INTRODUCTION TO DATA SCIENCE

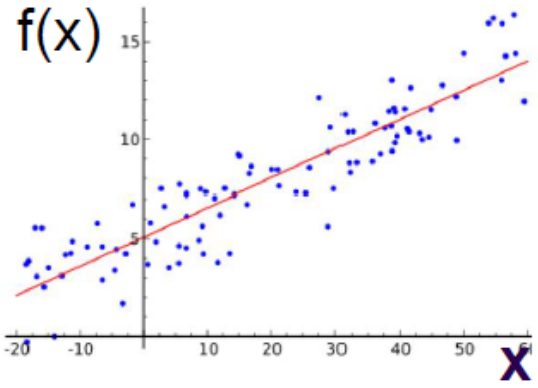
11/12/2024

WFAiS UJ, Informatyka Stosowana

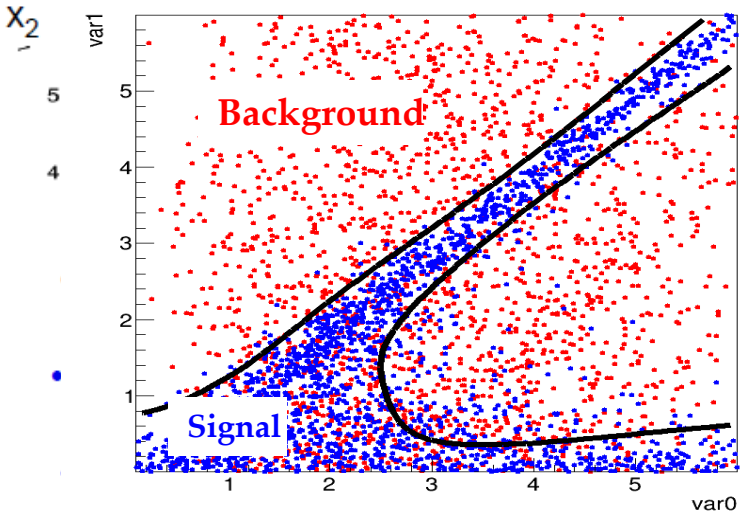
I stopień studiów

What are Multivariate Techniques

→ Many things … starting from "linear regression" …



to multivariate event classification



→ or w/o prior 'analytic' model

→ typically "multivariate"

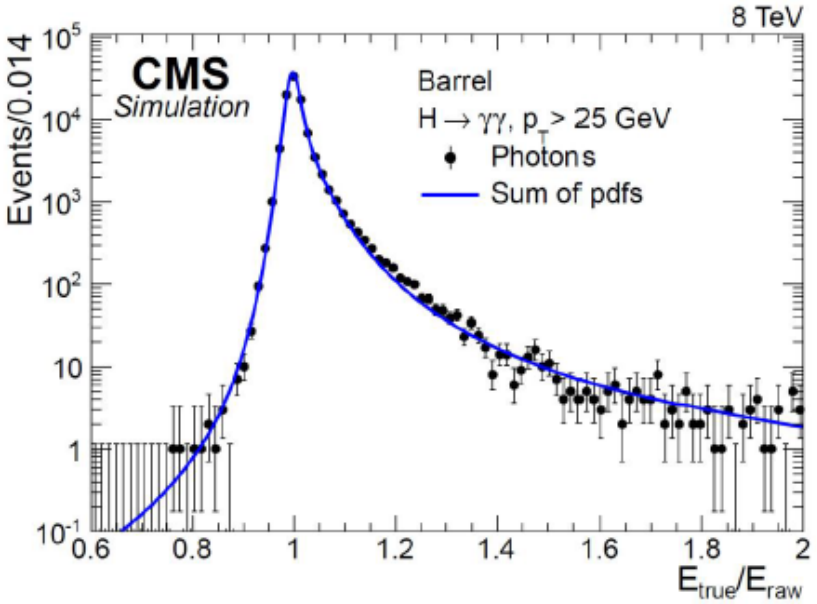
- Parameters depend on the 'joint distribution' $f(x_1, x_2)$

- 'learning from experience' → known data points

Machine Learning - Multivariate Techniques

- fitted (non-)analytic function may approximate:
  - target value → 'regression'

    (e.g. calorimeter calibration/correction function)



