^{(1)} = f\,(w_{0,0}\, a_0^{(0)} + w_{0,1}\, a_1^{(0)} + \ldots + w_{0,n}\, a_n^{(0)} + b_0)$$

activation function

weights

bias

$$\begin{bmatrix} a_0^{(1)} \\ \ldots \\ a_n^{(1)} \end{bmatrix} = f\left(\begin{bmatrix} w_{0,0} & \cdots & w_{0,n} \\ \vdots & \ddots & \vdots \\ w_{k,0} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ \ldots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ \ldots \\ b_n \end{bmatrix} \right)$$

How do NNs learn?

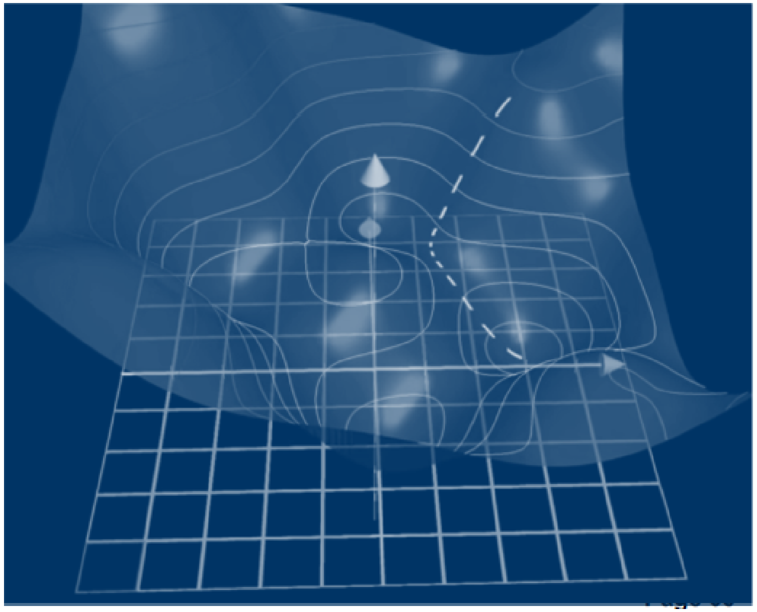
After we constructed a network, our task is to **assign proper weights** so neurons will react correctly to incoming signals.

- define a loss function to measure how far the response is from the truth

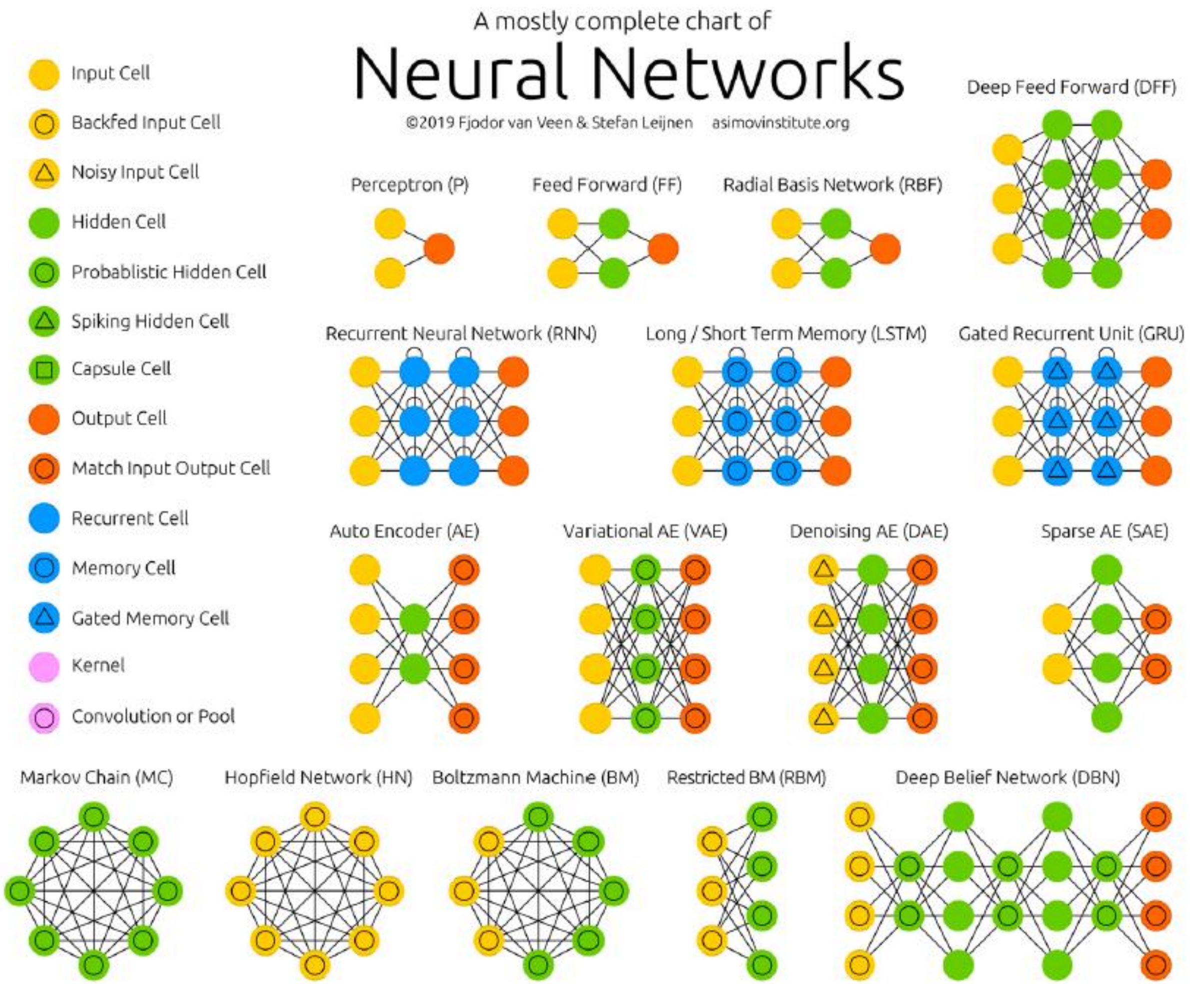
This function is a function of all the weights and biases in the NN (a priori a very large number), and the goal of training is to find its minimum.

- To start with, all weights are assigned randomly.
- After evaluating the NN on the training dataset, we can compute all the per-neuron differences with respect to the correct result.
- Computing the gradient of the loss, gives us a direction in which to tune the weights towards a local minimum

The process of correcting the weights is called backpropagation an error.



How do NNs learn?



There are many more…

Deep-learning Neural Network

TensorFlow™

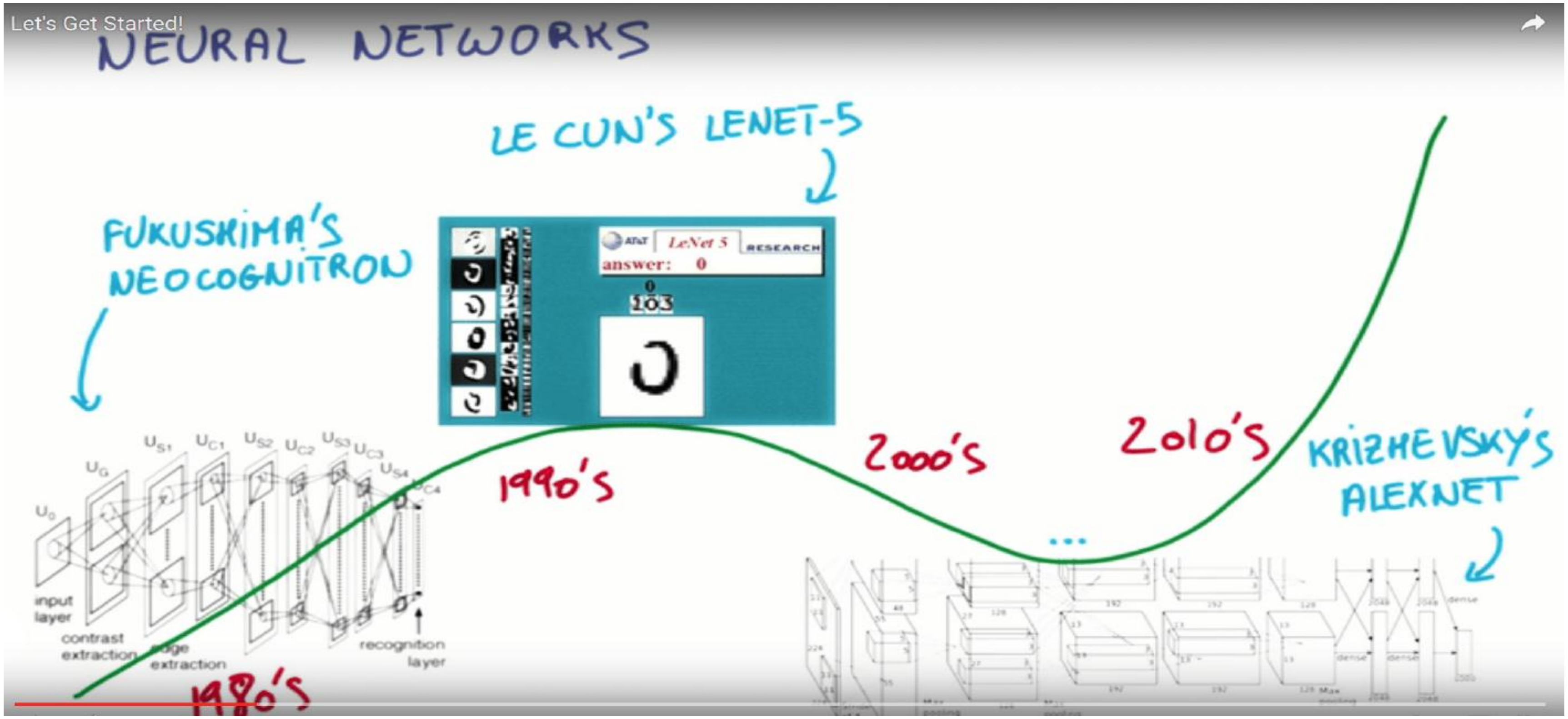
MNIST example

Scientific application:

Higgs CP measurement at LHC

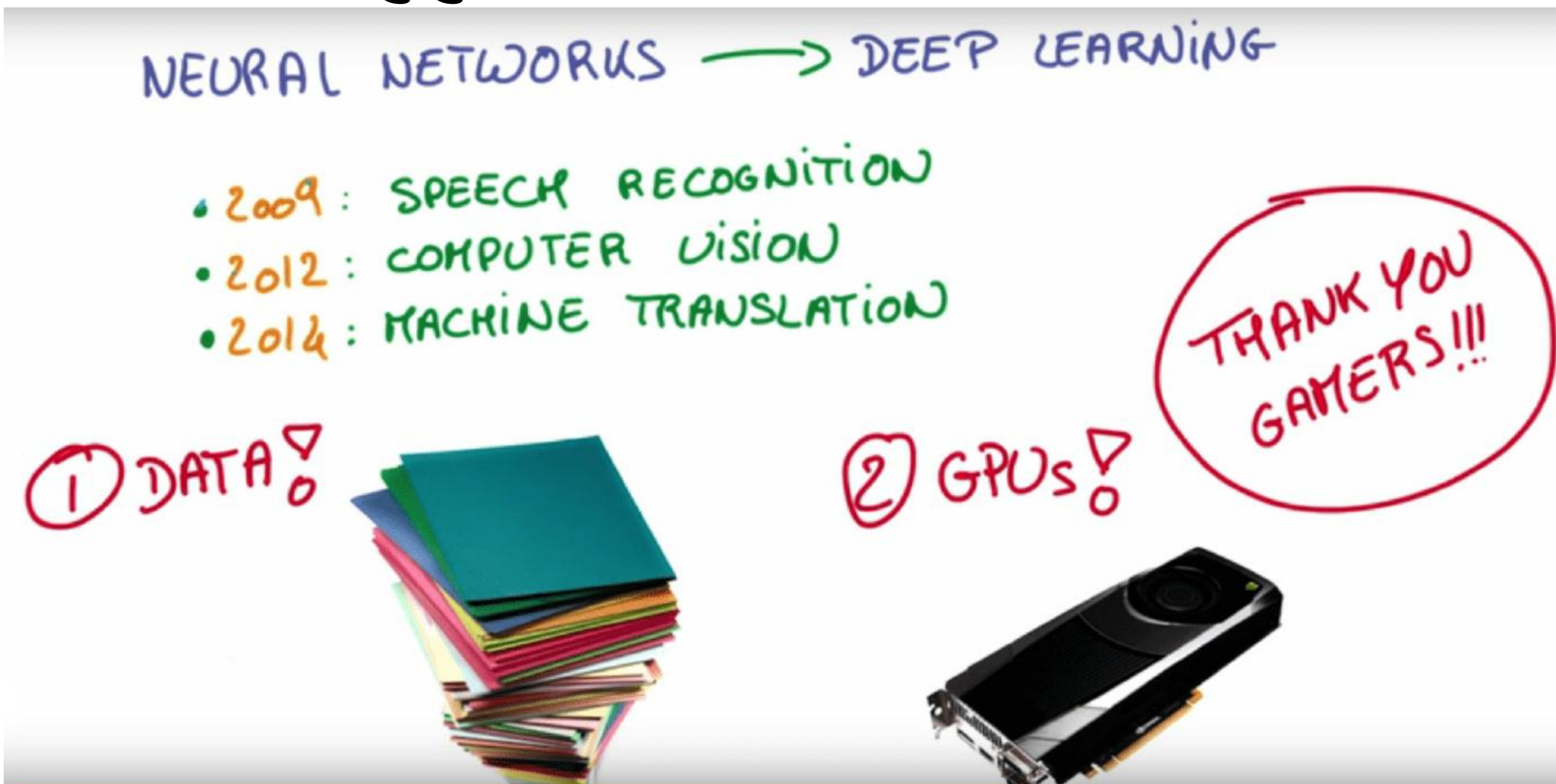
Neural network

Since 2010 new era in Machine Learning: rapidly increasing areas of applications



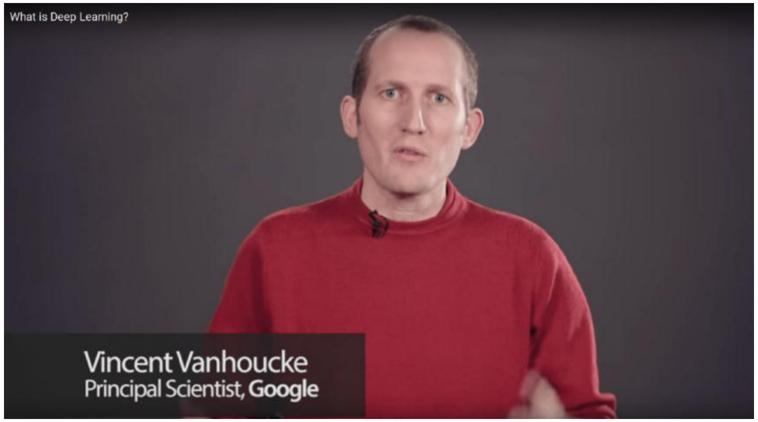
Neural network

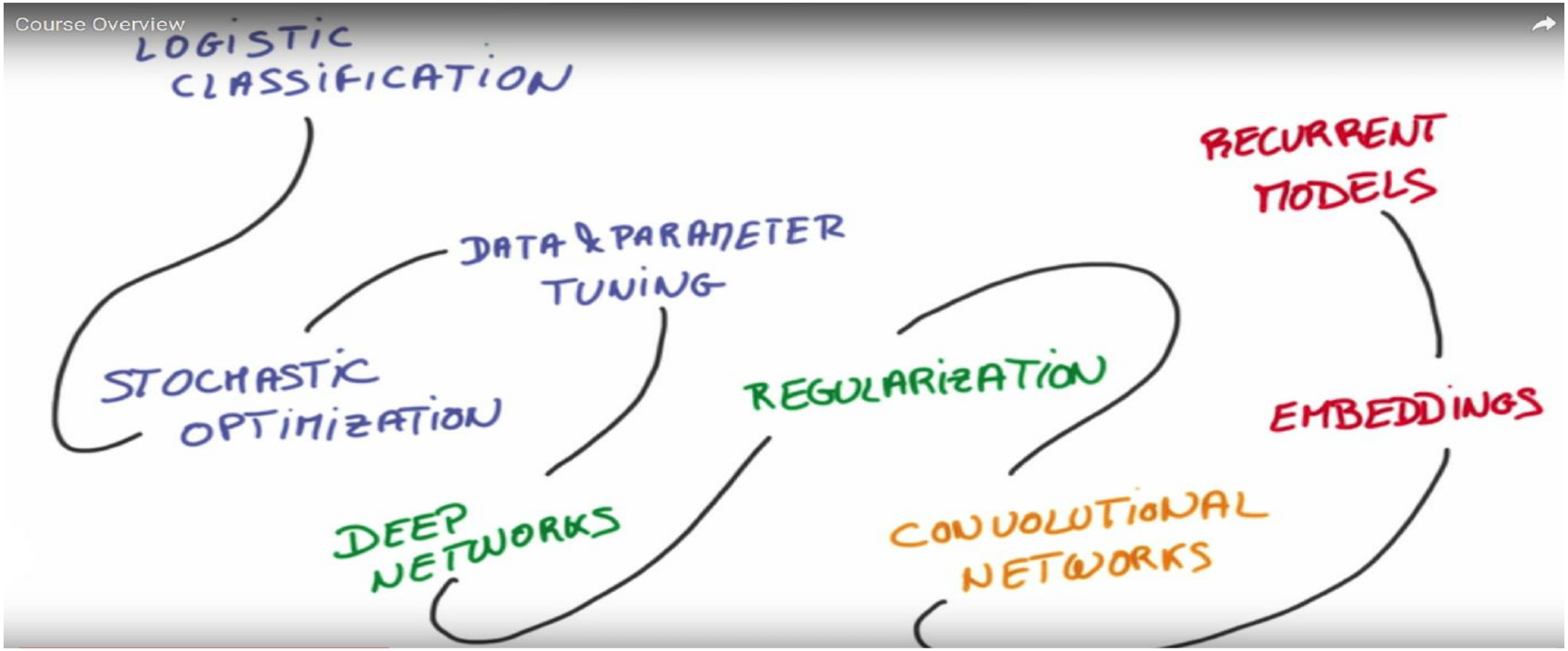
Since 2010 new era: rapidly increasing areas of applications