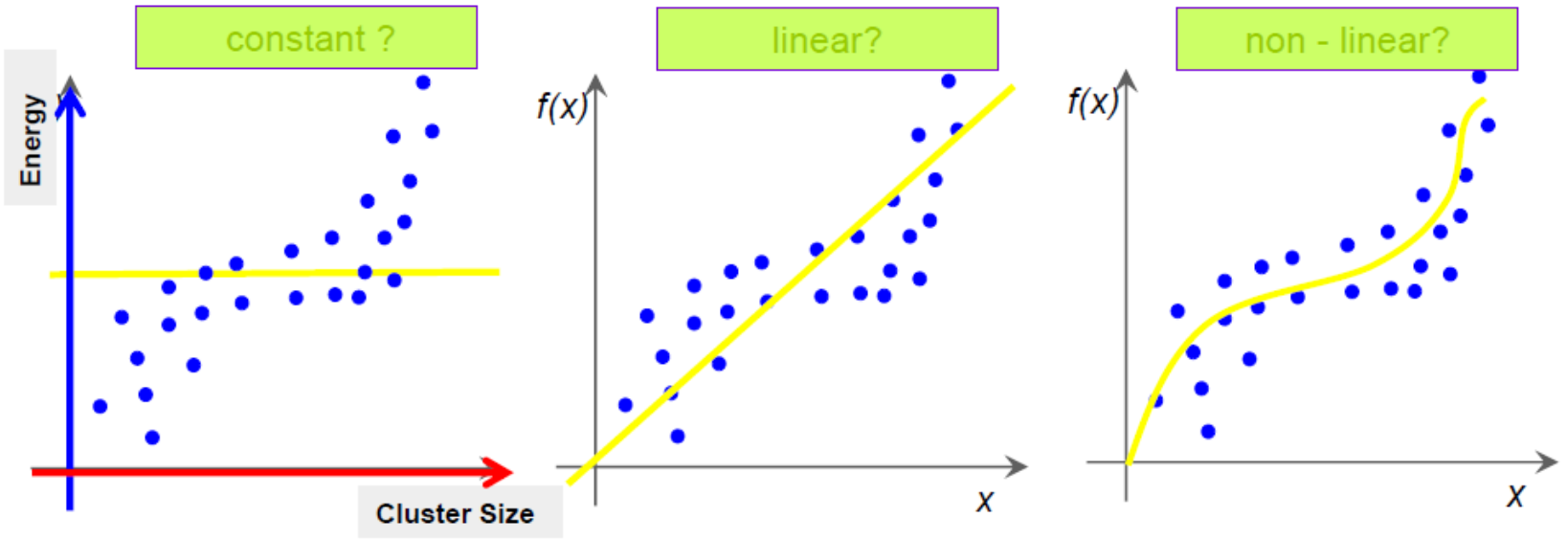
MC sample: γ +jets

- Raw energy in crystals, η, Φ
- Cluster shape variables
- Local cluster position variables (energy leakage)
- Pile-up estimators

→ predict energy correction (i.e. parameters in crystal-ball: pdf for energy measurement)