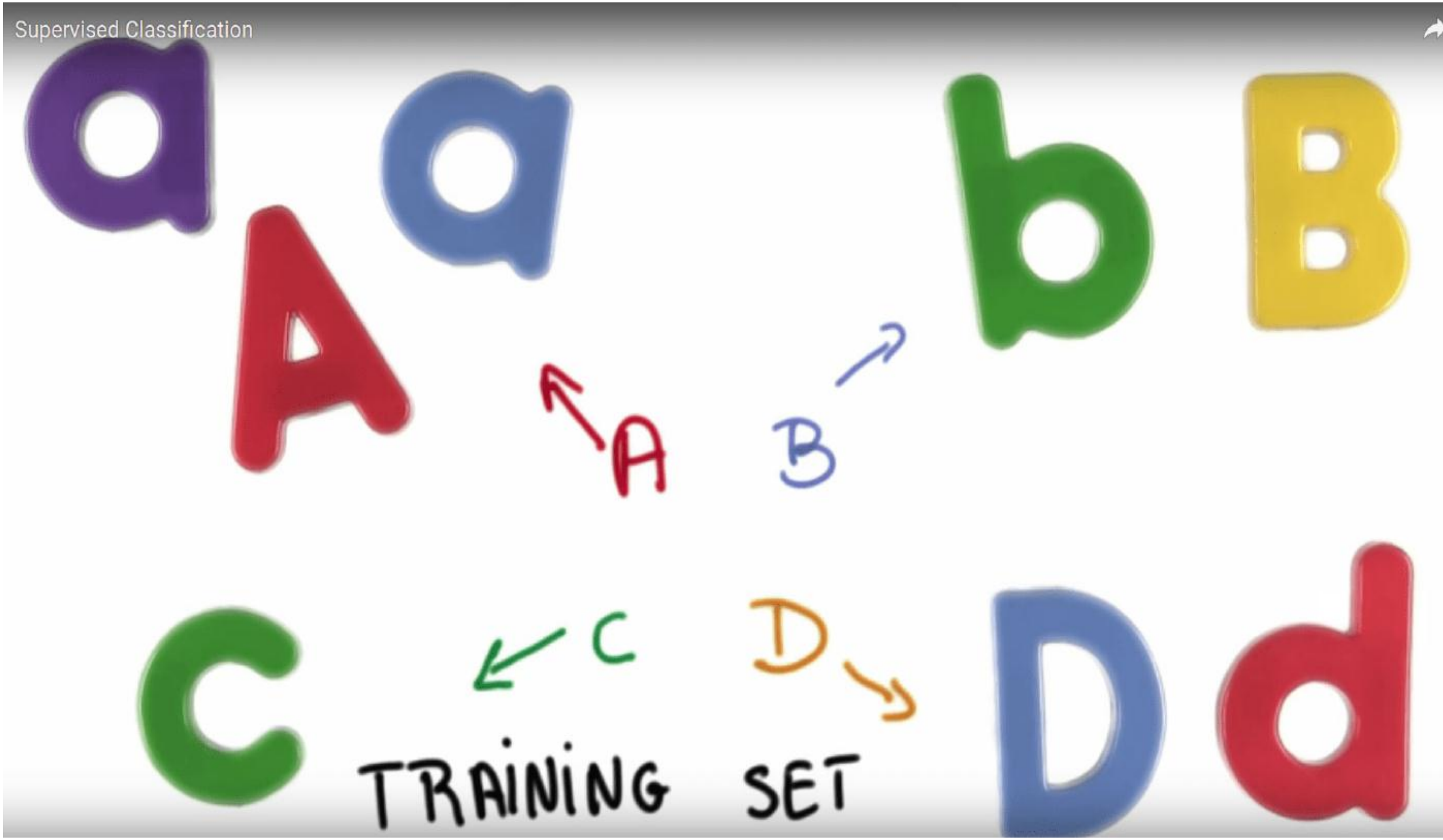
Deep-Learning tutorial @ udacity

https://www.udacity.com/course/deep-learning--ud730

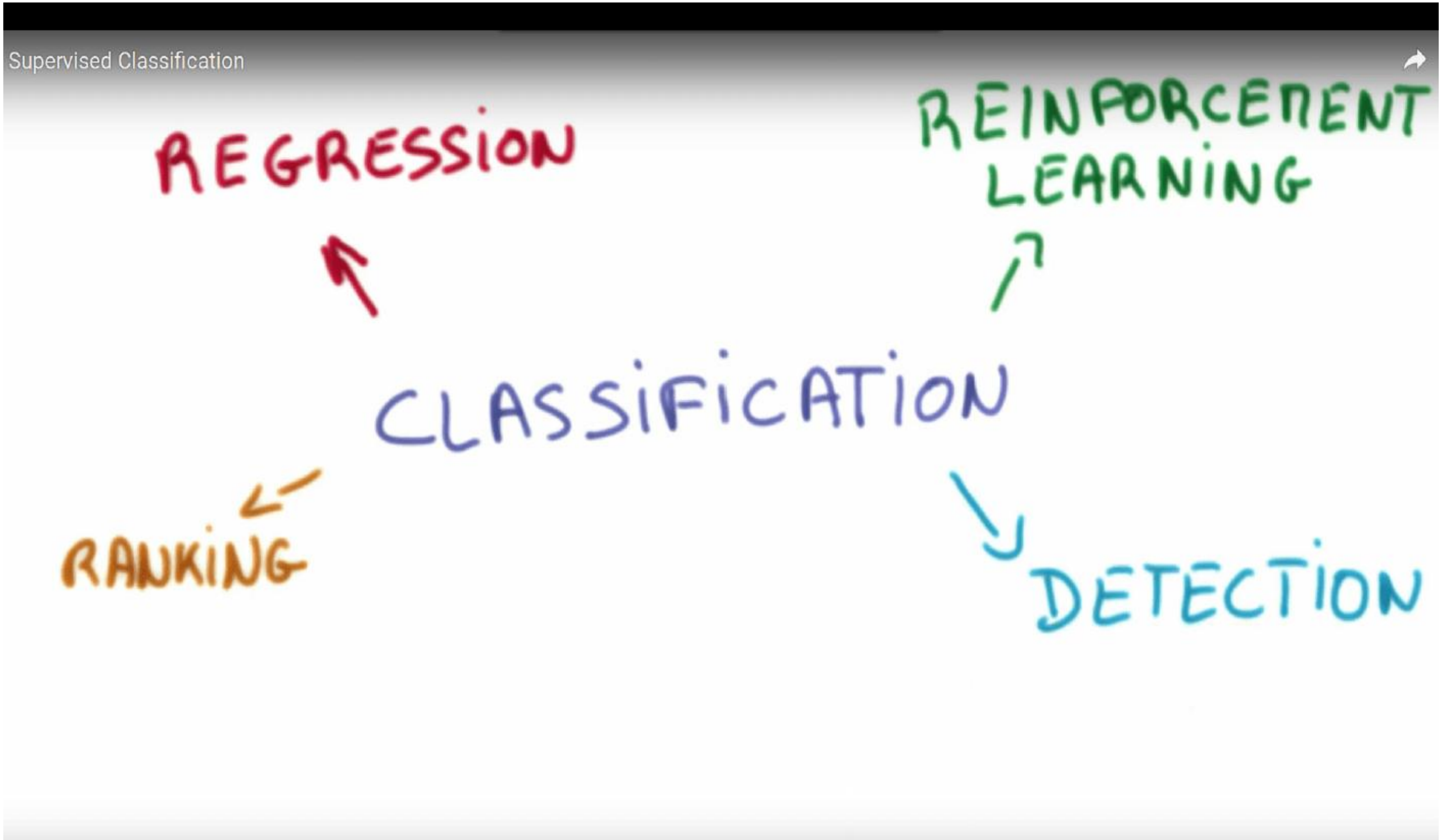




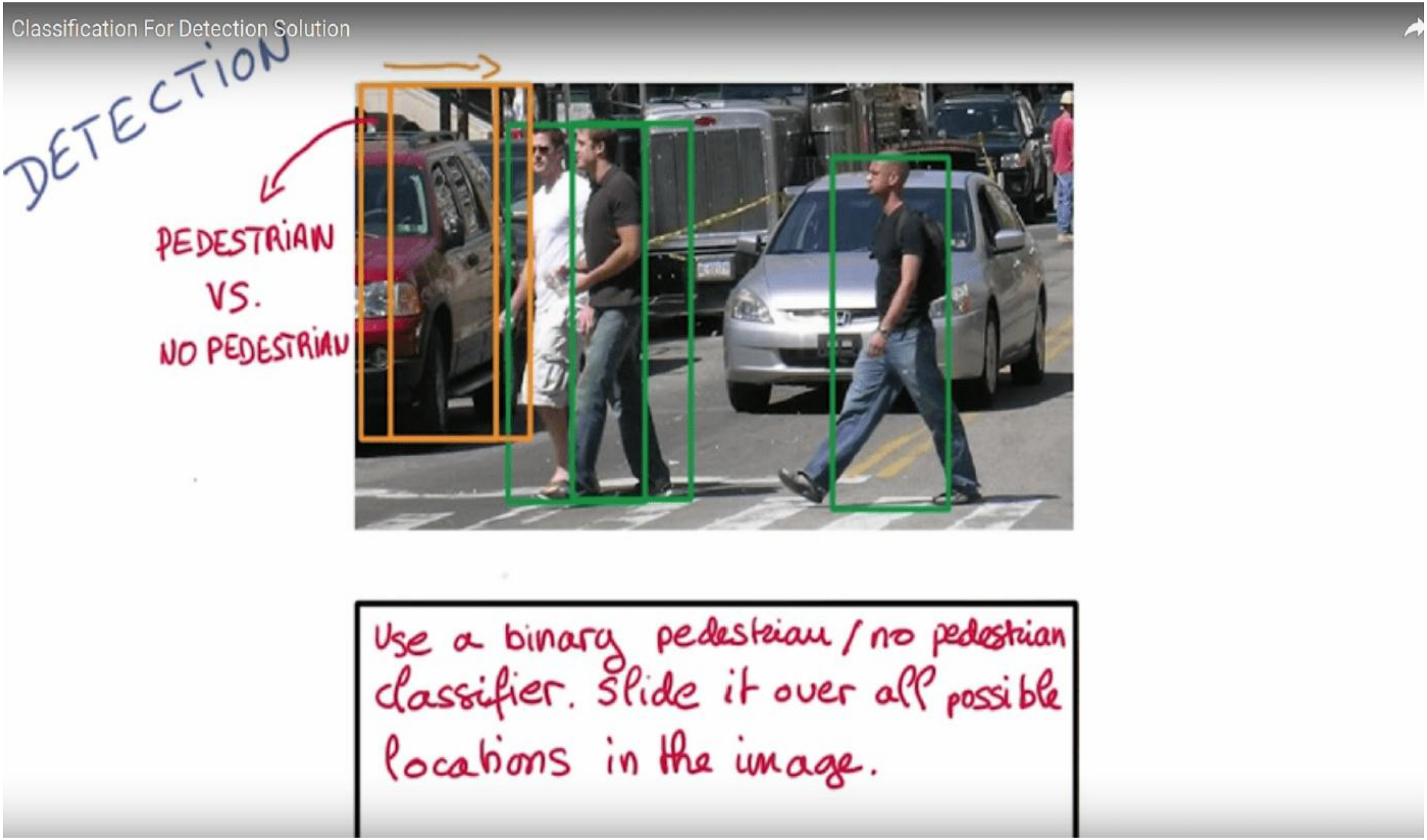
Supervised Classifications



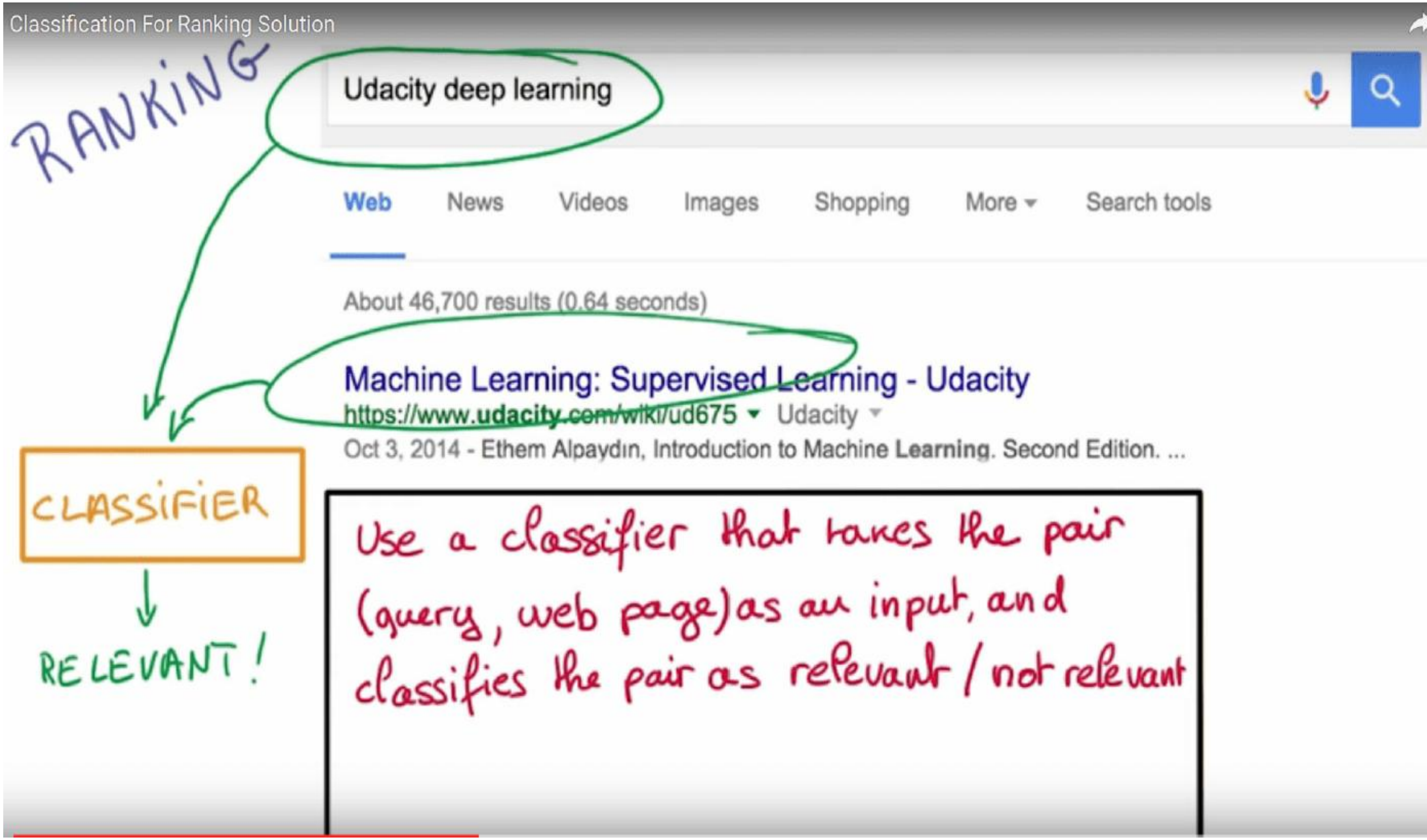
Supervised Classifications



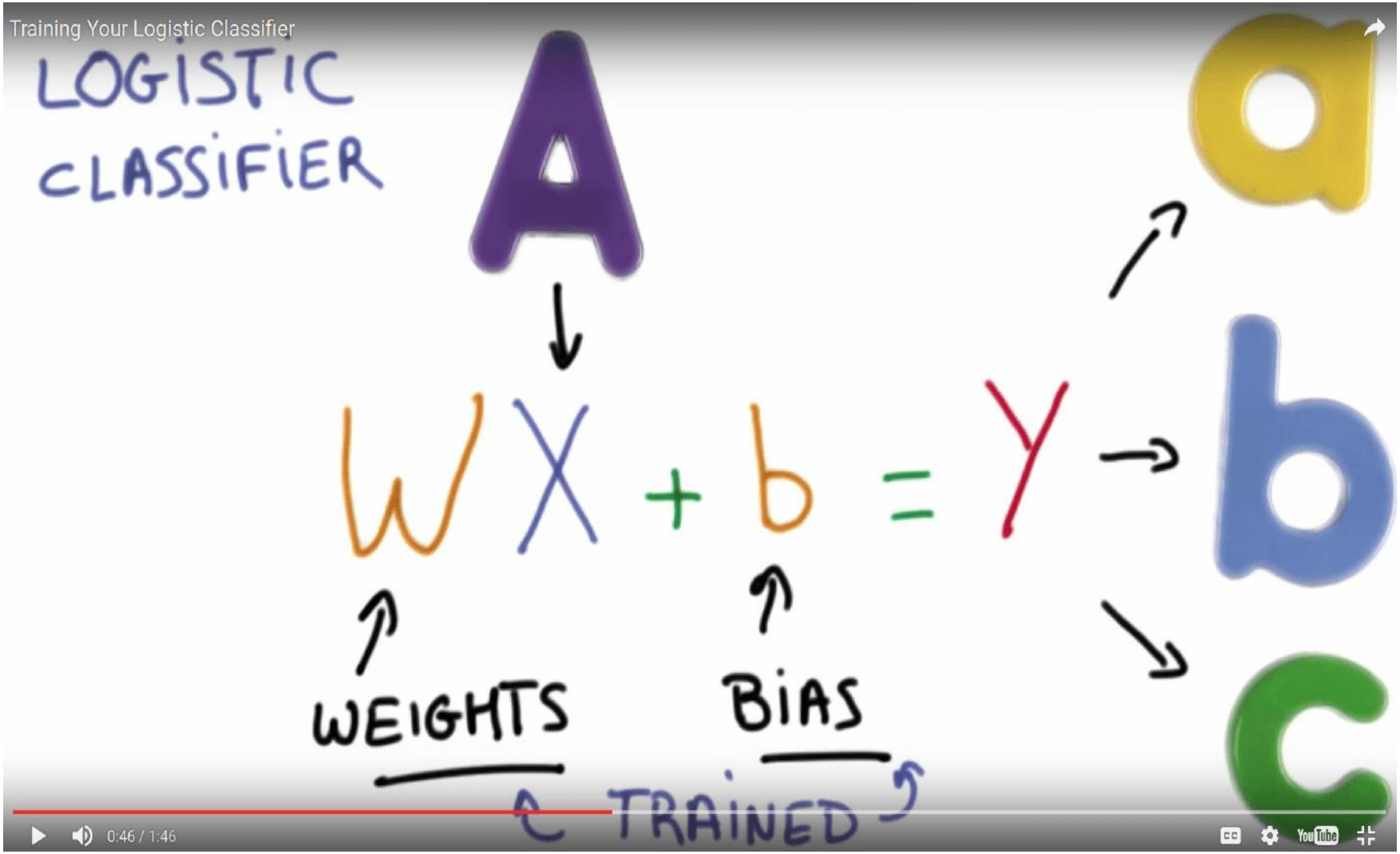
Classifications for Detection



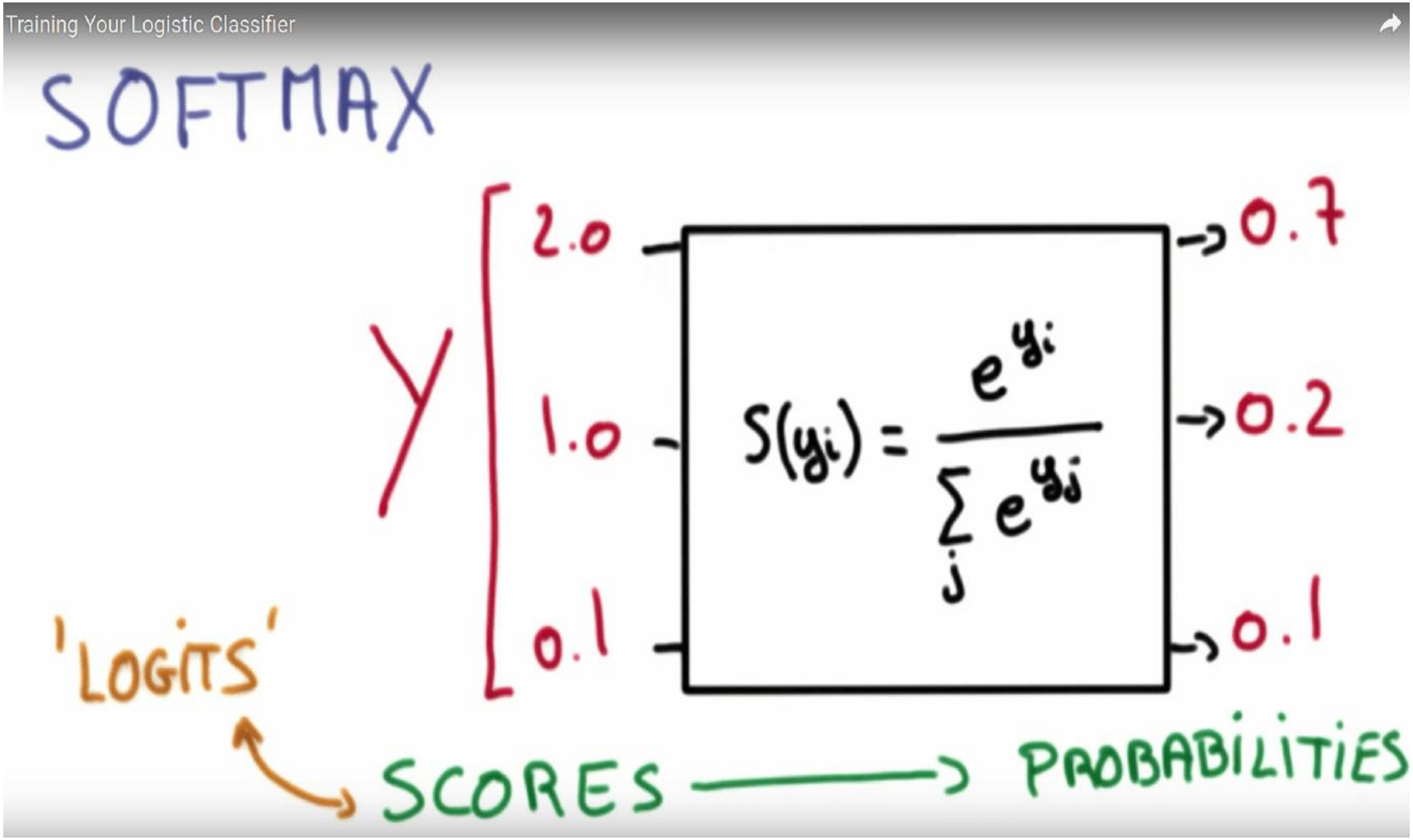
Classifications for Ranking



Logistic classifier: Linear model

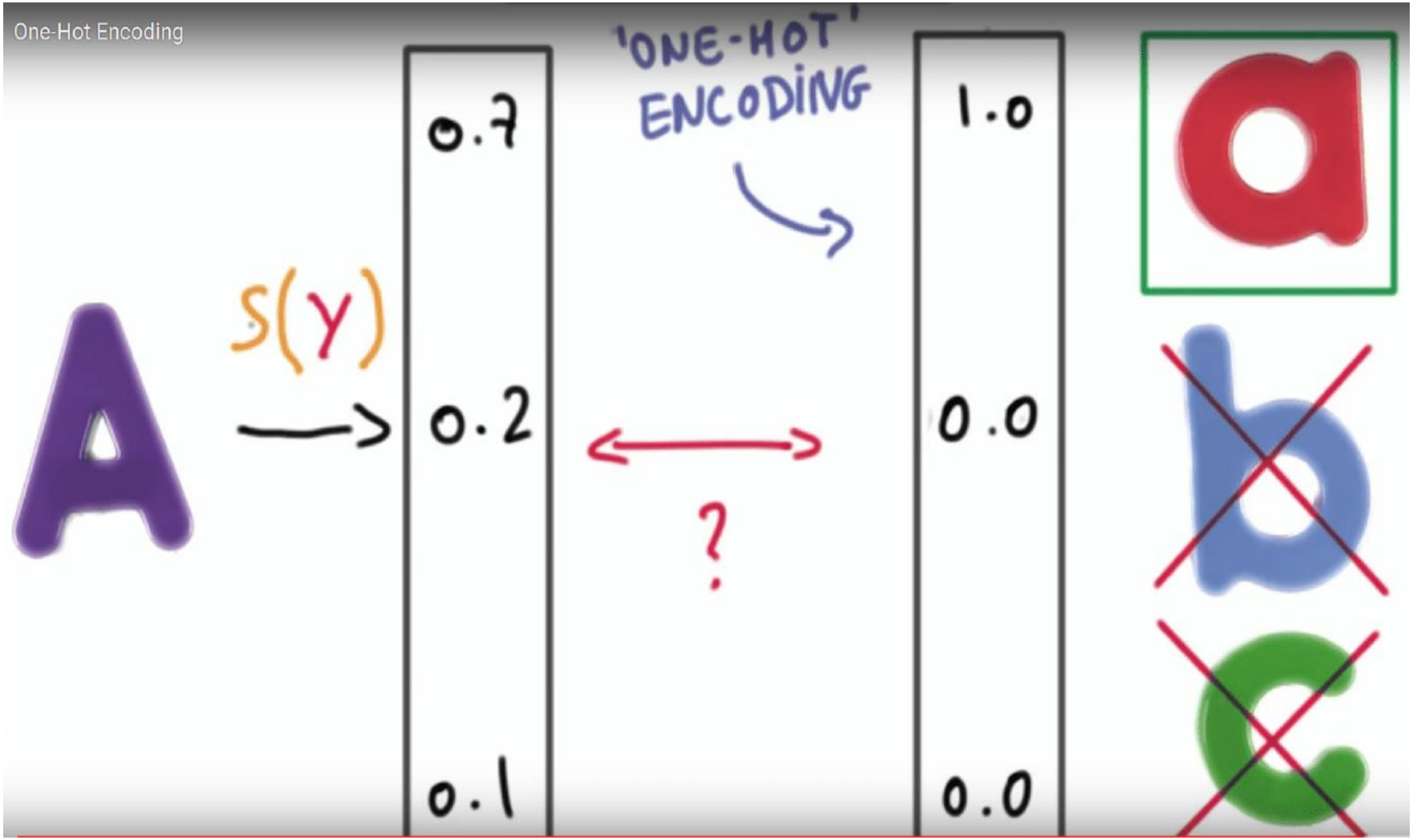


Softmax

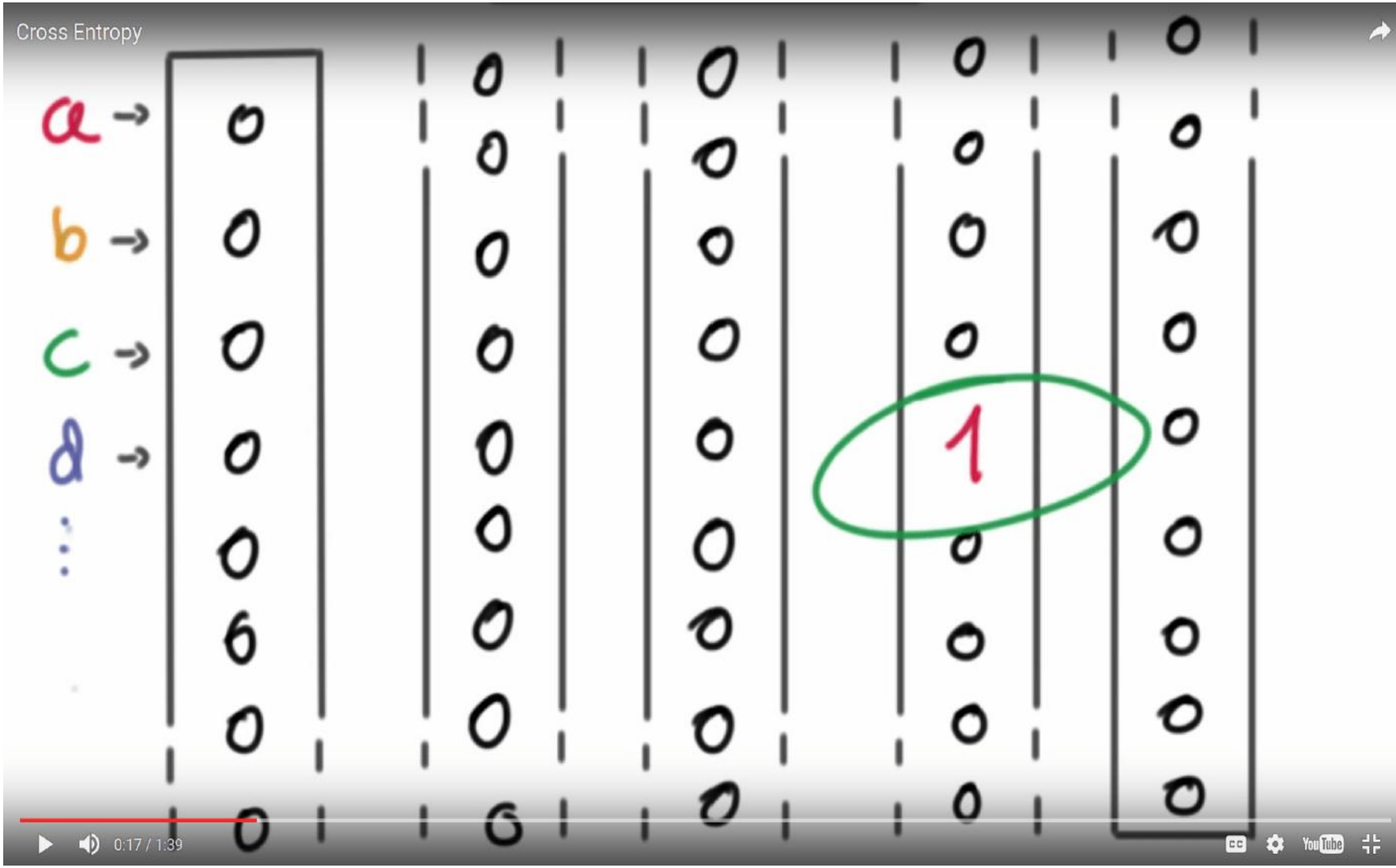


Training Your Logistic Classifier
SOFTMAX
Y
2.0
1.0
0.1
$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$
0.7
0.2
0.1
'LOGITS'
SCORES
PROBABILITIES