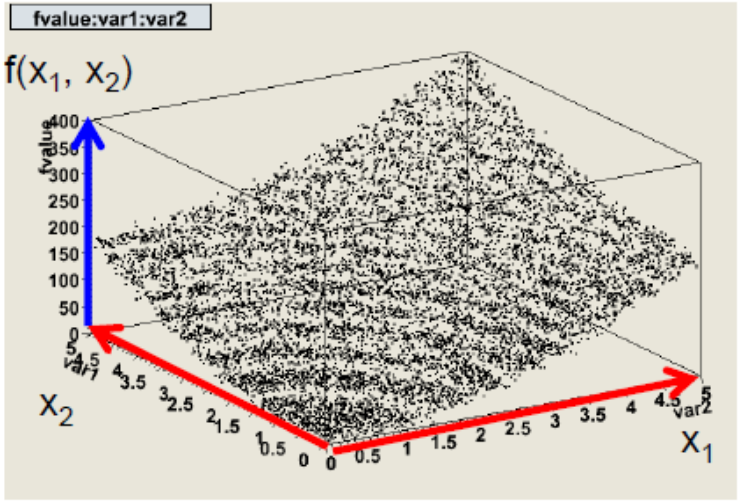
Regression

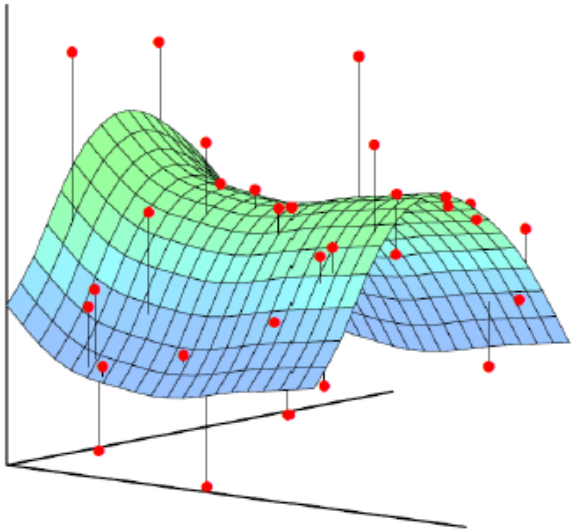
- 'known measurements" → model "functional behaviour"
- e.g. : photon energy as function "D"-variables: ECAL shower parameters + …



- known analytic model (i.e. nth -order polynomial) → Maximum Likelihood Fit)
- no model ?
  → "draw any kind of curve" and parameterize it?
- **seems trivial ?** → human brain has very good pattern recognition capabilities!
- what if you have **many** input variables?

Regression -> model functional behaviour





- "standard" regression → fit a known analytic function
  - *e.g.* $f(\mathbf{x}) = ax_1^2+bx_2^2+c$
- BUT most times: don't have a reasonable "model" ? → need something more general:
  - *e.g.* piecewise defined splines, kernel estimators, decision trees to approximate f(x)

Note: we are not interested in the 'fitted parameter(s)', it is not: "Newton deriving F=m·a"
→ just provide prediction of function values f(x) for new measurements x

Multi-Variate Classification

Consider events which can be either signal or background events.

Each event is characterized by *n* observables:

$$\vec{x} = (x_1, \ldots, x_n)$$ "feature vector"

Goal: classify events as signal or background in an optimal way.

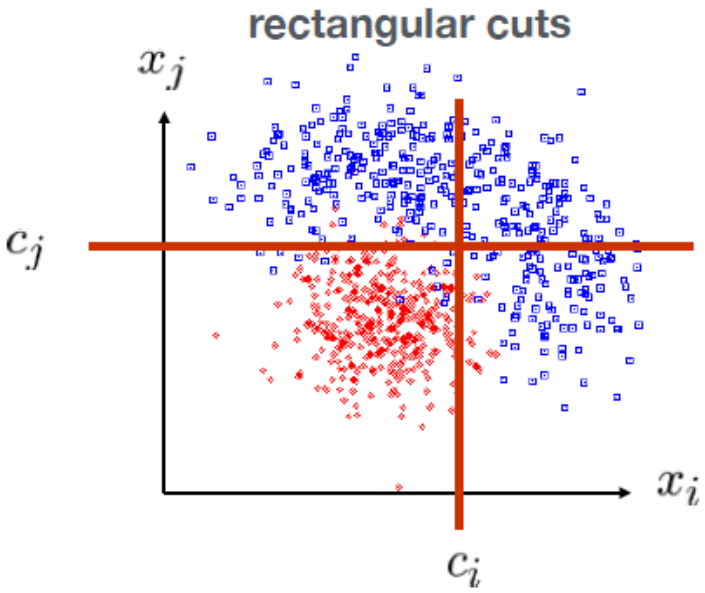
This is usually done by mapping the feature vector to a single variable, i.e., to scalar test statistic:

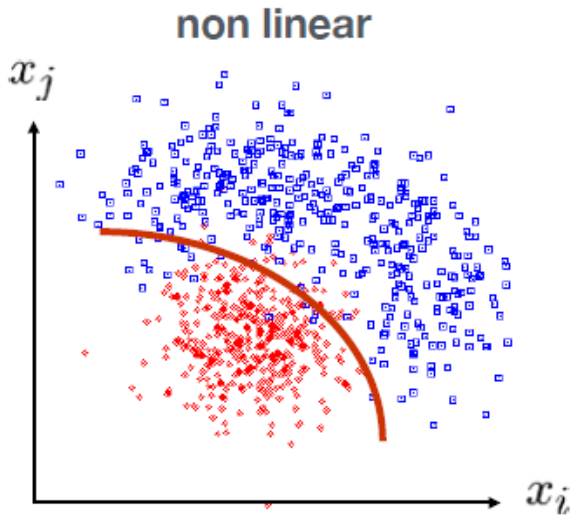
$$\mathbb{R}^n \to \mathbb{R}: \quad y(\vec{x})$$