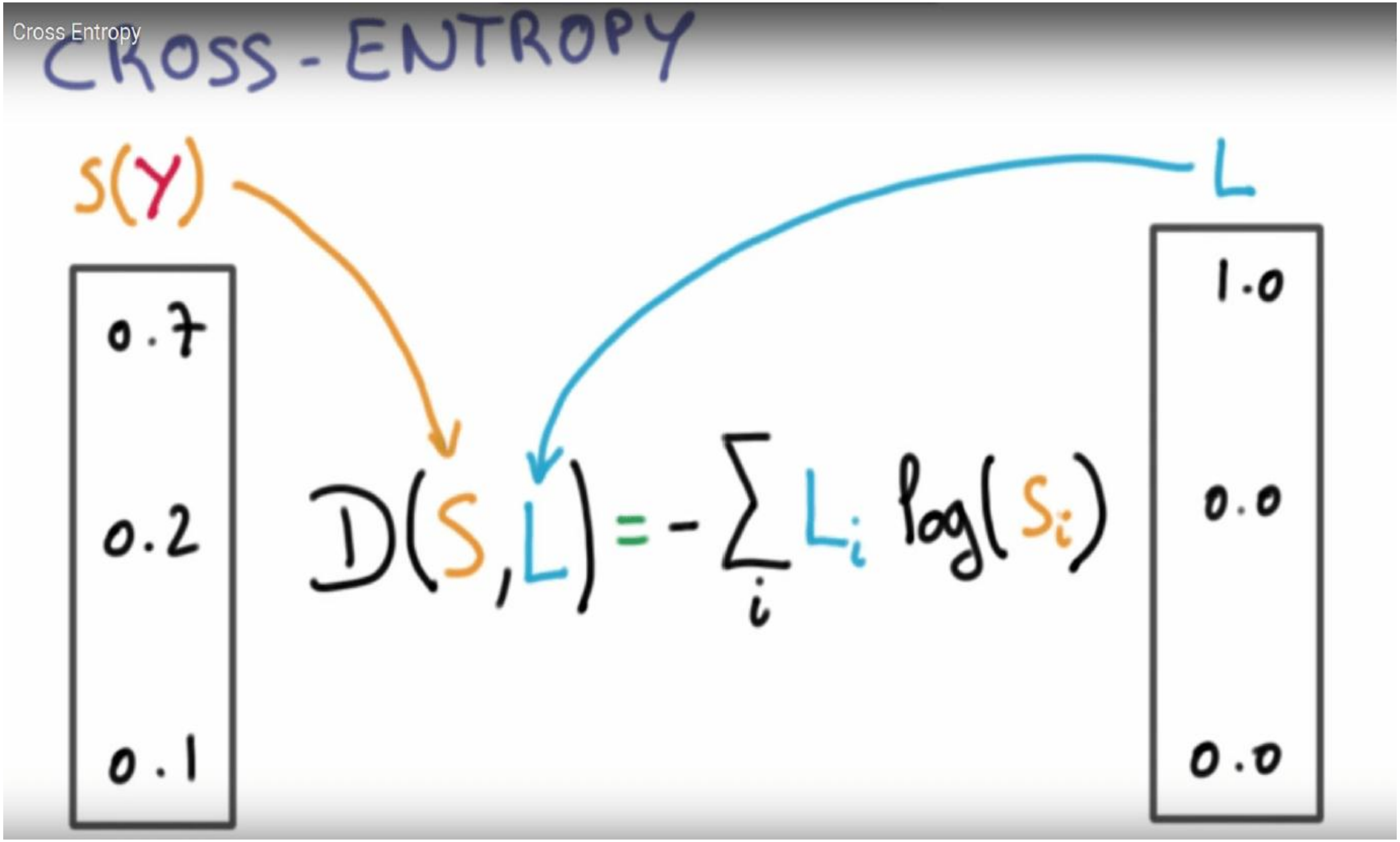
„One hot” encoding



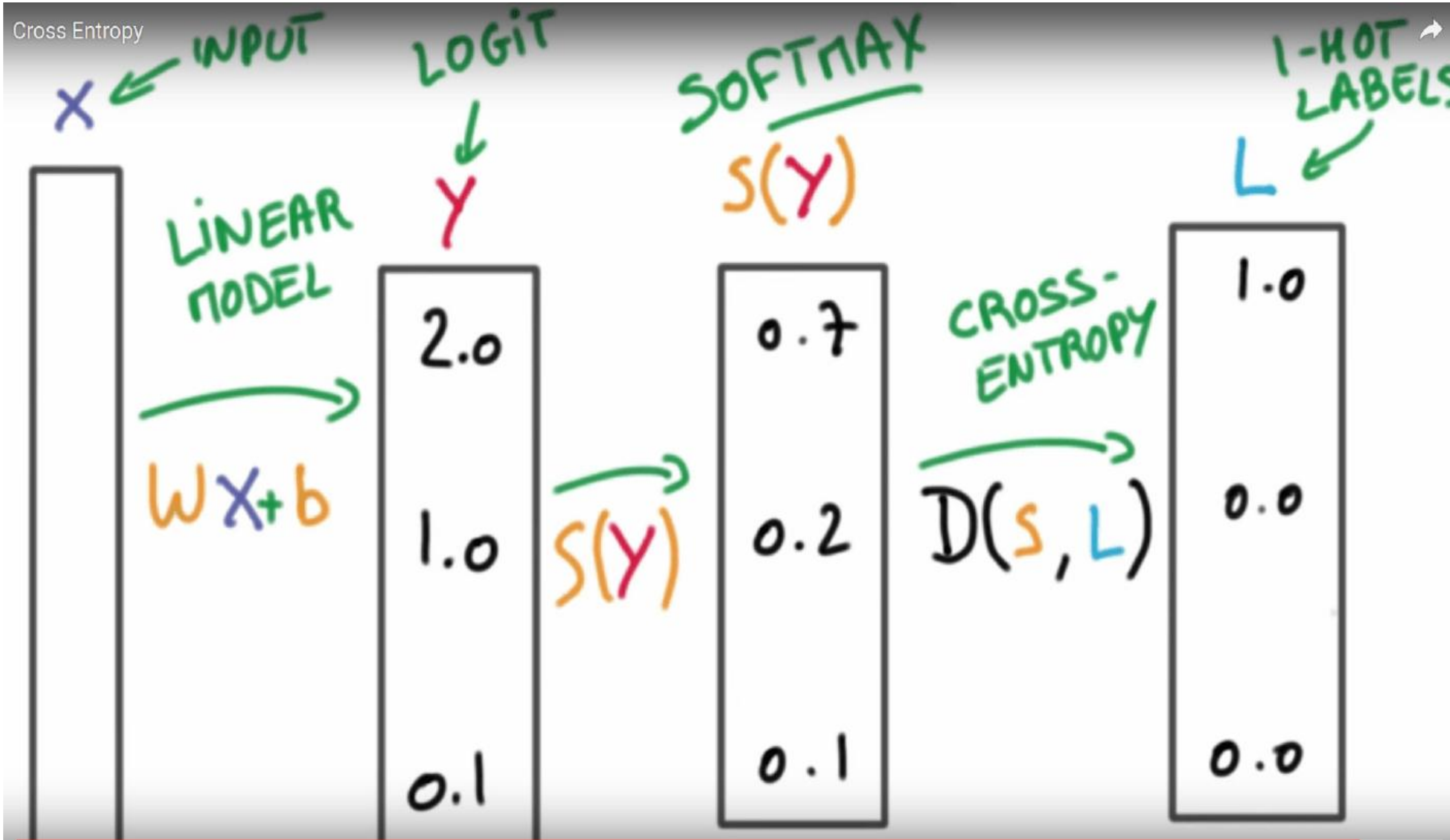
„One hot" encoding



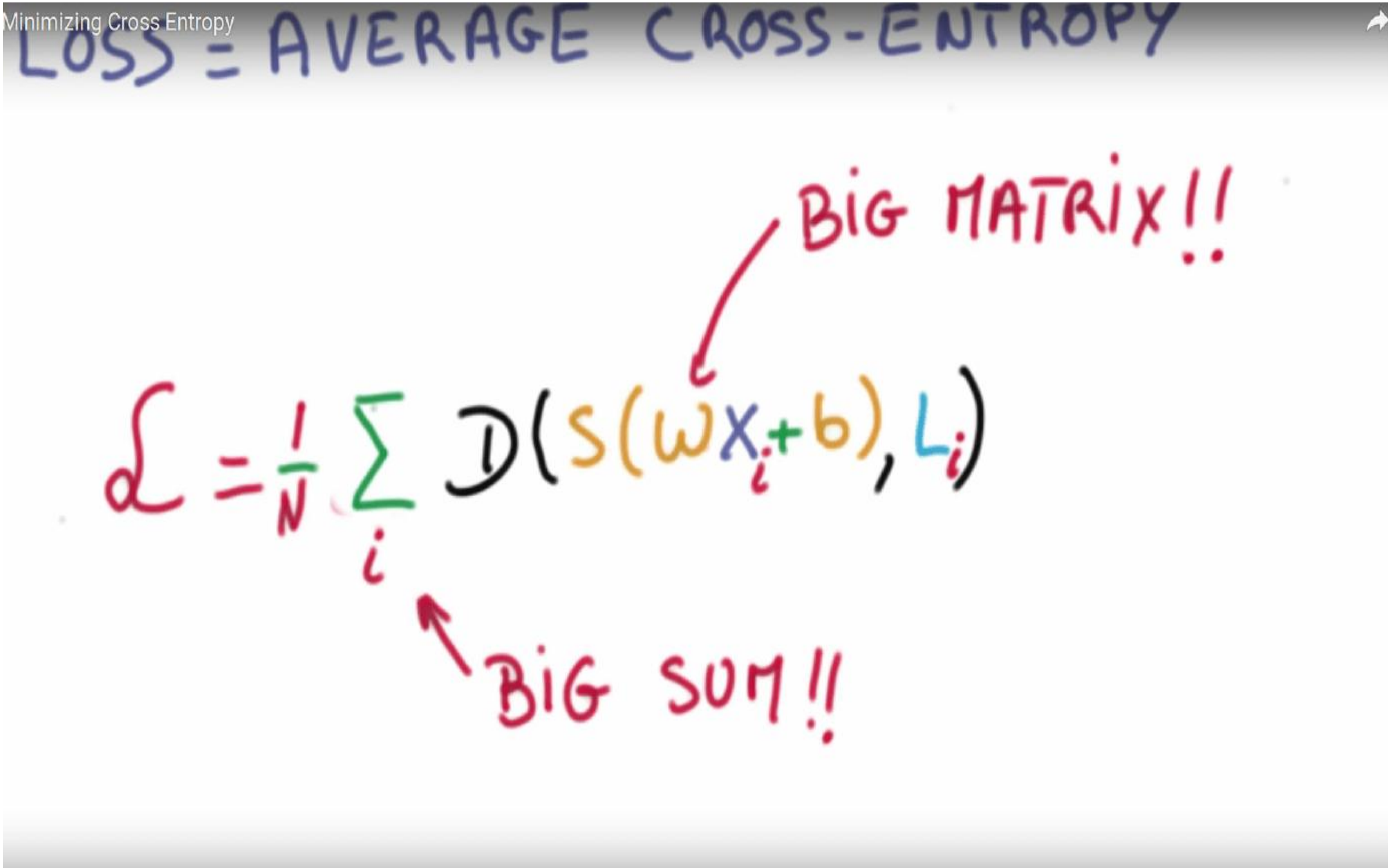
Optimisation: Cross-Entropy



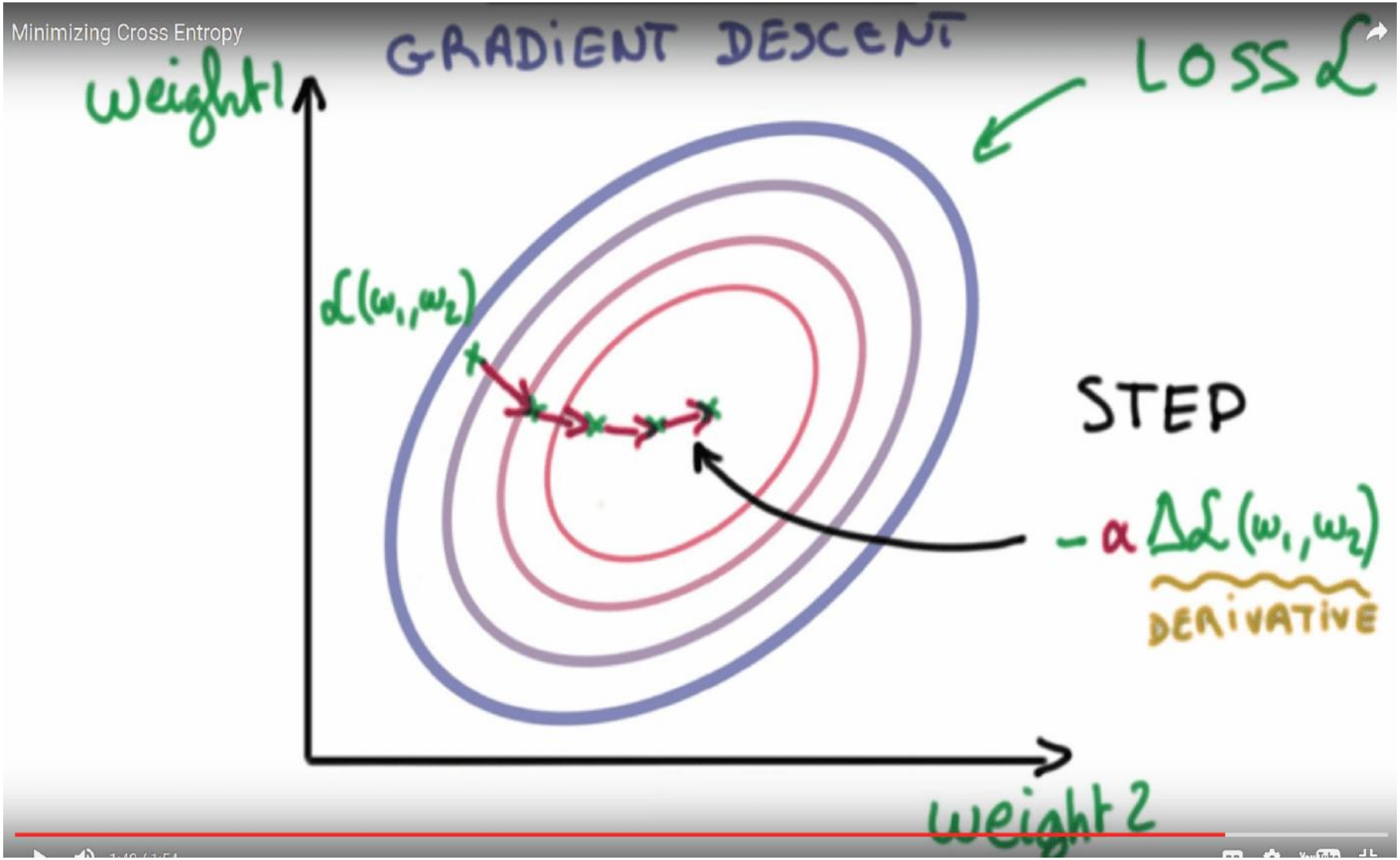
Multinomial logistic classification



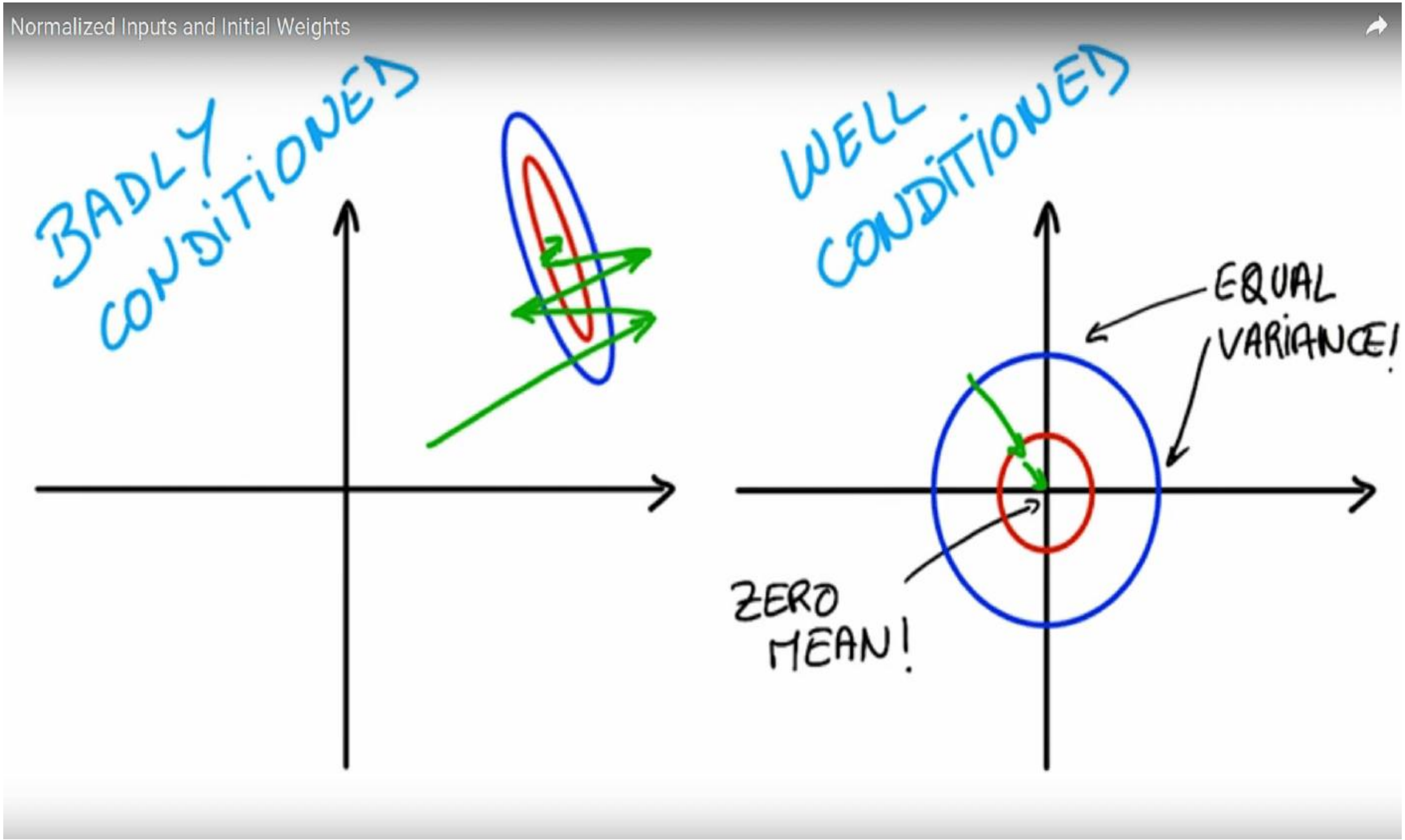
Optimisation of average loss



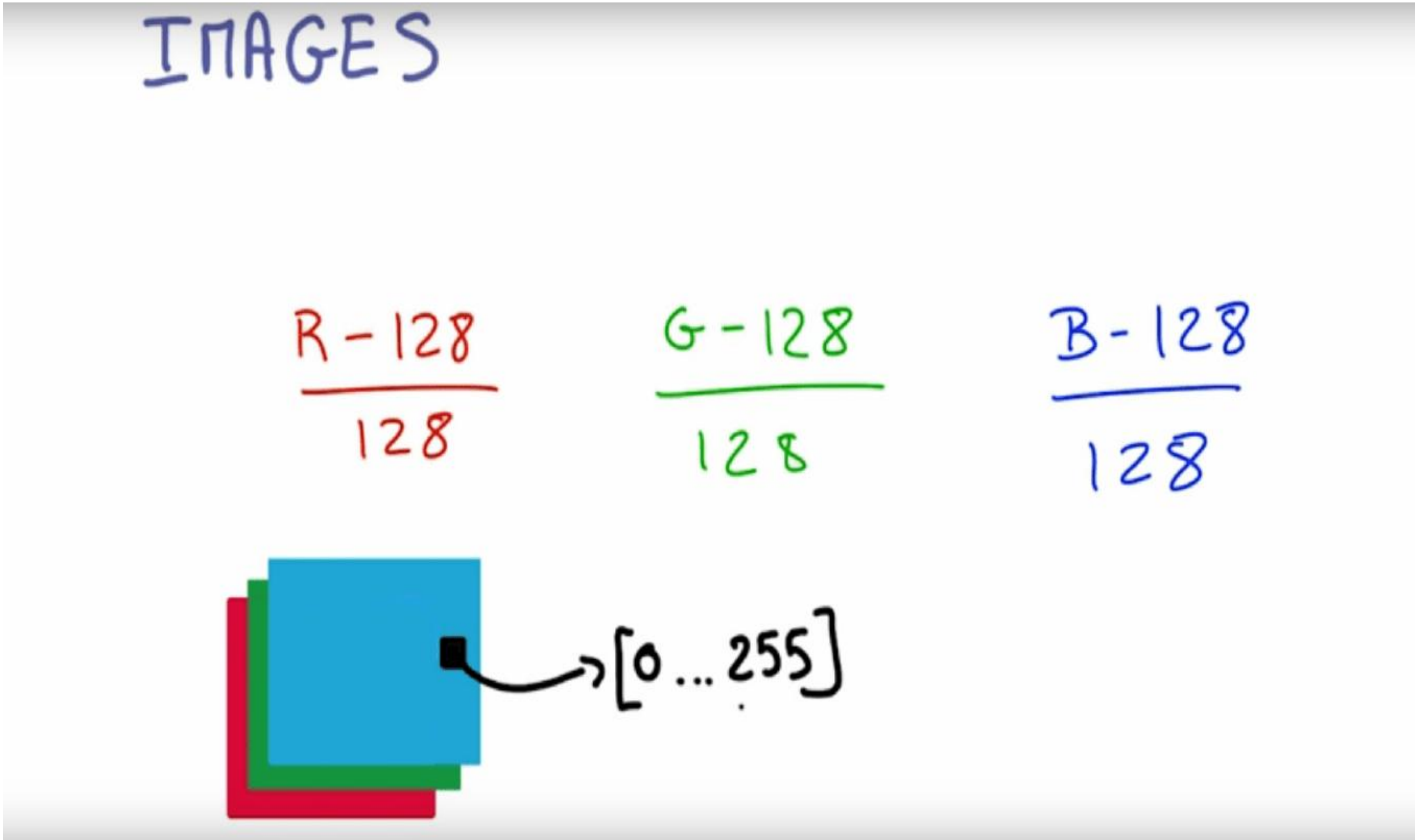
Gradient decent



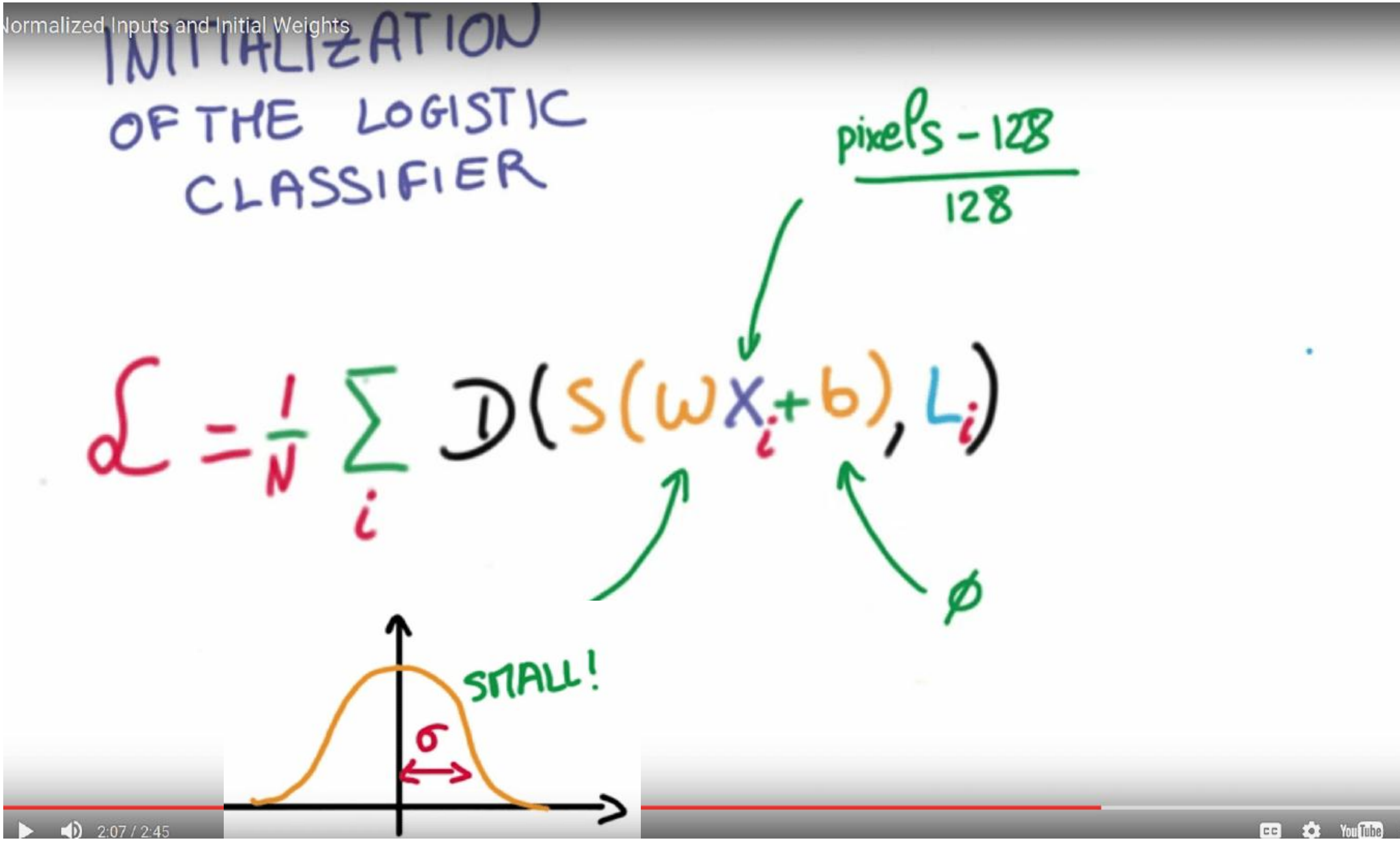
Normalised input and output



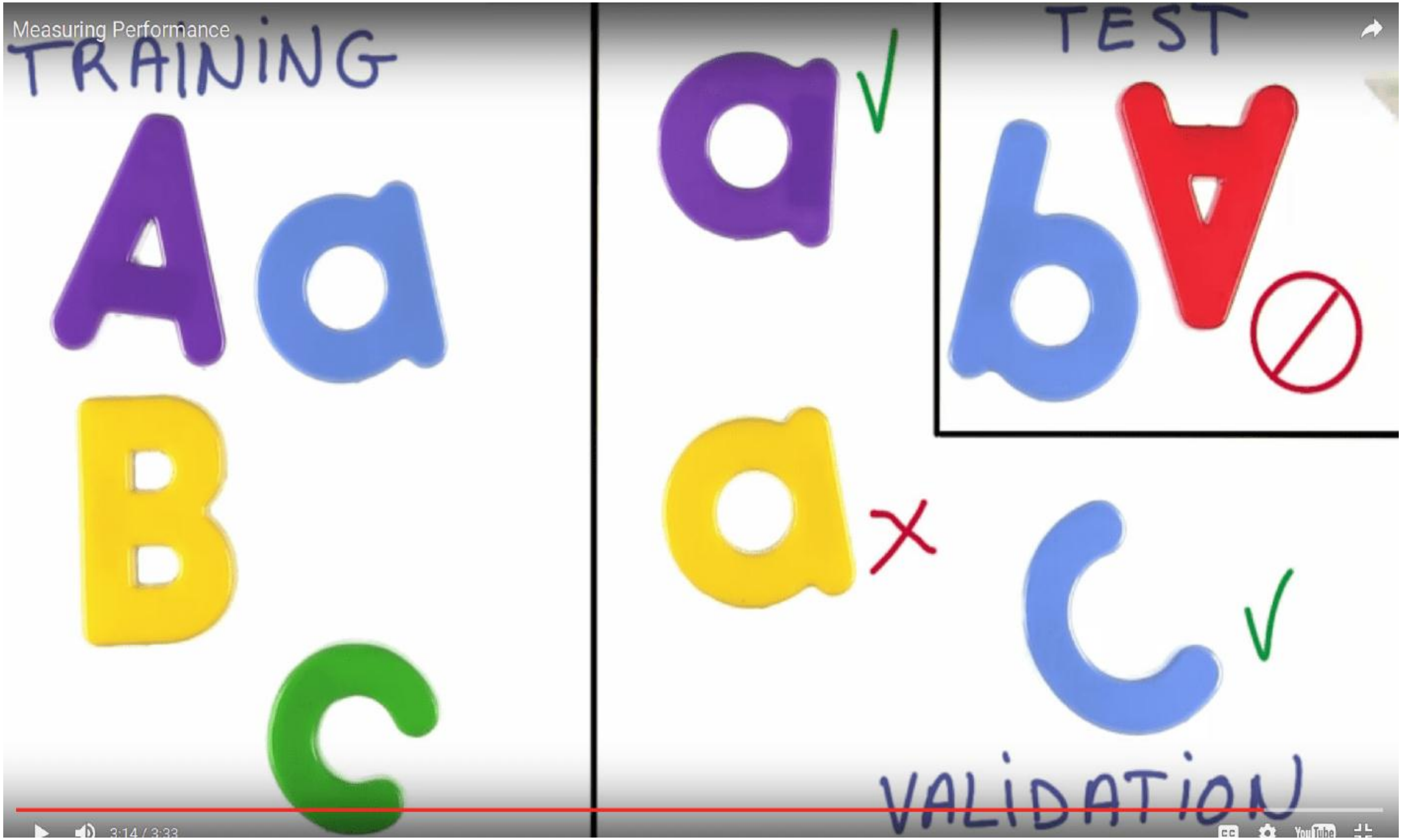
Normalised input and output



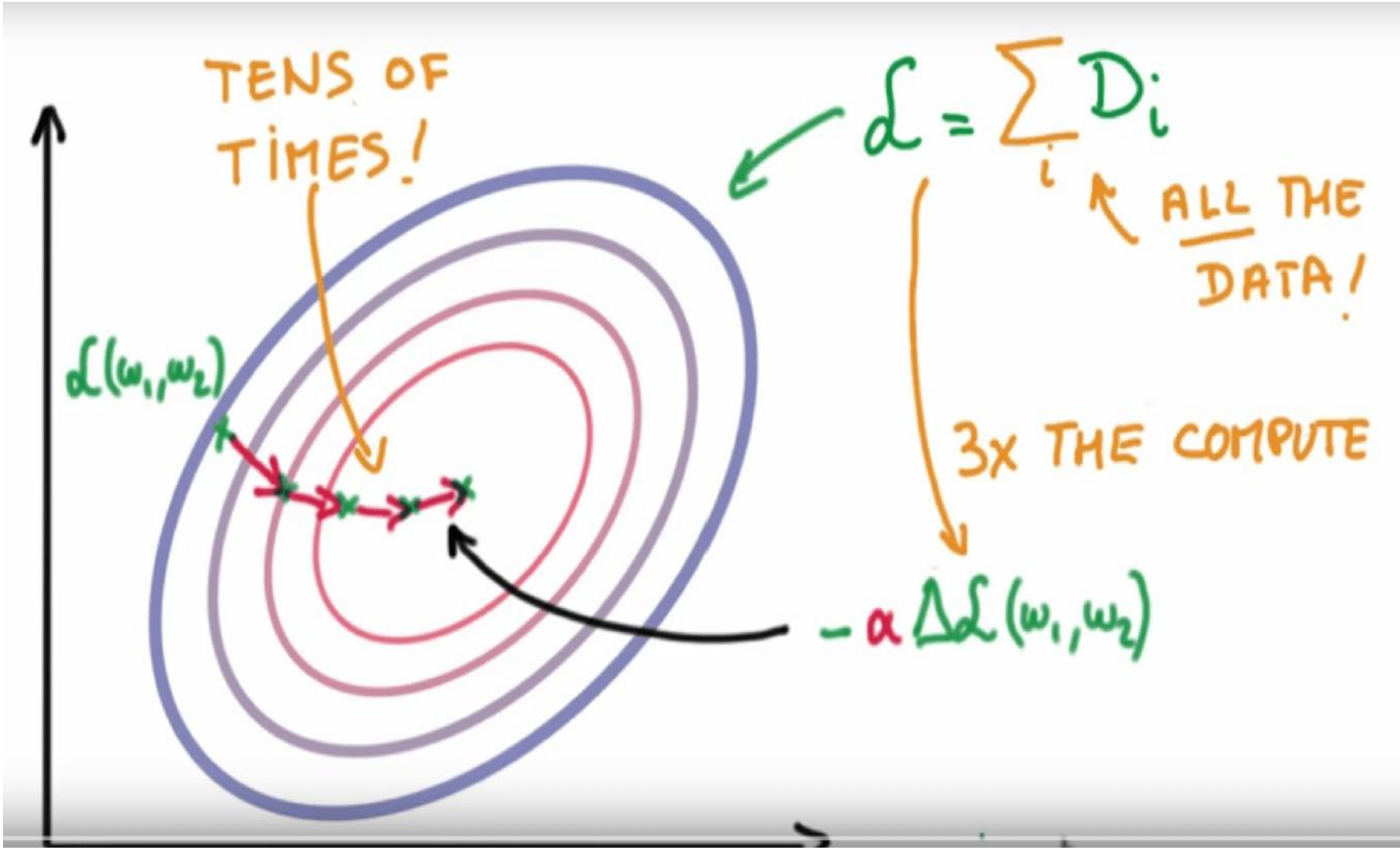