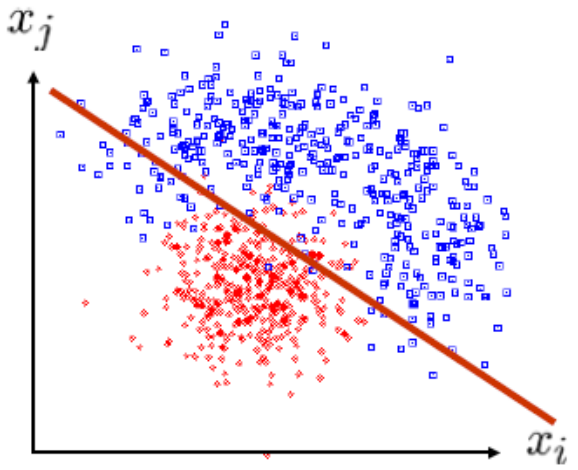
A cut $y > c$ to classify events as signal corresponds to selecting a potentially complicated hyper-surface in feature space. In general superior to classical "rectangular" cuts on the x_i.

Problem closely related to *machine learning* (*pattern recognition*, *data mining*, …)

Classification: Different Approaches



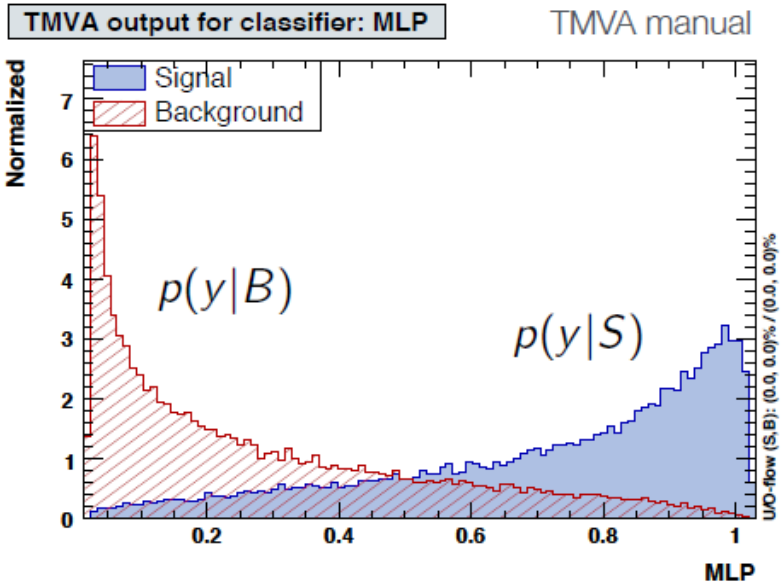




k-Nearest-Neighbor,
Boosted Decision Trees,
Multi-Layer Perceptrons,
Support Vector Machines
…

Signal Probability Instead of Hard Decisions

Example: test statistic y for signal and background from a Multi-Layer Perceptron (MLP):



Instead of a hard yes/no decision one can also define the probability of an event to be a signal event:

$$P_s(y) \equiv P(S|y) = \frac{p(y|S) \cdot f_s}{p(y|S) \cdot f_s + p(y|B) \cdot (1 - f_s)}, \qquad f_s = \frac{n_s}{n_s + n_b}$$

Machine learning: 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.

CLASSICAL MACHINE LEARNING

Data is pre-categorized or numerical

Data is not labeled in any way

SUPERVISED

Predict a category

Predict a number

CLASSIFICATION

«Divide the socks by color»

REGRESSION

«Divide the ties by length»

OUR FOCUS

UNSUPERVISED

Divide by similarity

Find hidden dependencies

Identify sequences

CLUSTERING

«Split up similar clothing into stacks»

ASSOCIATION

«Find what clothes I often wear together»

DIMENSION REDUCTION (generalization)

«Make the best outfits from the given clothes»

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

Where are the neural networks?