Initialisation



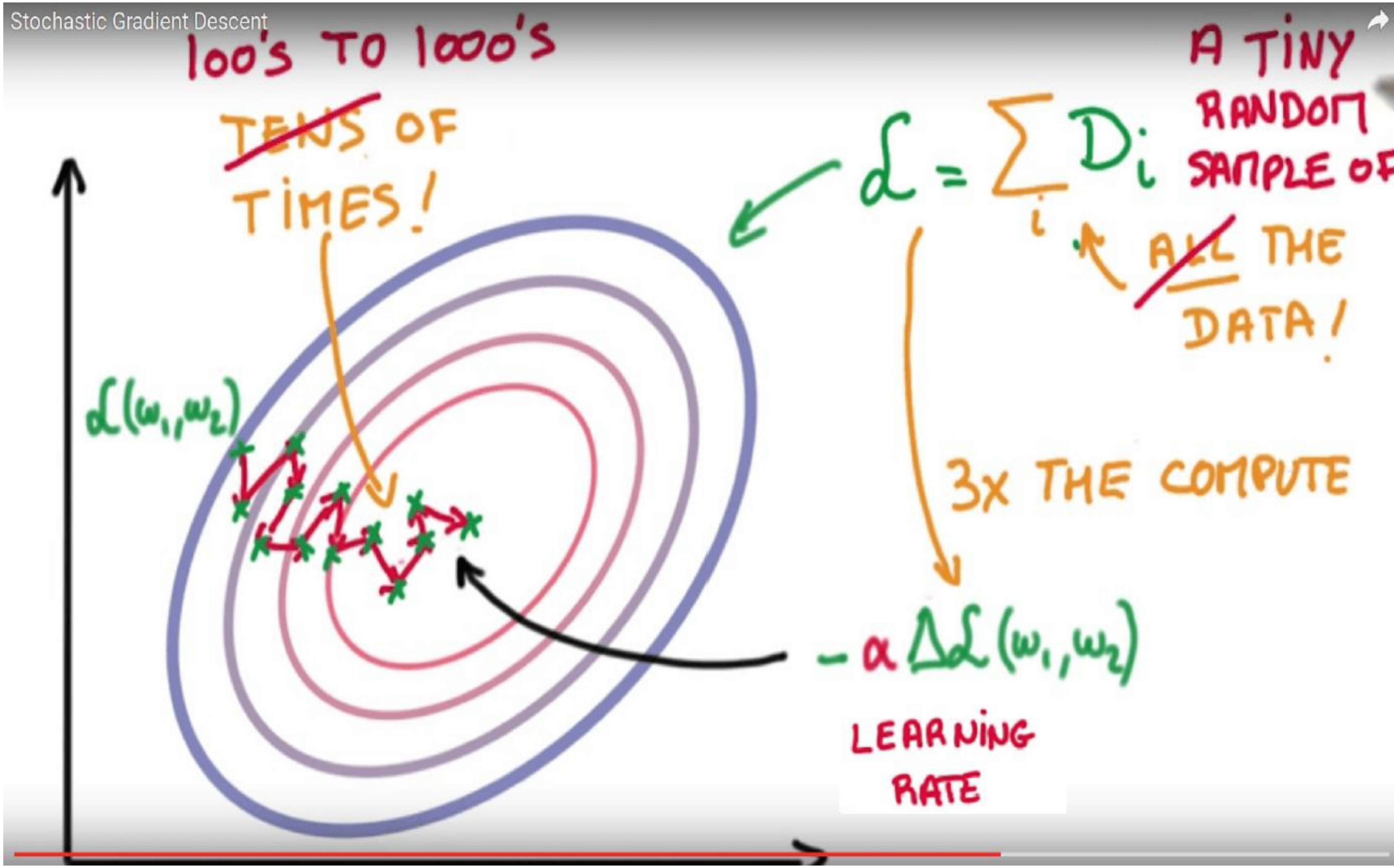
Training, validation, testing



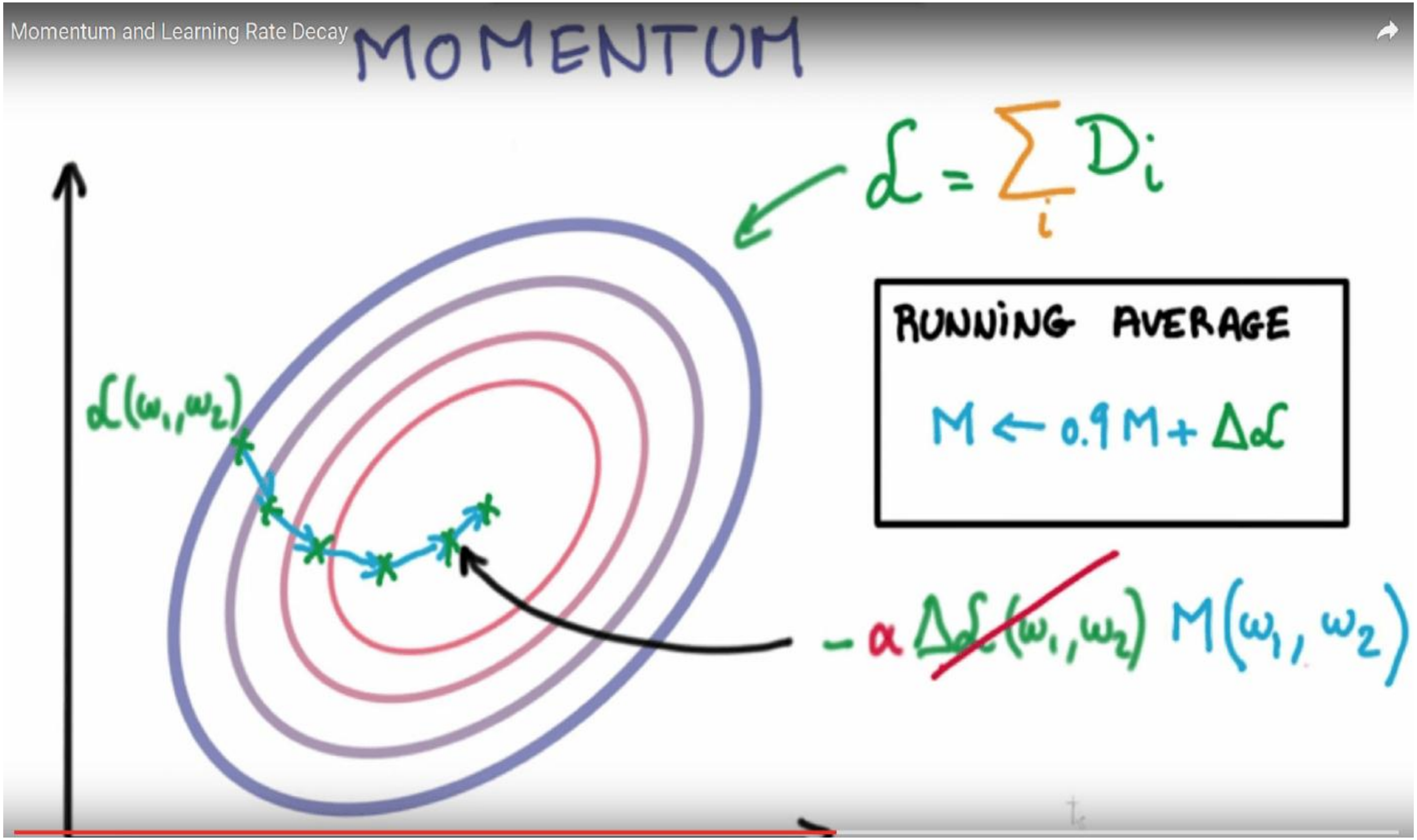
Gradient Descent



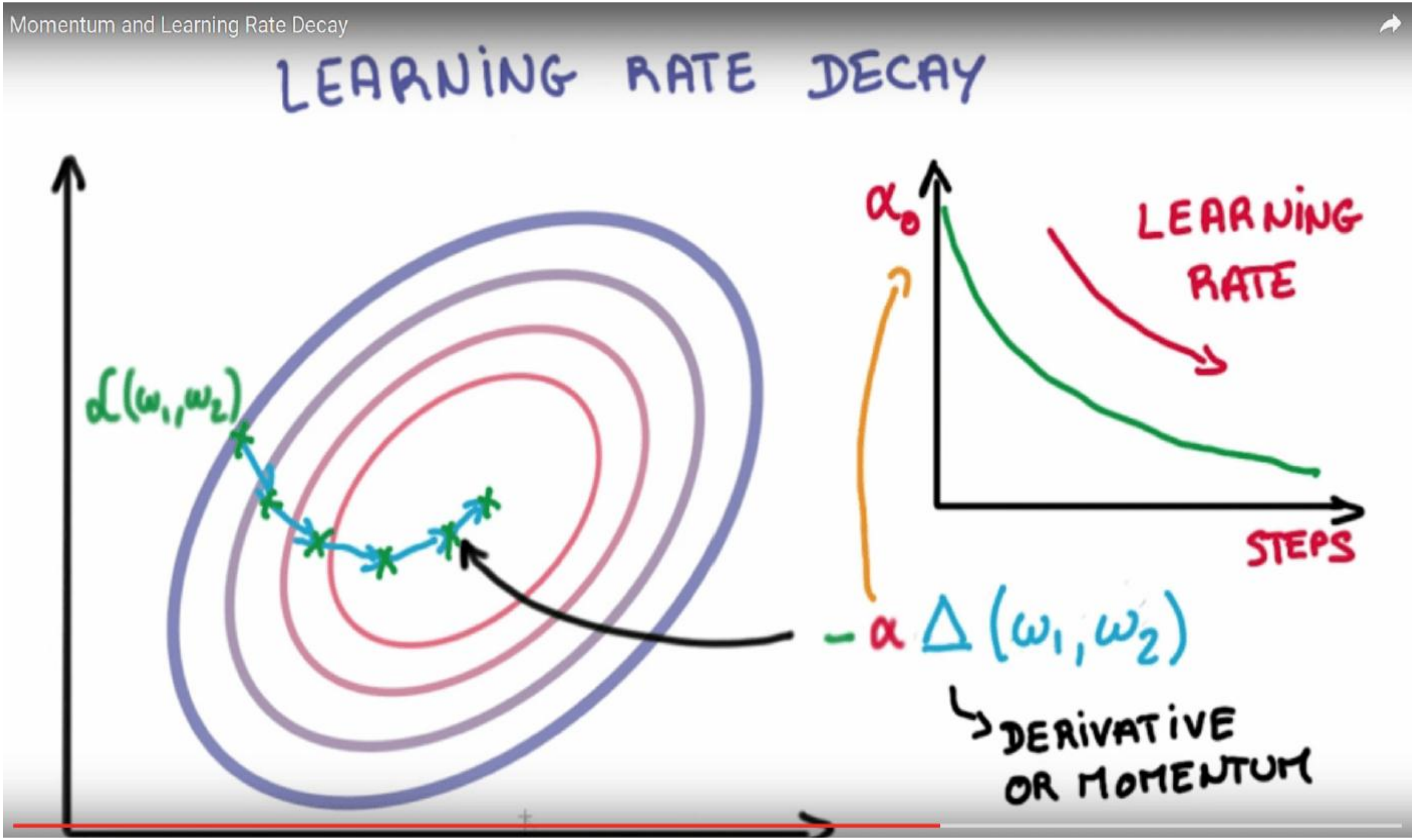
Stochastic Gradient Descent



SDG: optimising with momentum



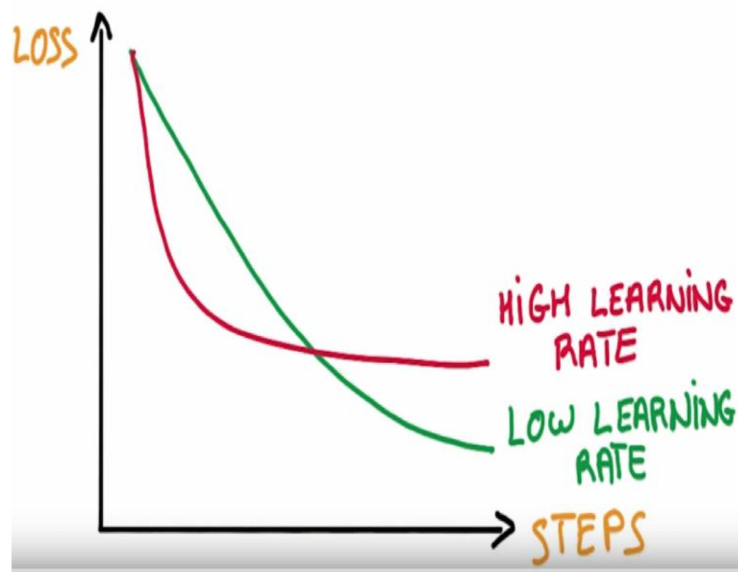
SDG: learning rate



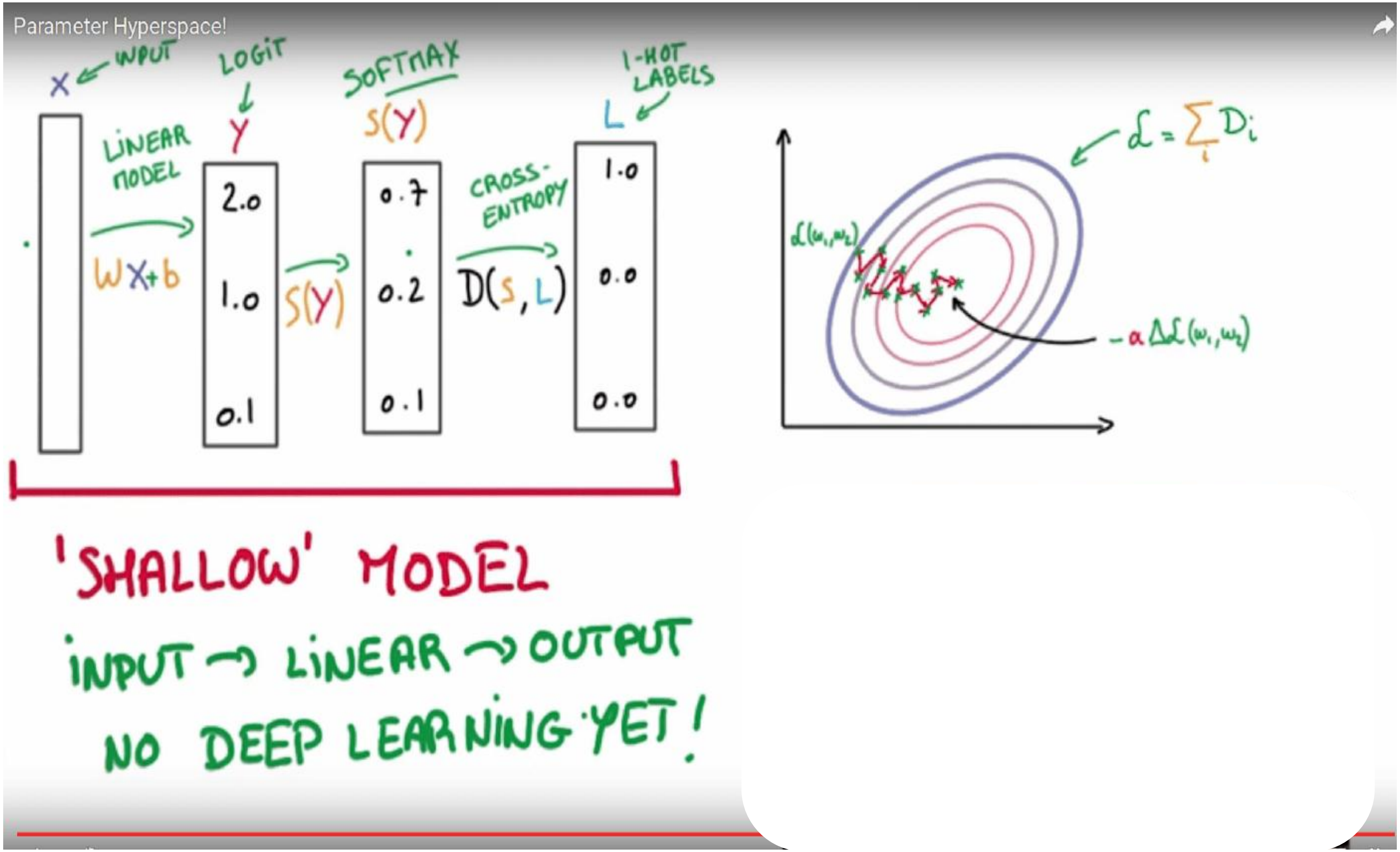
SDG: „black magic”

MANY HYPER-PARAMETERS

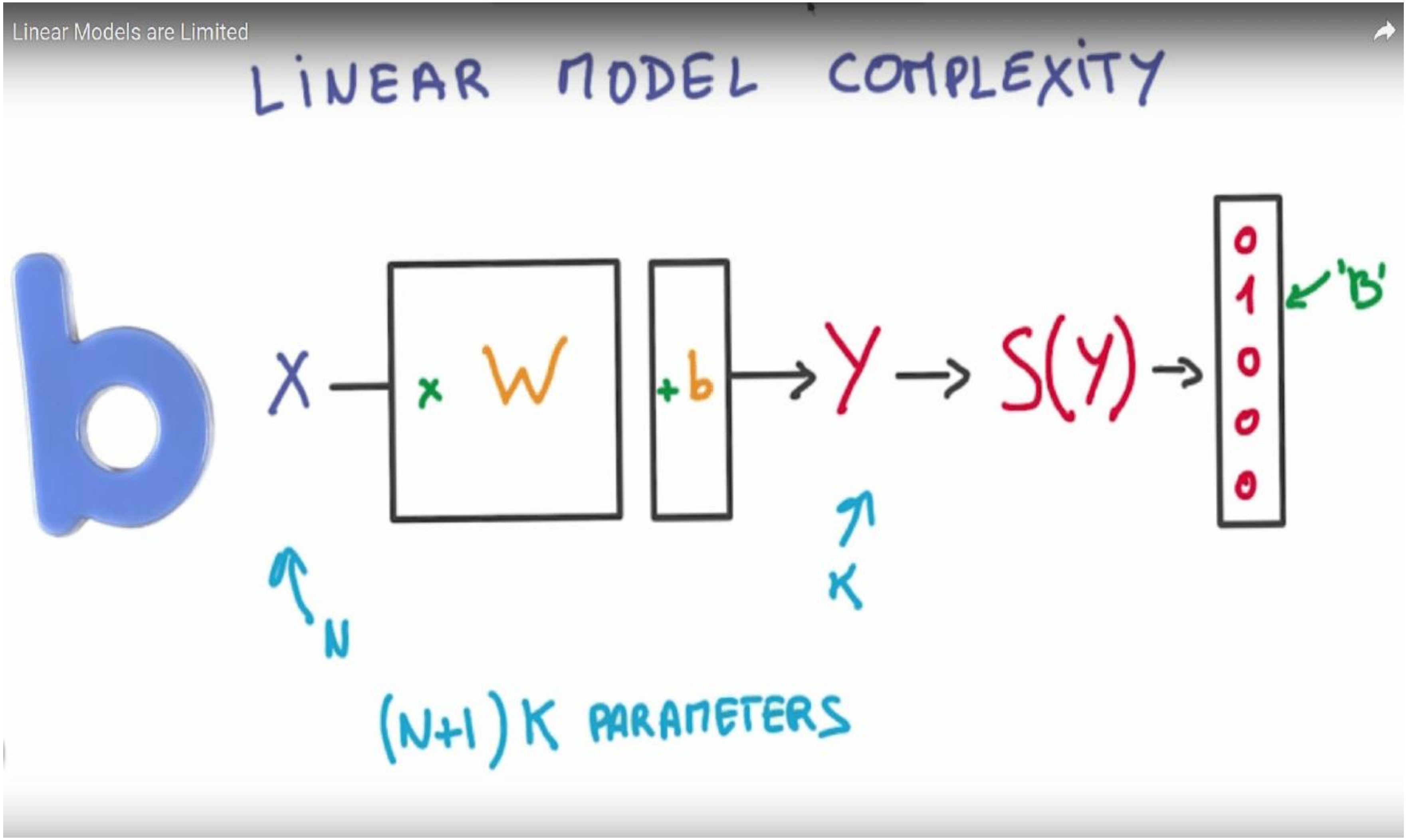
- INITIAL LEARNING RATE
- LEARNING RATE DECAY
- MOMENTUM
- BATCH SIZE
- WEIGHT INITIALIZATION



Input – linear - output



Linear models are linear



Linear models are stable

$$Y = WX \longrightarrow \underbrace{\Delta Y}_{\text{SMALL}} \sim \underbrace{|W|}_{\text{BOUNDED}}\,\underbrace{\Delta X}_{\text{SMALL}}$$

$$Y = WX \longrightarrow \frac{\Delta Y}{\Delta X} = W^T \qquad \frac{\Delta Y}{\Delta W} = X^T$$

CONSTANTS

Linear models are here to stay

- **This is still linear**

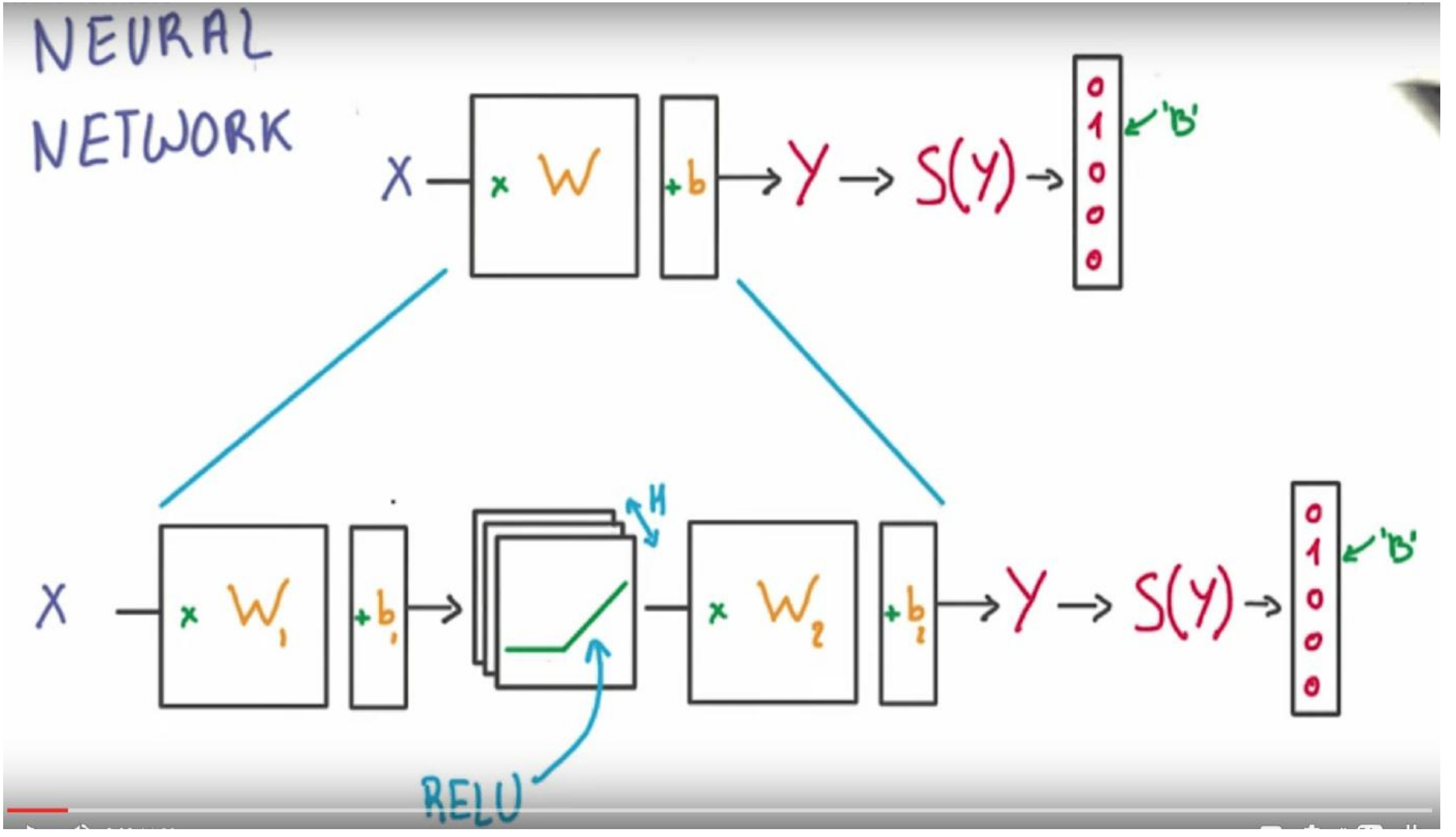
$$Y = W_1 W_2 W_3 X = W X$$

- **Lets introduce non-linearity**

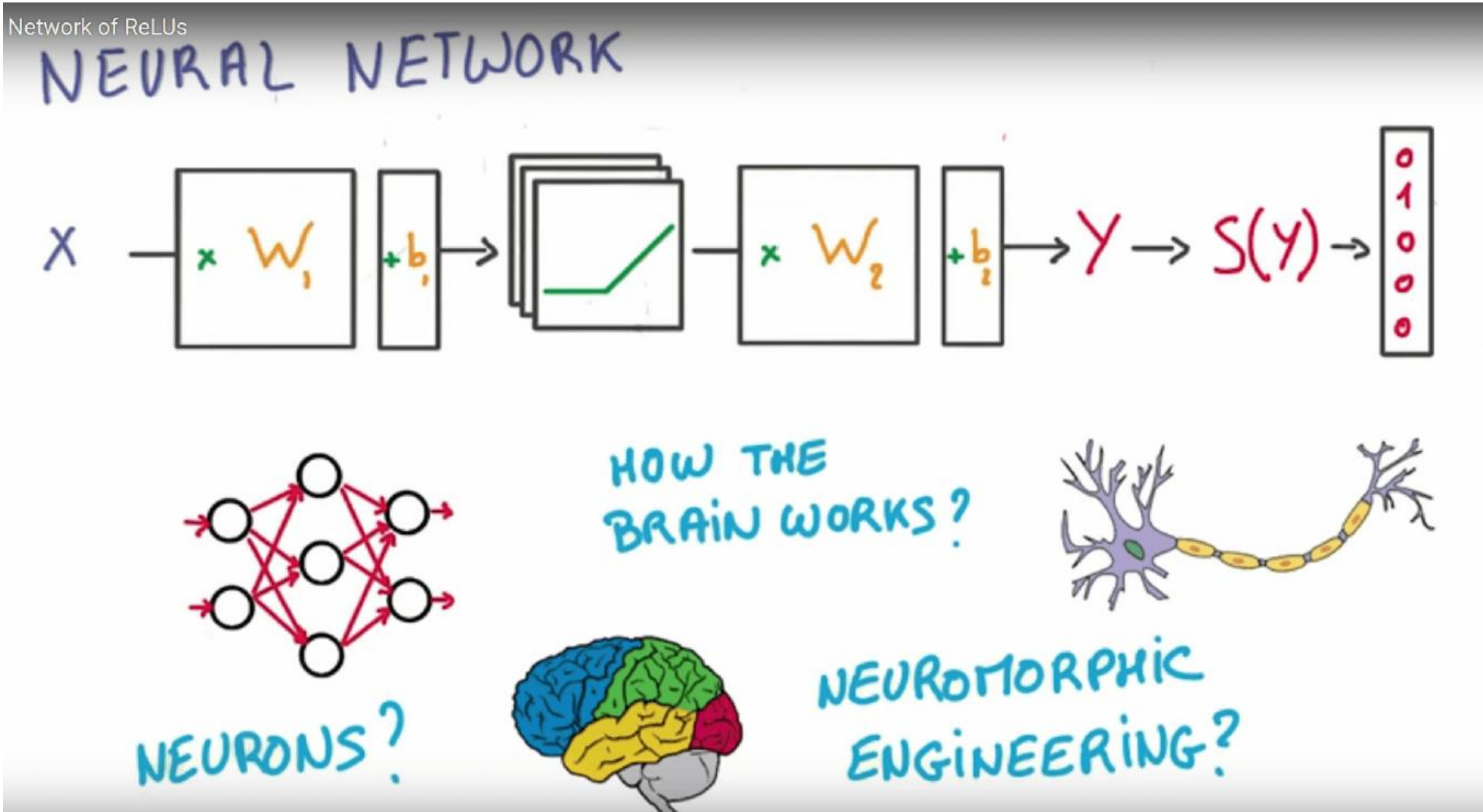
$$Y = W_1 \underset{\uparrow}{\,} W_2 \underset{\uparrow}{\,} W_3 X = \cancel{W} X$$