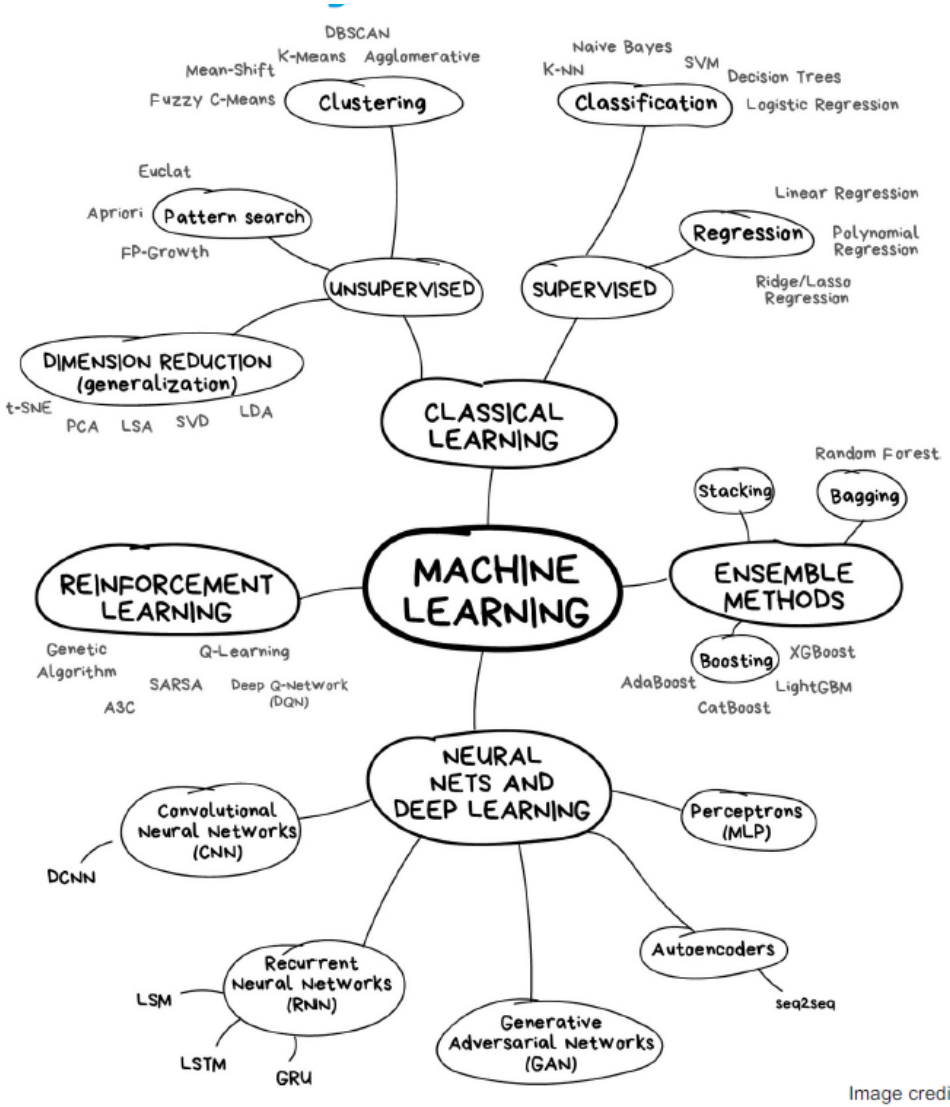


Image credit: https://vas3k.com/blog/machine_learning/

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*.

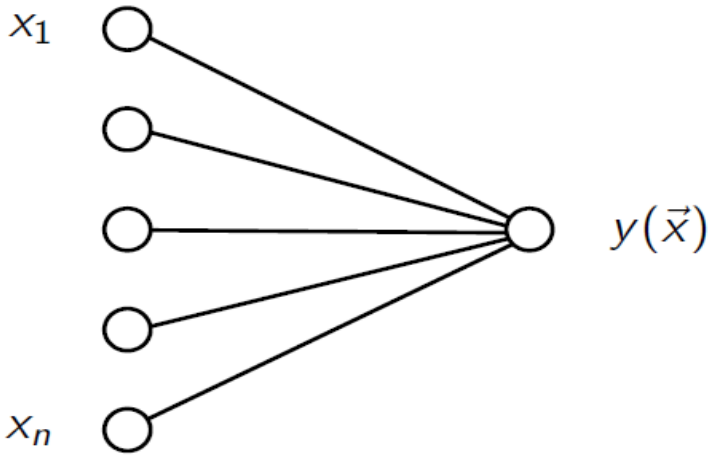
- These weights tell the neuron to respond more to one input and less to another. Weights are adjusted when training — that's how the network learns.

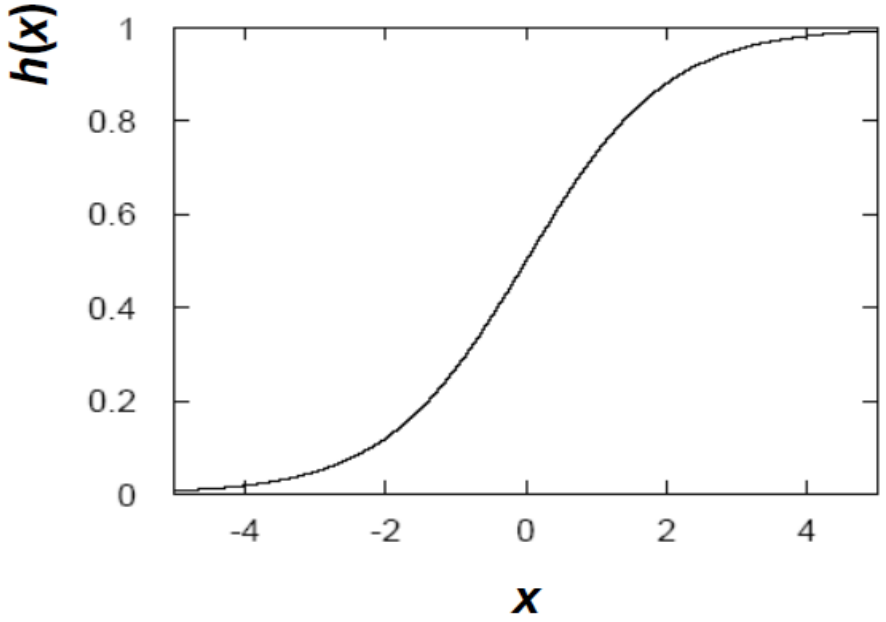
Perceptron

Discriminant: $$y(\vec{x}) = h\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)$$

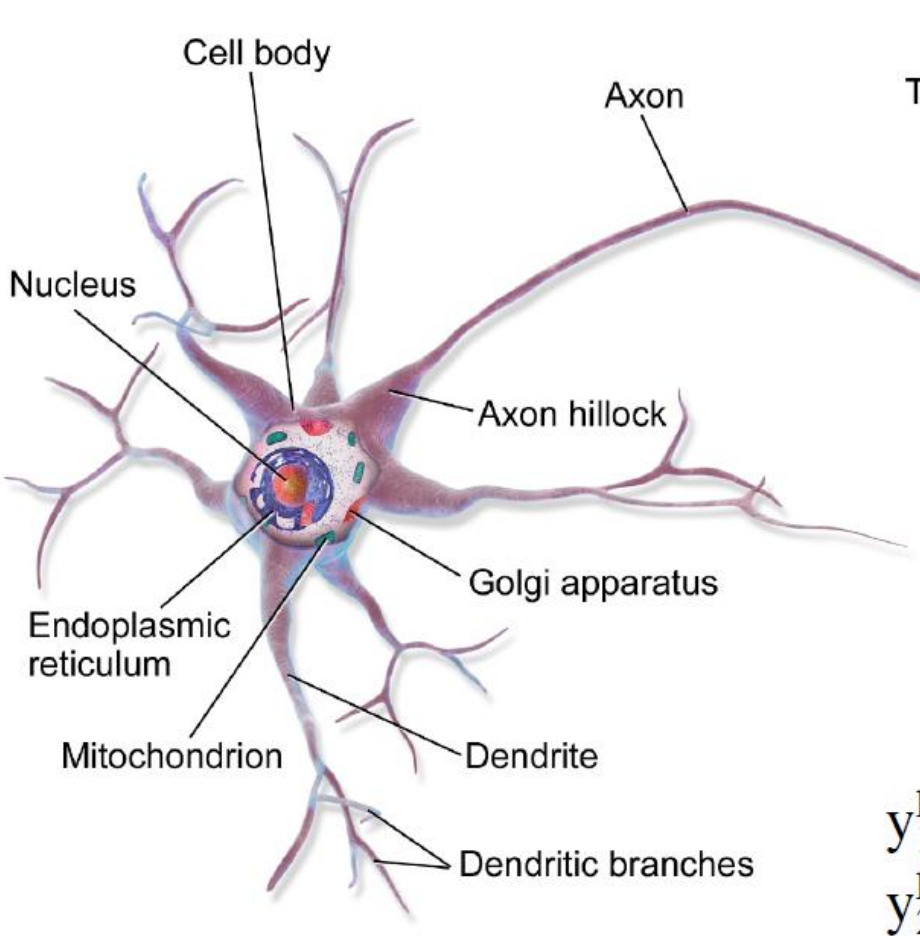
The nonlinear, monotonic function h is called *activation function*.