NON - LINEARITIES

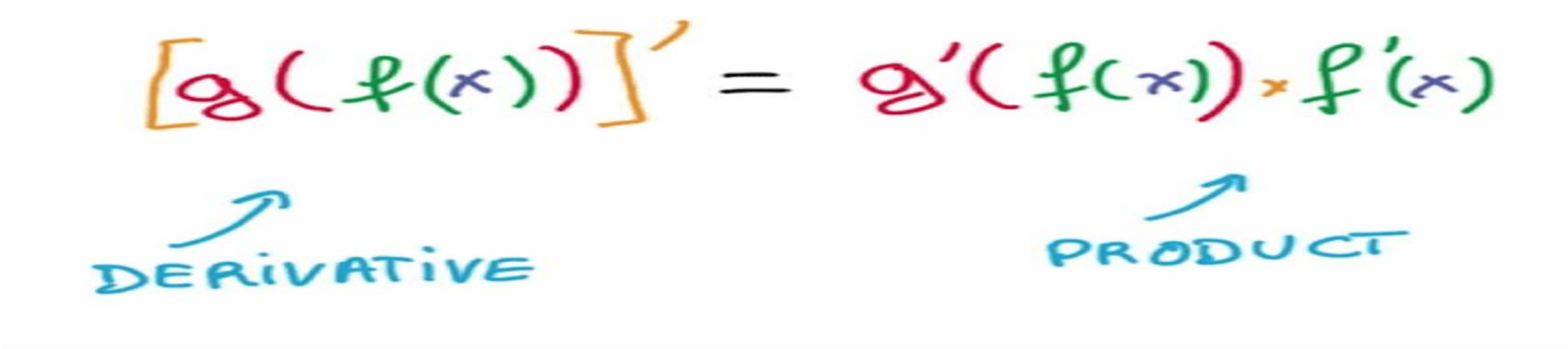
RELU: Rectified Linear Unit

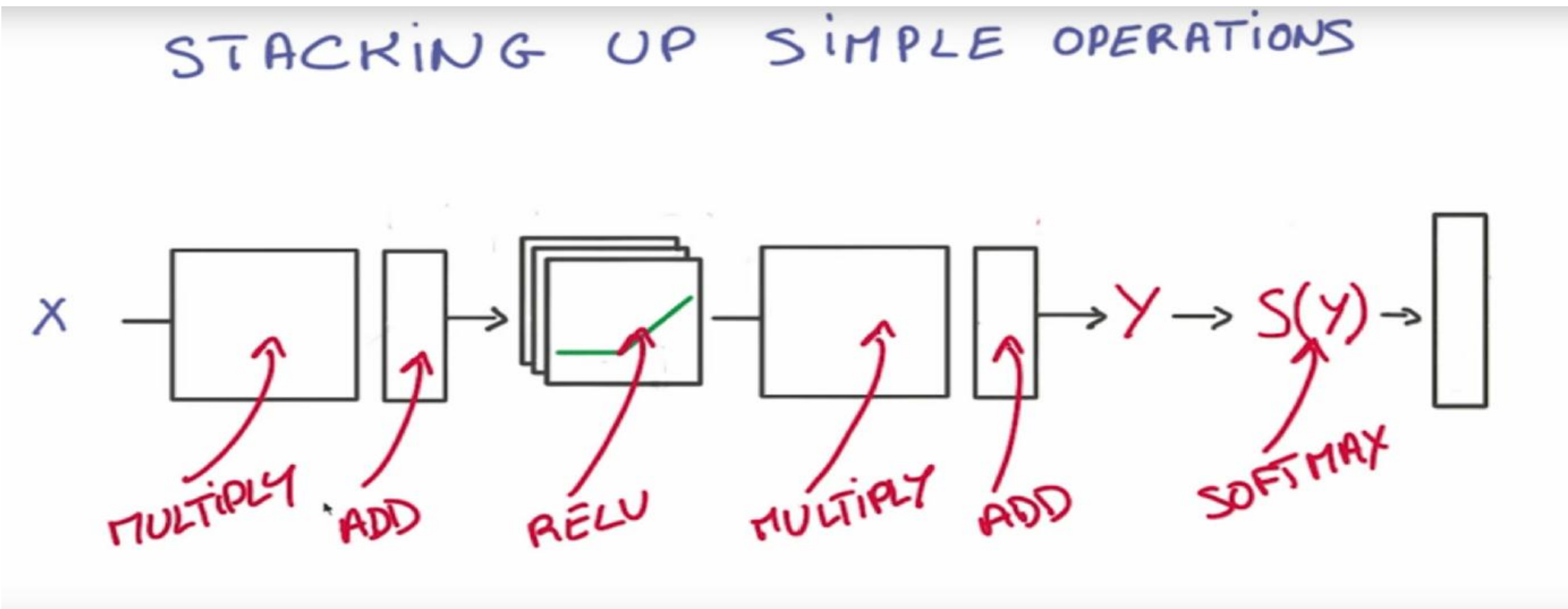


Networks of RELU

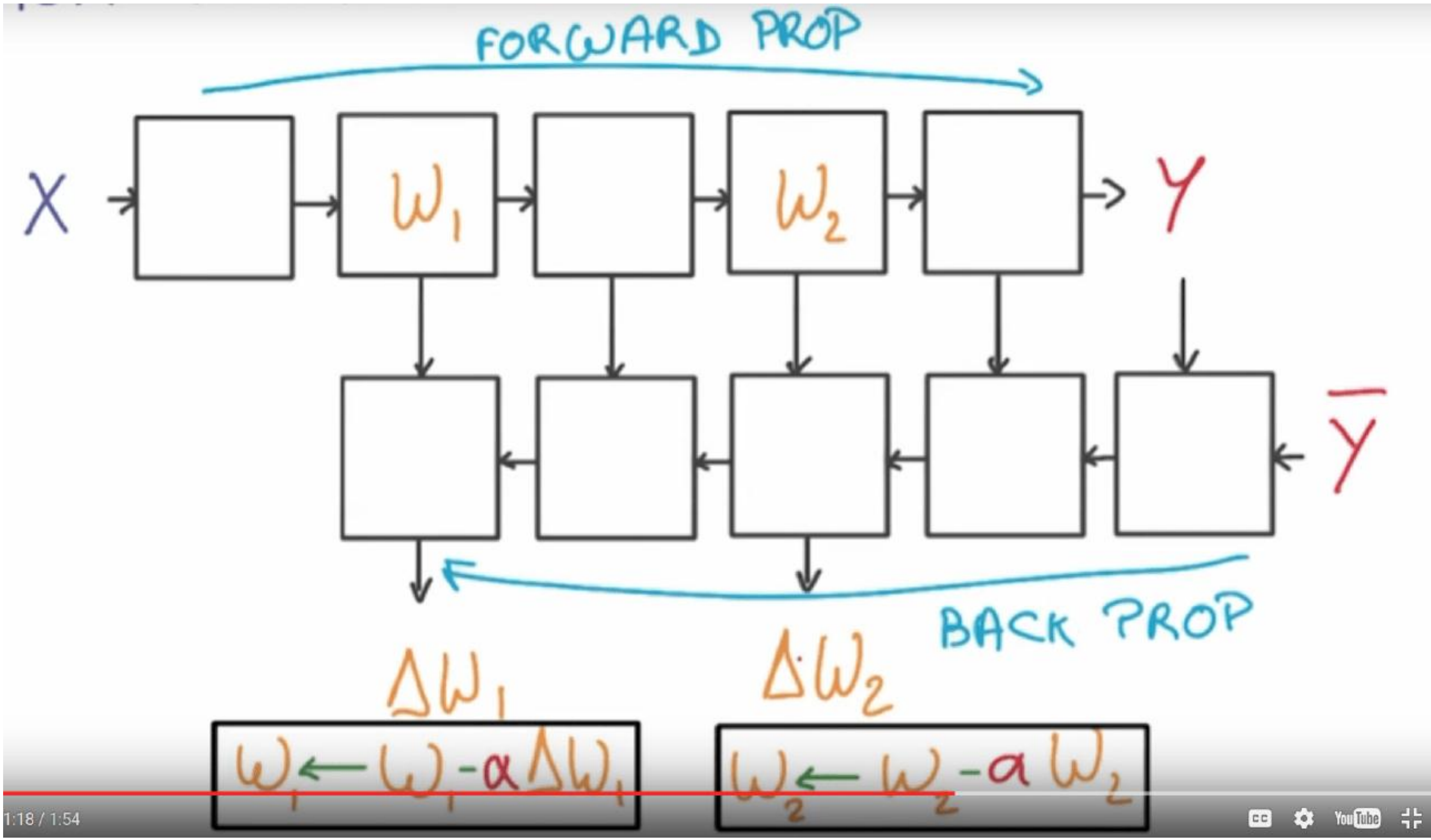


The Chain Rule

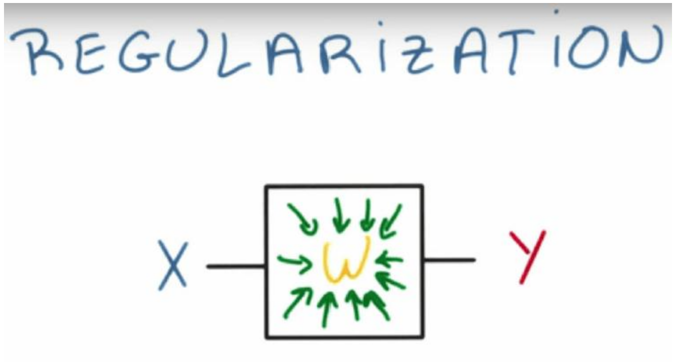


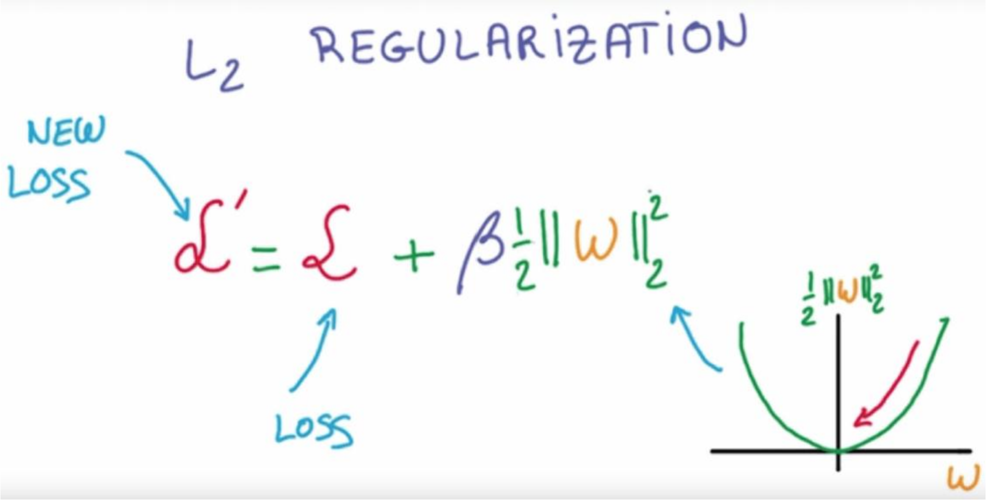


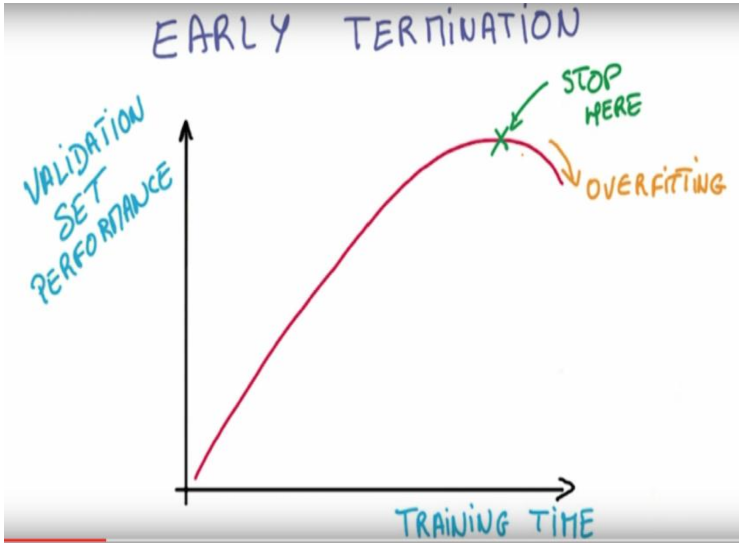
Back - propagation

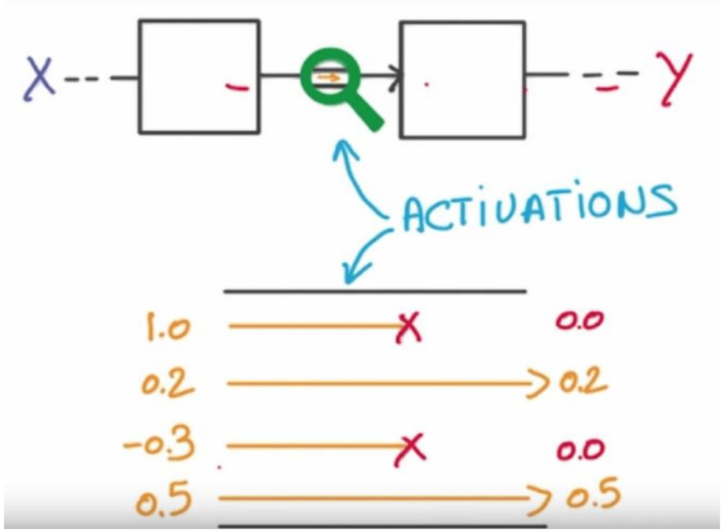


Optimisation tricks

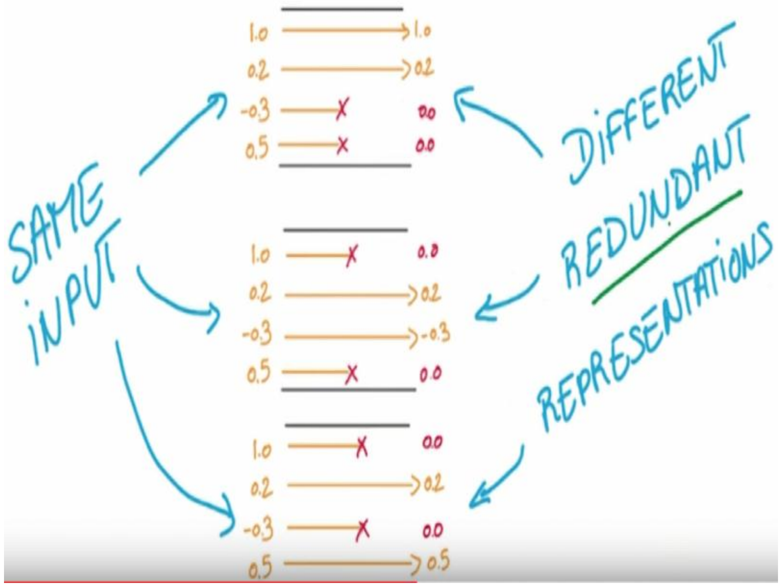


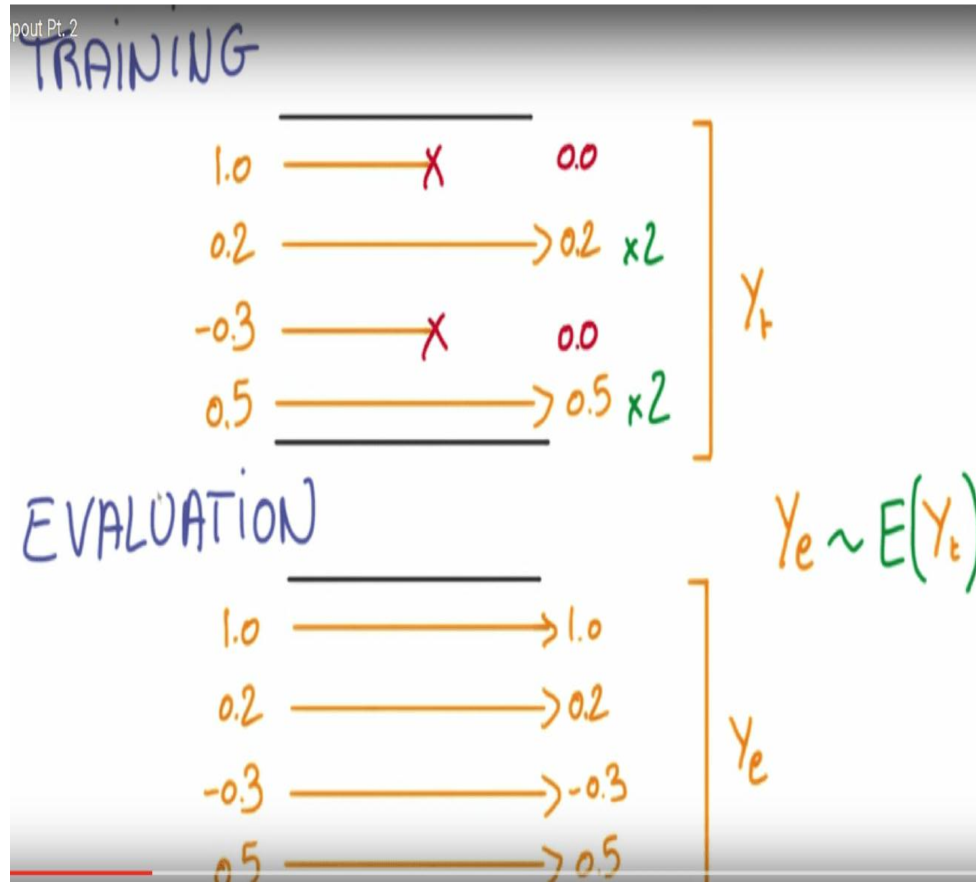




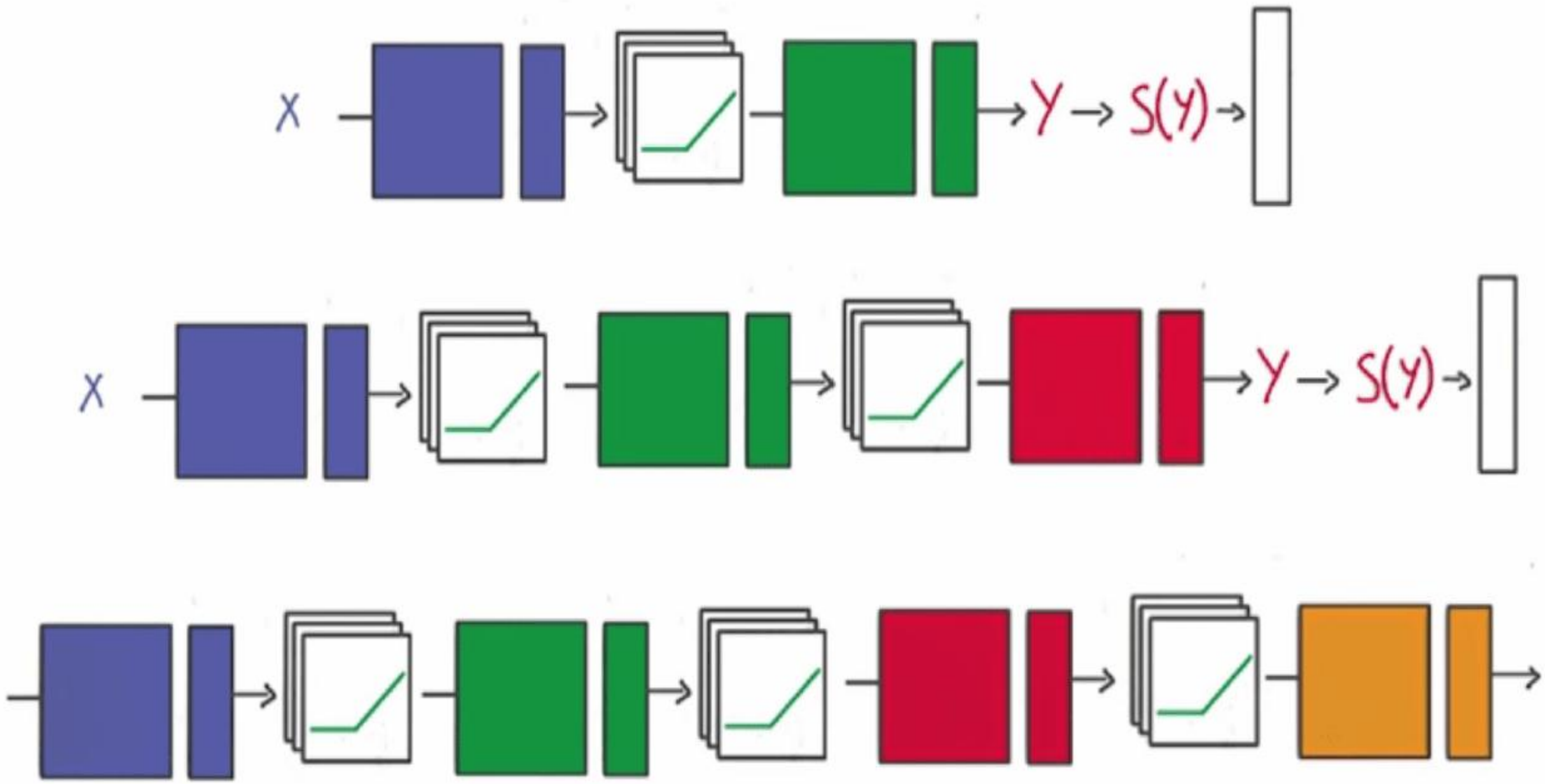


Optimisation trick: dropout

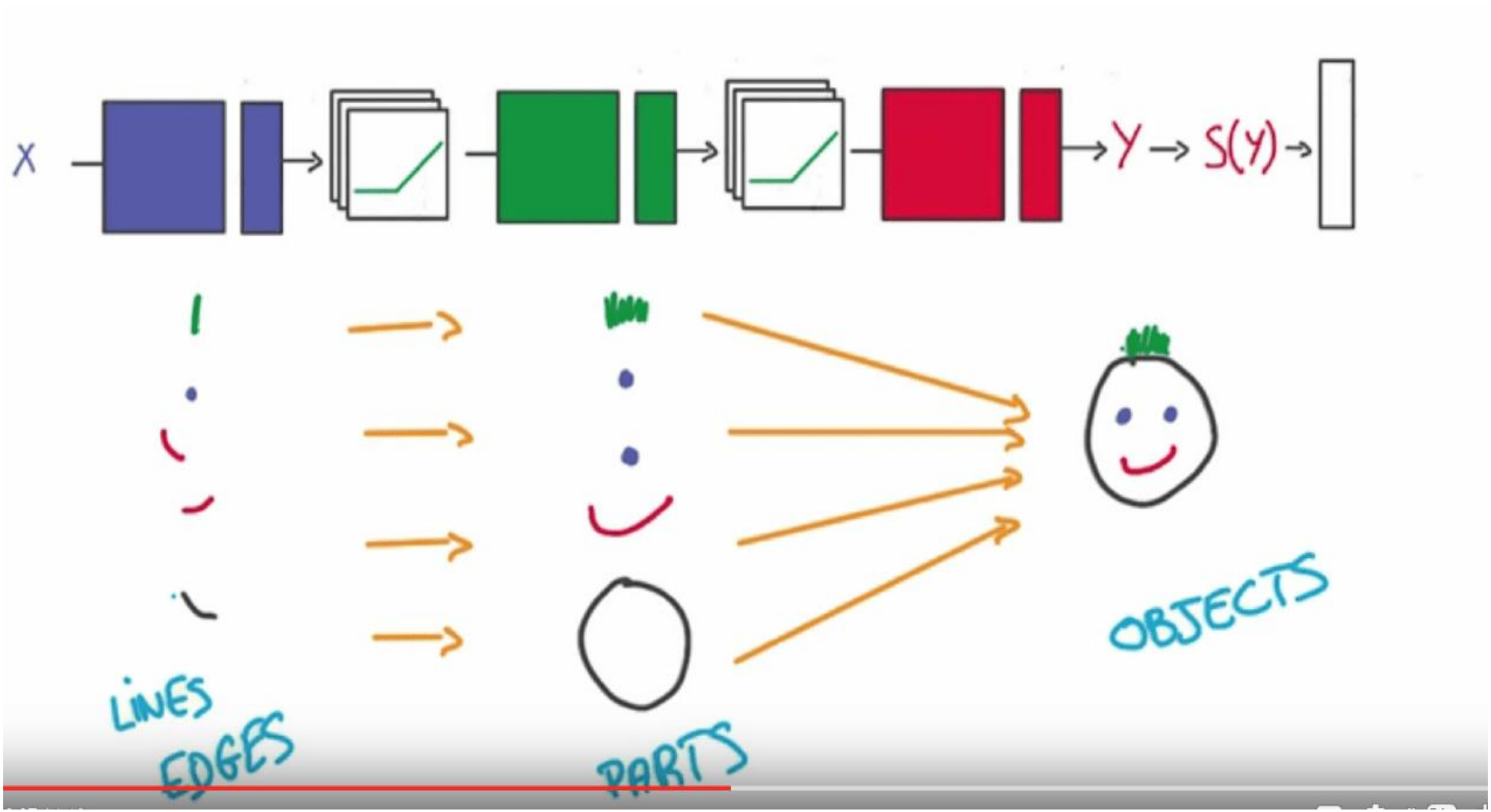




Deep networks



Deep networks



tensorflow.org/paper/whitepaper2015.pdf

TensorFlow:
Large-Scale Machine Learning on Heterogeneous Distributed Systems

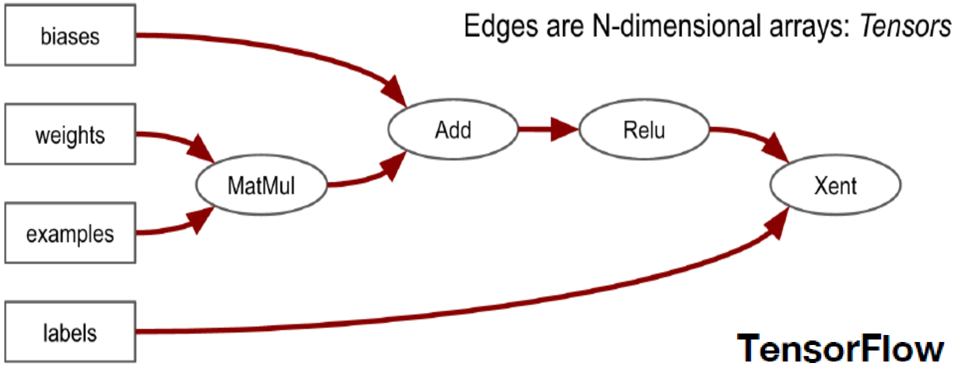
(Preliminary White Paper, November 9, 2015)

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng
Google Research*

Abstract

TensorFlow [1] is an interface for expressing machine learning algorithms, and an implementation for executing such algorithms. A computation expressed using TensorFlow can be executed with little or no change on a wide variety of heterogeneous systems, ranging from mobile devices such as phones and tablets up to large-scale distributed systems of hundreds of machines and thousands of computational devices such as GPU cards. The system is flexible and can be used to express a wide variety of algorithms, including training and inference algorithms for deep neural network models, and it has been used for conducting research and for deploying machine learning systems into production across more than a dozen areas of computer science and other fields, including speech recognition, computer vision, robotics, information retrieval, natural language processing, geographic information extraction, and computational drug discovery. This paper describes the TensorFlow interface and an implementation of that interface that we have built at Google. The TensorFlow API and a reference implementation were released as an open-source package under the Apache 2.0 license in November, 2015 and are available at www.tensorflow.org.

TensorFlow is an open source software library for numerical computation using data flow graphs.



TensorFlow - lazy evaluation

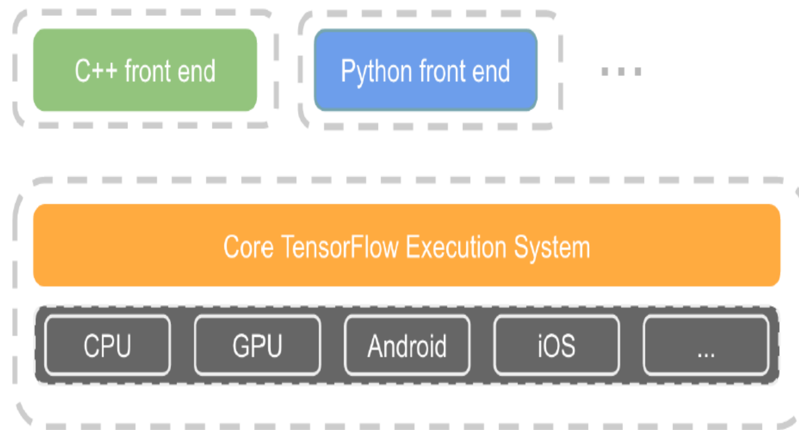
What is a Data Flow Graph?

Data flow graphs describe mathematical computation with a directed graph of nodes & edges. Nodes typically implement mathematical operations, but can also represent endpoints to feed in data, push out results, or read/write persistent variables. Edges describe the input/output relationships between nodes. These data edges carry dynamically-sized multidimensional data arrays, or tensors. The flow of tensors through the graph is where TensorFlow gets its name. Nodes are assigned to computational devices and execute asynchronously and in parallel once all the tensors on their incoming edges becomes available.

Key Features

- Deep Flexibility
- True Portability
- Connect Research and Production
- Auto-Differentiation
- Language Options
- Maximize Performance

TensorFlow



Parallel Execution

- Launch graph in a Session
- Request output of some Ops with Run API
- TensorFlow computes set of Ops that must run to compute the requested outputs
- Ops execute, in parallel, as soon as their inputs are available

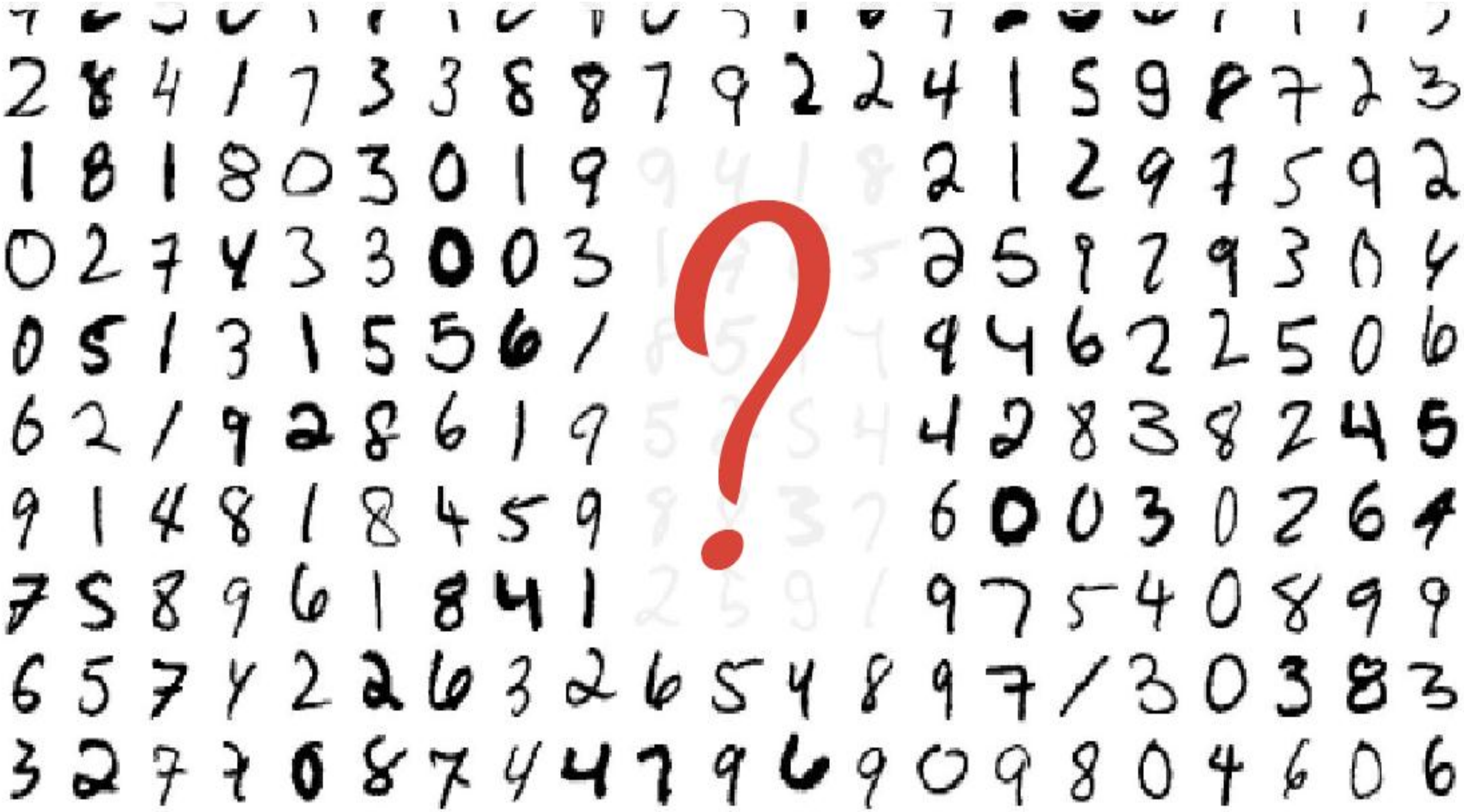
There are hundreds of predefined Ops (and easy to add more)

- Basics: constant, random, placeholder, cast, shape
- Variables: assign, assign_sub, assign_add
- Queues: enqueue, enqueue_batch, dequeue, blocking or not.
- Logical: equal, greater, less, where, min, max, argmin, argmax.
- Tensor computations: all math ops, matmul, determinant, inverse, cholesky.
- Images: encode, decode, crop, pad, resize, color spaces, random perturbations.
- Sparse tensors: represented as 3 tensors.
- ...

And many neural net specific Ops

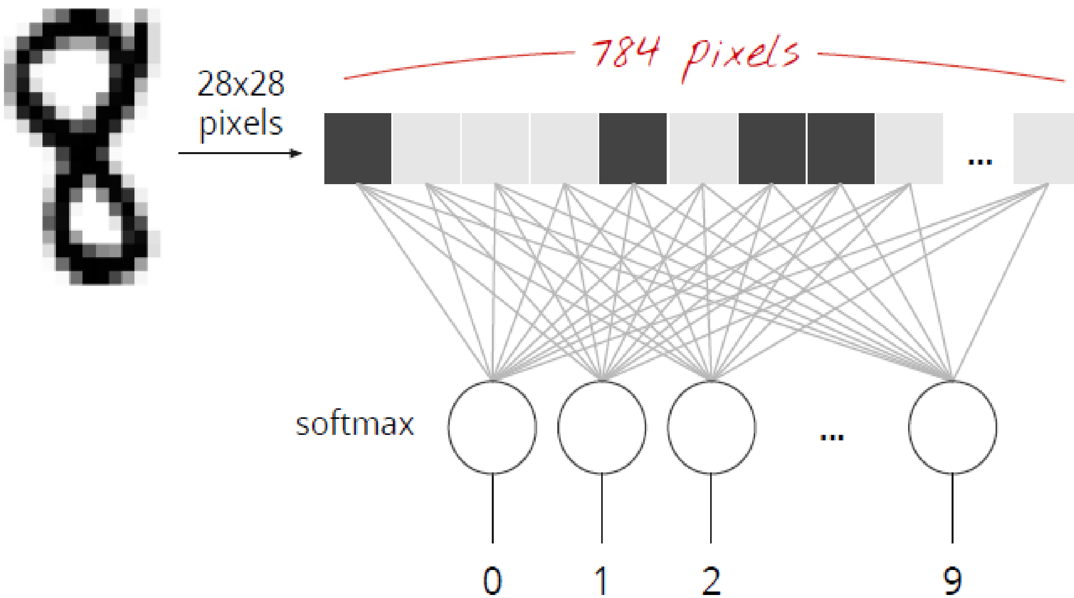
- Activations: sigmoid, tanh, relu, dropout, ...
- Pooling: avg, max.
- Convolutions: with many options.
- Normalization: local, batch, moving averages.
- Classification: softmax, softmax loss, cross entropy loss, topk.
- Embeddings: lookups/gather, scatter/updates.
- Sampling: candidate sampler (various options), sampling softmax.
- Updates: "fused ops" to speed-up optimizer updates (Adagrad, Momentum.)
- Summaries: Capture information for visualization.

Hand-written diggits: MNIST



MNIST = Mixed National Institute of Standards and Technology - Download the dataset at http://yann.lecun.com/exdb/mnist/

Simple linear model



weighted sum of all pixels + bias

$$softmax(L_n) = \frac{e^{L_n}}{||e^L||}$$

neuron outputs

Slides from M. Gorner tutorial
http://www.youtube.com/watch?v=vq2nnJ4g6NO

TensorFlow full python code

initialisation

```
import tensorflow as tf

X = tf.placeholder(tf.float32, [None, 28, 28, 1])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
init = tf.initialize_all_variables()

# model
Y=tf.nn.softmax(tf.matmul(tf.reshape(X,[-1, 784]), W) + b)

# placeholder for correct answers
Y_ = tf.placeholder(tf.float32, [None, 10])

# loss function
cross_entropy = -tf.reduce_sum(Y_ * tf.log(Y))

# % of correct answers found in batch
is_correct = tf.equal(tf.argmax(Y,1), tf.argmax(Y_,1))
accuracy = tf.reduce_mean(tf.cast(is_correct,tf.float32))
```

model

success metrics

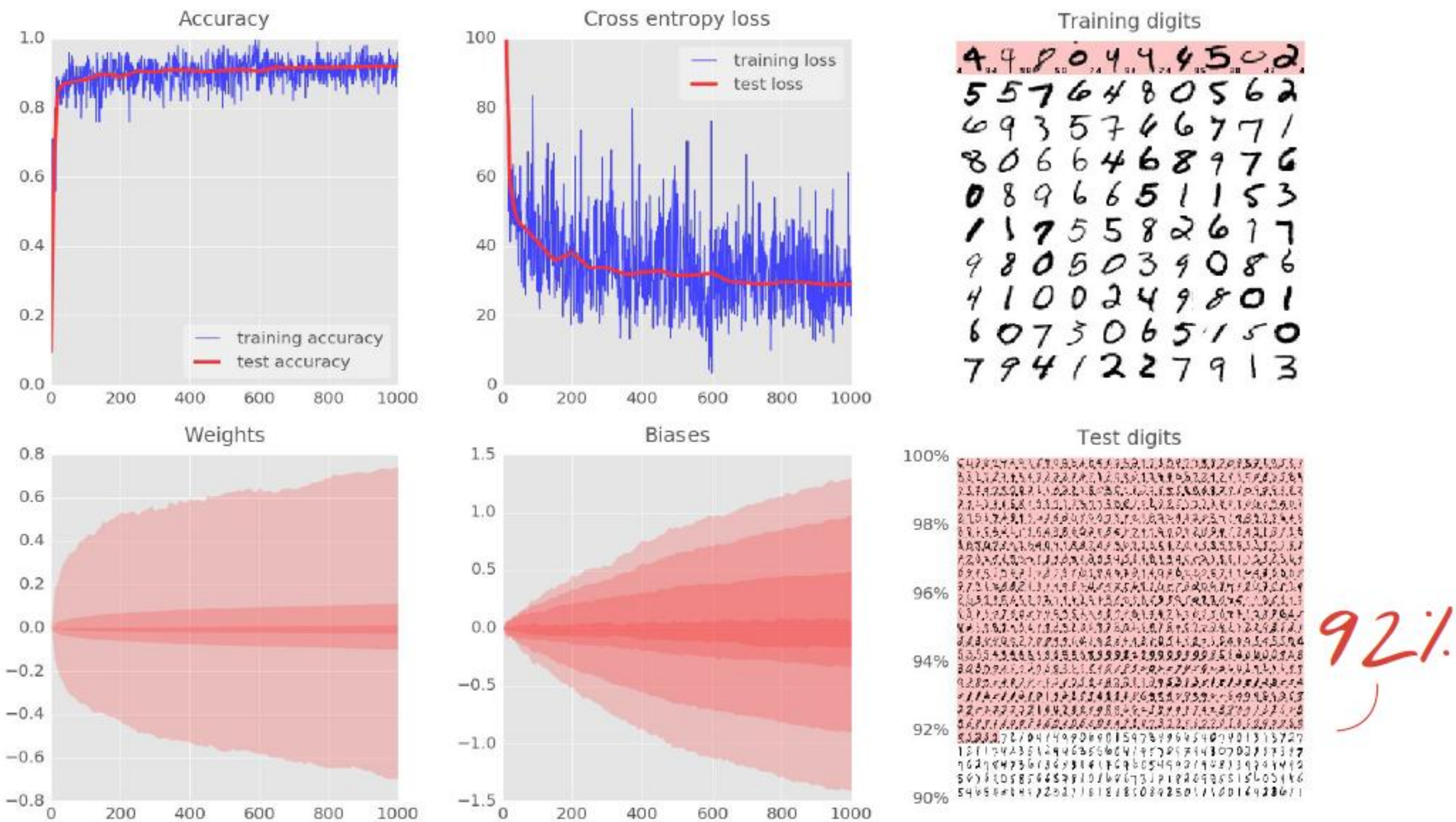
training step

```
optimizer = tf.train.GradientDescentOptimizer(0.003)
train_step = optimizer.minimize(cross_entropy)

sess = tf.Session()
sess.run(init)

for i in range(10000):
    # load batch of images and correct answers
    batch_X, batch_Y = mnist.train.next_batch(100)
    train_data={X: batch_X, Y_: batch_Y}

    # train
    sess.run(train_step, feed_dict=train_data)

    # success ? add code to print it
    a,c = sess.run([accuracy, cross_entropy], feed=train_data)

    # success on test data ?
    test_data={X:mnist.test.images, Y_:mnist.test.labels}
    a,c = sess.run([accuracy, cross_entropy], feed=test_data)
```