Typical choices for h: $\frac{1}{1+e^{-x}}$ ("sigmoid"), $\tanh x$



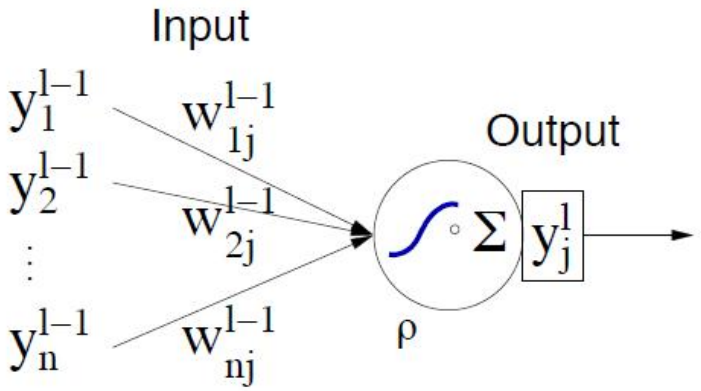


The Biological Inspiration: the Neuron

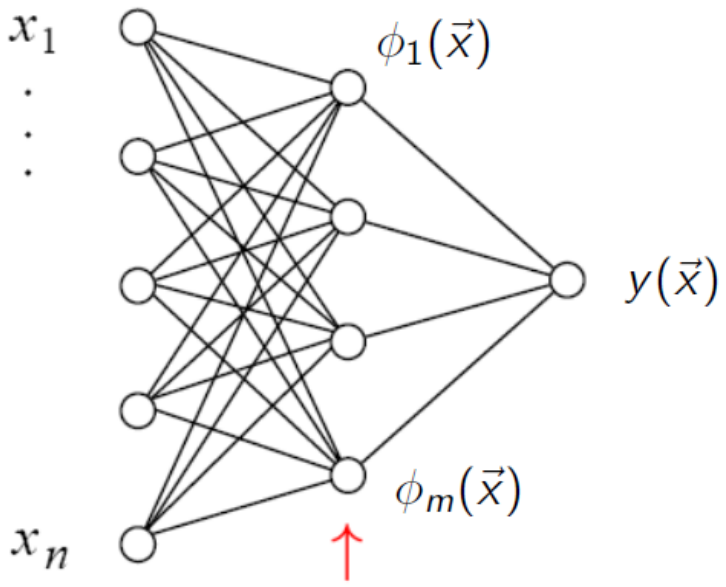


https://en.wikipedia.org/wiki/Neuron

Human brain:
10^{11} neurons, each with on average
7000 synaptic connections



Feedforward Neural Network with One Hiden Layer



superscripts indicates layer number

$$\phi_i(\vec{x}) = h\left(w_{i0}^{(1)} + \sum_{j=1}^{n} w_{ij}^{(1)} x_j\right)$$

$$y(\vec{x}) = h\left(w_{10}^{(2)} + \sum_{j=1}^{m} w_{1j}^{(2)} \phi_j(\vec{x})\right)$$

Straightforward to generalize to multiple hidden layers

Network Training

$\vec{x}_a$: training event, $a = 1, \ldots, N$

t_a : correct label for training event a

e.g., $t_a = 1, 0$ for signal and background, respectively

$\vec{w}$: vector containing all weights

Error function:

$$E(\vec{w}) = \frac{1}{2}\sum_{a=1}^{N}(y(\vec{x}_a, \vec{w}) - t_a)^2 = \sum_{a=1}^{N} E_a(\vec{w})$$

Weights are determined by minimizing the error function.

Backpropagation

Start with an initial guess $\vec{w}^{(0)}$ for the weights an then update weights after each training event:

$$\vec{w}^{(\tau+1)} = \vec{w}^{(\tau)} - \eta \nabla E_a(\vec{w}^{(\tau)})$$

η: learning rate

Let's write network output as follows:

$$y(\vec{x}) = h(u(\vec{x})) \quad \text{with} \quad u(\vec{x}) = \sum_{j=0}^{m} w_{1j}^{(2)} \phi_j(\vec{x}), \quad \phi_j(\vec{x}) = h\left(\sum_{k=0}^{n} w_{jk}^{(1)} x_k\right) \equiv h\left(v_j(\vec{x})\right)$$

Here we defined $\phi_0 = x_0 = 1$ and the sums start from 0 to include the offsets.