Run

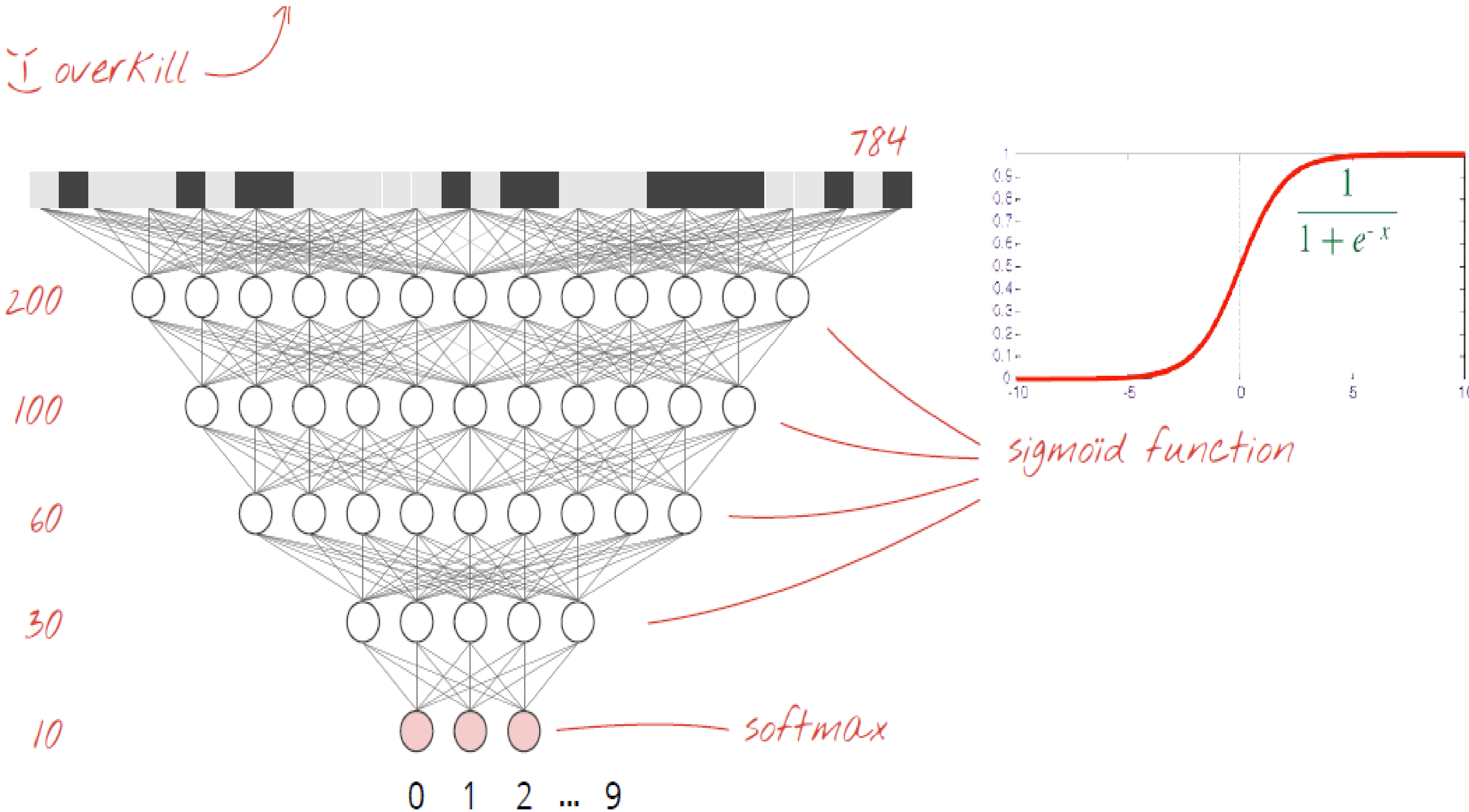
Slides from M. Gorner@youtube

Simple linear model



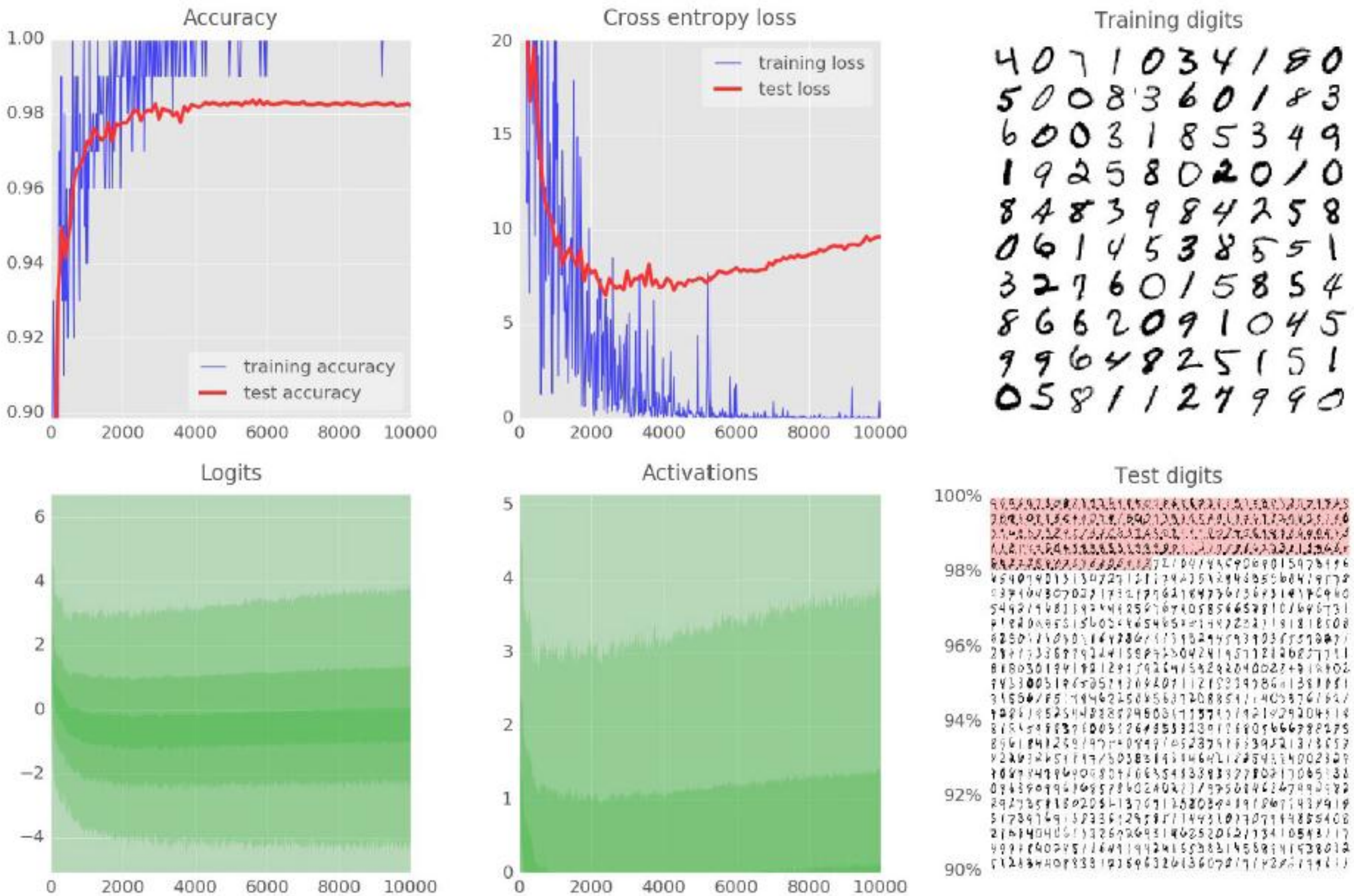
Slides from M. Gorner@youtube

Multi-layer connected network



Slides from M. Gorner@youtube

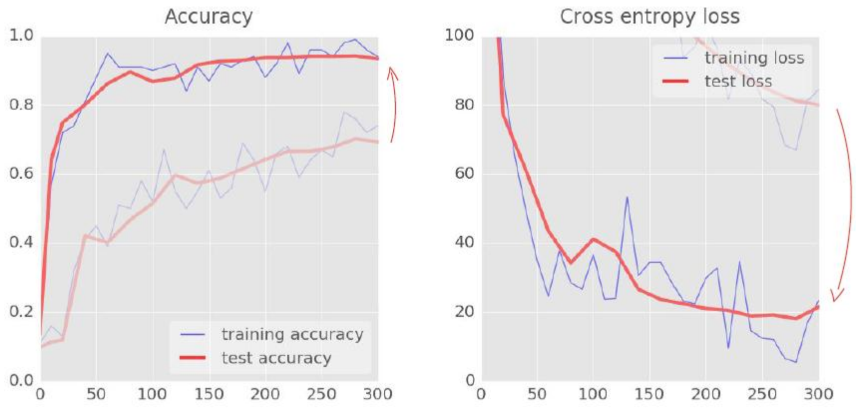
Multi-layer connected network



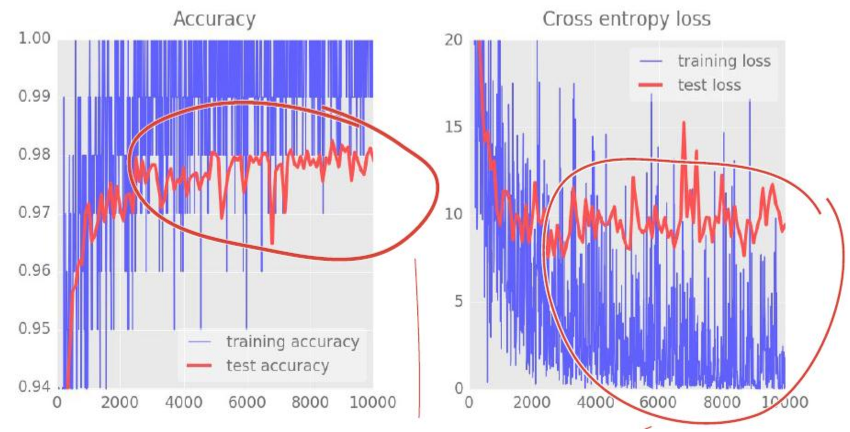
98%

All tricks count

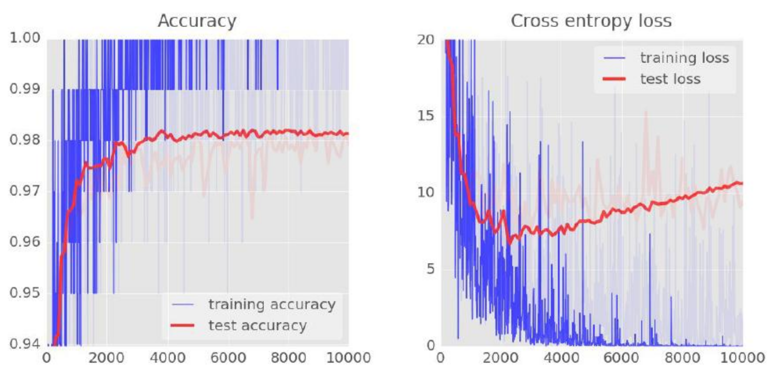
Use RELU



But noisy accuracy

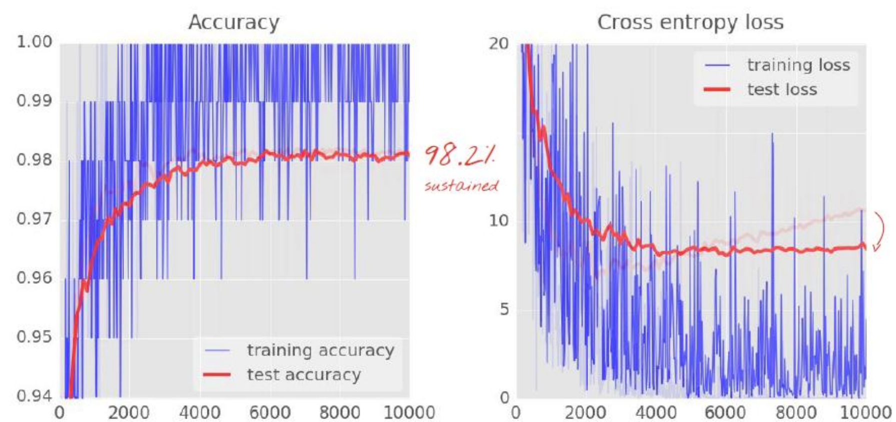


Exponentialy reduce learning rates



Learning rate 0.003 at start then dropping exponentially to 0.0001

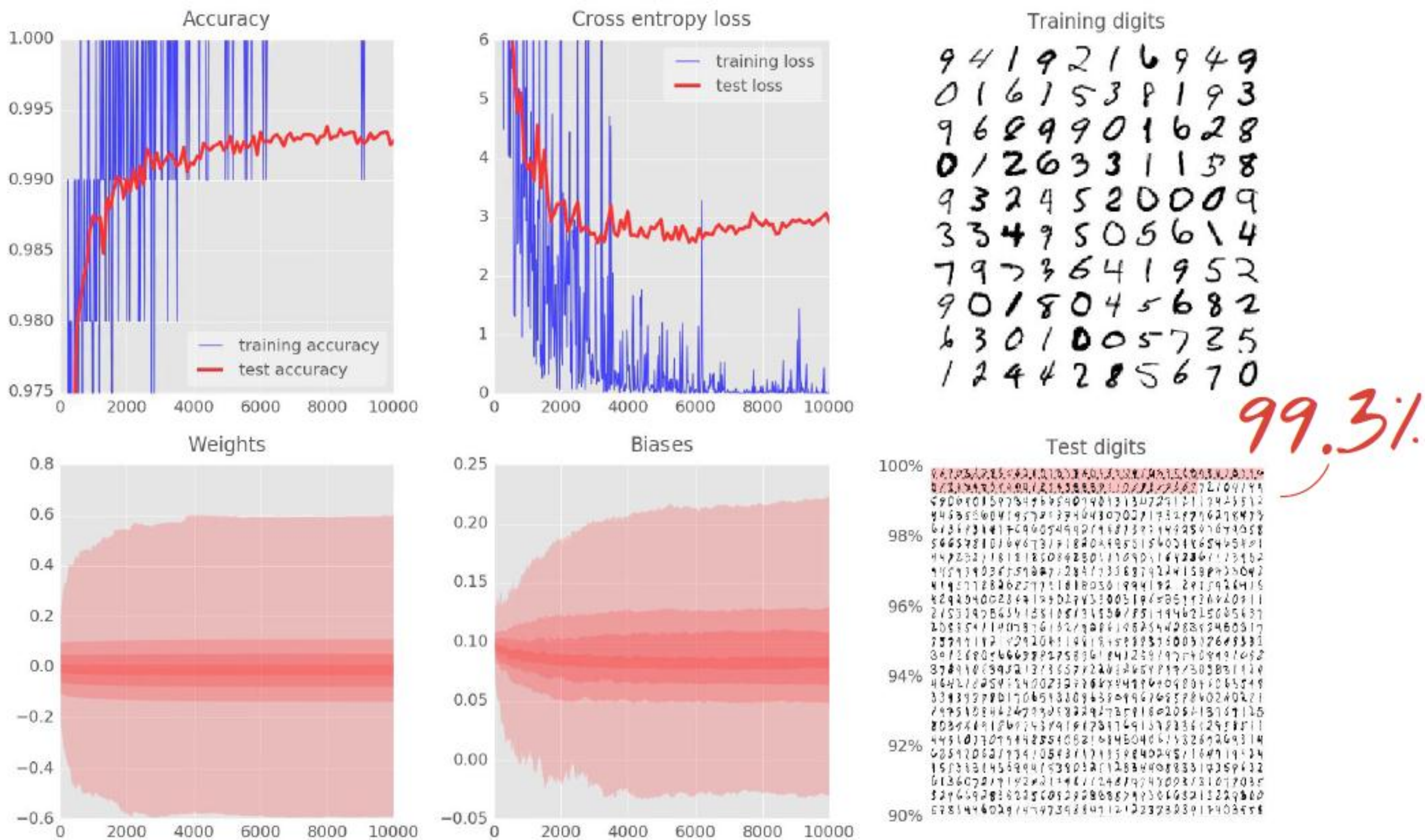
Add drop-out



RELU, decaying learning rate 0.003 -> 0.0001 and dropout 0.75

Slides from M. Gorner@youtube

Can do better with conv network



References

http://www.deeplearning.book.org

http://download.tensorflow.org/paper/whitepaper2015.pdf

https://www.tensorflow.org/

http://www.youtube.com/watch?v=vq2nnJ4g6NO

https://www.udacity.com/course/deep-learning--ud730

TensorFlow application

PHYSICAL REVIEW D 94, 093001 (2016)

Potential for optimizing the Higgs boson *CP* measurement in $H \to \tau\tau$ decays at the LHC including machine learning techniques

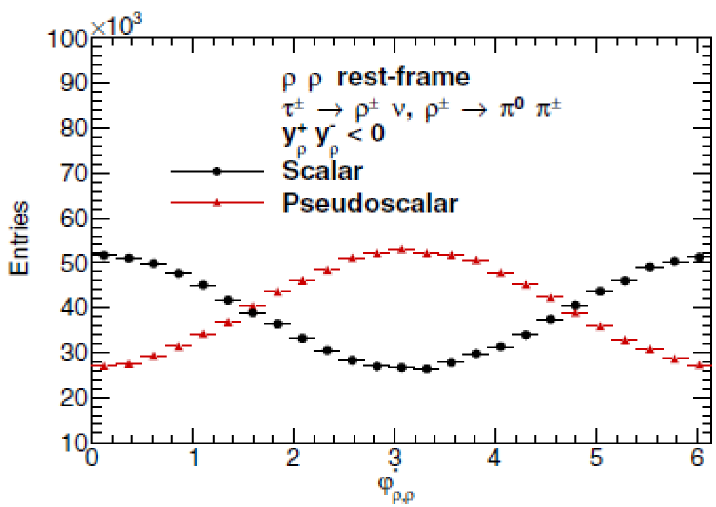
R. Józefowicz,[1] E. Richter-Was,[2] and Z. Was[3]
[1]*Open AI, San Francisco, California 94110, USA*
[2]*Institute of Physics, Jagellonian University, Lojasiewicza 11, 30-348 Krakow, Poland*
[3]*Institute of Nuclear Physics Polish Academy of Sciences, PL-31342 Krakow, Poland*
(Received 15 August 2016; published 7 November 2016)

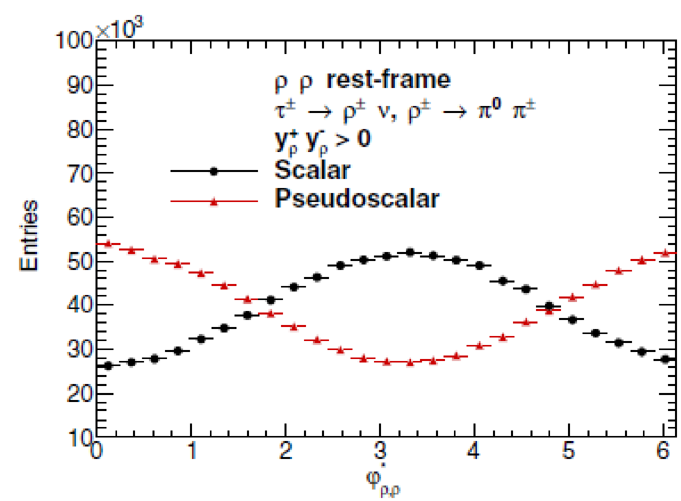
- **Multi-particle final state (cascade decays): 4 vectors**
- **CP information in correlations between decay planes and angles**
- **Physics intuition allowed to 1D optimal observables, but is there more to be explored.**

Defining the problem

TABLE II. Dimensionality of the features which may be used in each discussed configuration of the decay modes. Note that in principle $y_i^{\pm}$, $y_k^{\mp}$ may be calculated in the rest frame of the resonance pair used to define $\varphi^*_{i,k}$ planes, but in practice, a choice of the frames is of no numerically significant effect. We do not distinguish such variants.

Features/variables	Decay mode: $\rho^{\pm} - \rho^{\mp}$ $\rho^{\pm} \to \pi^0\pi^{\pm}$	Decay mode: $a_1^{\pm} - \rho^{\mp}$ $a_1^{\pm} \to \rho^0\pi^{\pm}$, $\rho^0 \to \pi^+\pi^-$ $\rho^{\mp} \to \pi^0\pi^{\mp}$	Decay mode: $a_1^{\pm} - a_1^{\mp}$ $a_1^{\pm} \to \rho^0\pi^{\pm}$, $\rho^0 \to \pi^+\pi^-$
$\varphi^*_{i,k}$	1	4	16
$\varphi^*_{i,k}$ and y_i, y_k	3	9	24
$\varphi^*_{i,k}$, 4-vectors	25	36	64
$\varphi^*_{i,k}$, y_i, y_k and m_i, m_k	5	13	30
$\varphi^*_{i,k}$, y_i, y_k, m_i, m_k and 4-vectors	29	45	78
$\varphi^*_{i,k}$, y_i, y_k, m_i, m_k and 4-vectors	29	45	78





Defining NN model

- 6 hiden layers, each 300 nodes, with RELU activation function

$$D \to 300 \to 300 \to 300 \to 300 \to 300 \to 300 \to 1.$$

- Sigmoid function on last layer [$\text{sigmoid}(x) = 1/(1+\exp(-x))$]
- Metric: negative log-likelihood of the rue targets under Bernoulli distribution

$$-\log p(y|y_h) = -(y == 0) * \log(y_h) - (y == 1) * \log(1 - y_h),$$

- Optimisation: SDG Adam algorithm
- Optimisation: batch normalisation, dropout
- Final score: weighted AUC

Defining NN model

```
# Linearly transforms X of shape [batch_size, size1] into [batch_size, size].
# Applies X -> XW+b, where W and b are trainable parameters.
def linear(x, name, size, bias=True):
    w = tf.get_variable(name + "/W", [x.get_shape()[1], size])
    b = tf.get_variable(name + "/b", [1, size],
                        initializer=tf.zeros_initializer)
    return tf.matmul(x, w)+b
```

```
# Applies batch normalization trick from https://arxiv.org/abs/1502.03167
# by normalizing each feature in a batch.
def batch_norm(x, name):
    mean, var = tf.nn.moments(x, [0])
    normalized_x = (x - mean) * tf.rsqrt(var + 1e-8)
    gamma = tf.get_variable(name + "/gamma", [x.get_shape()[-1]],
          initializer=tf.constant_initializer(1.0))
    beta = tf.get_variable(name + "/beta", [x.get_shape()[-1]])
    return gamma * normalized_x + beta
```

```
class NeuralNetwork(object):

  def __init__(self, num_features, batch_size, num_layers=6, size=300, lr=1e-3):
    # Each input x is represented by a given number of features
    # and corresponding weights for target distributions A and B.
    self.x = x = tf.placeholder(tf.float32, [batch_size, num_features])
    self.wa = wa = tf.placeholder(tf.float32, [batch_size])
    self.wb = wb = tf.placeholder(tf.float32, [batch_size])
    # The model will predict a single number, which is a probability of input x
    # belonging to class A. That probability is equal to wa / (wa+wb).
    y=wa / (wa+wb)
    y = tf.reshape(y, [-1, 1])

    # We apply multiple layers of transformations where each layer consists of
    # linearly transforming the features, followed by batch normalization
    (described above)
    # and ReLU nonlinearity (which is an elementwise operation: x -> max(x, 0))
    for i in range(num_layers):
      x = tf.nn.relu(batch_norm(linear(x, "linear_%d" % i, size), "bn_%d" % i))

    # Finally, the output is transformed into a single number.
    # After applying sigmoid nonlinearity (x -> 1 / (1 + exp(-x))) we'll interpret
    that number
    # as a probability of x belonging to class A.
    x = linear(x, "regression", 1)
    self.p = tf.nn.sigmoid(x)

    # The objective to optimize is negative log likelihood under Bernoulli
    distribution:
    # loss = - (p(y==A) * log p(y==A|x) + p(y==B) * log p(y==B|x))
    self.loss = loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits
    (x, y))
    # The model parameters are optimized using gradient-based Adam optimizer
    # (https://arxiv.org/abs/1412.6980) to minimize the loss on the training data.
     self.train_op = tf.train.AdamOptimizer(lr).minimize(loss)
```

Results

TABLE III. Average probability p_i (calculated as explained in Sec. III B) that a model predicts correctly event x_i to be of a type A (scalar), with training being performed for separation between types A and B (pseudoscalar).

Features/variables	Decay mode: $\rho^\pm - \rho^\mp$ $\rho^\pm \to \pi^0\pi^\pm$	Decay mode: $a_1^\pm - \rho^\mp$ $a_1^\pm \to \rho^0\pi^\mp$, $\rho^0 \to \pi^+\pi^-$ $\rho^\mp \to \pi^0\pi^\mp$	Decay mode: $a_1^\pm - a_1^\mp$ $a_1^\pm \to \rho^0\pi^\pm$, $\rho^0 \to \pi^+\pi^-$
True classification	0.782	0.782	0.782
$\varphi^*_{i,k}$	0.500	0.500	0.500
$\varphi^*_{i,k}$ and y_i, y_k	0.624	0.569	0.536
4-vectors	0.638	0.590	0.557
$\varphi^*_{i,k}$, 4-vectors	0.638	0.594	0.573
$\varphi^*_{i,k}$, y_i, y_k and m_i^2, m_k^2	0.626	0.578	0.548
$\varphi^*_{i,k}$, y_i, y_k, m_i^2, m_k^2 and 4-vectors	0.639	0.596	0.573

- **NN can capture „optimal variables" but can do better given simple of 4-vectors.**
- **Given „simple" and „higher level" features can still improve in more complicated case.**
- **Will try now on the experimental data.**