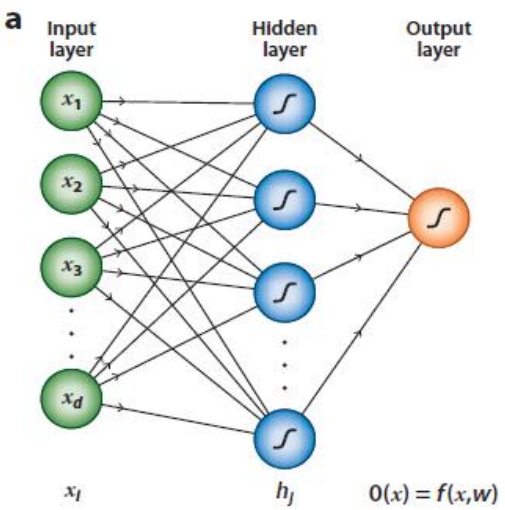
Weights from hidden layer to output:

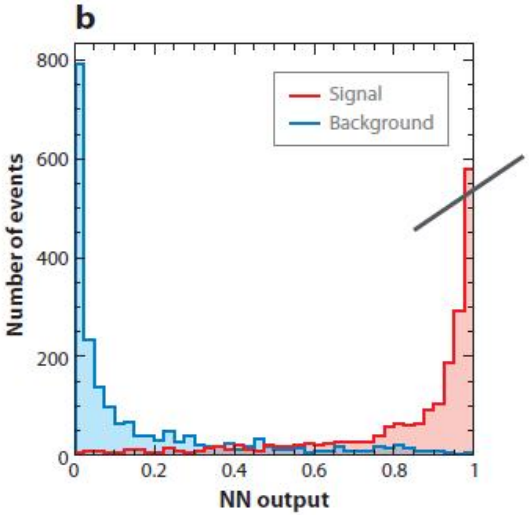
$$E_a = \frac{1}{2}(y_a - t_a)^2 \;\rightarrow\; \frac{\partial E_a}{\partial w_{1j}^{(2)}} = (y_a - t_a) h'(u(\vec{x}_a)) \frac{\partial u}{\partial w_{1j}^{(2)}} = (y_a - t_a) h'(u(\vec{x}_a)) \phi_j(\vec{x}_a)$$

Weights from input layer to hidden layer (→ further application of chain rule):

$$\frac{\partial E_a}{\partial w_{jk}^{(1)}} = (y_a - t_a) h'(u(\vec{x}_a)) w_{1j}^{(2)} h'\left(v_j(\vec{x}_a)\right) x_{a,k} \qquad \vec{x}_a \equiv (x_{a,1}, \ldots, x_{a,n})$$

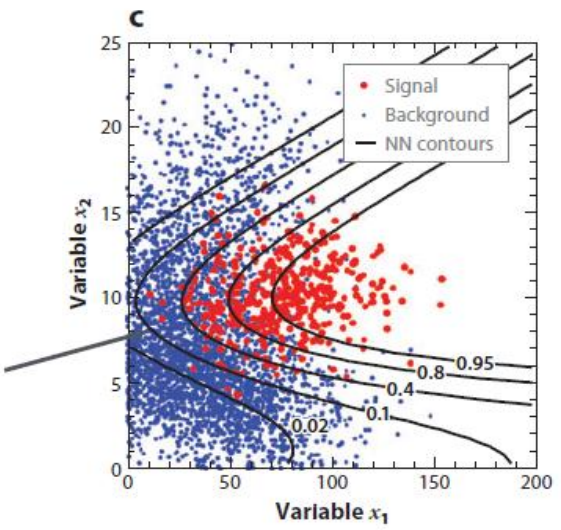
Neural Network Output and Decision Boundaries

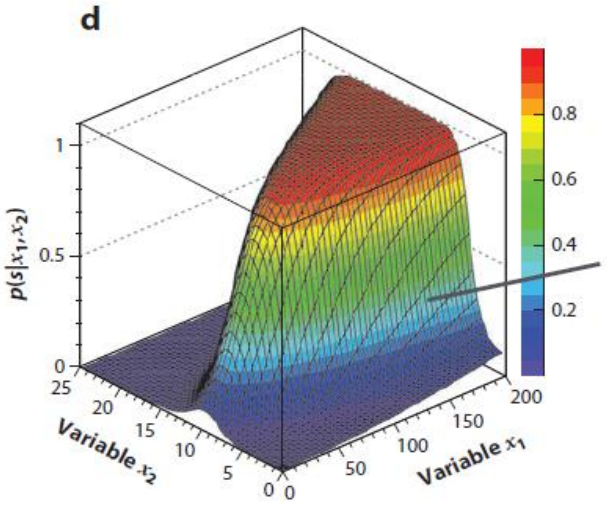




output of neural network

decision boundaries for different cuts on NN output

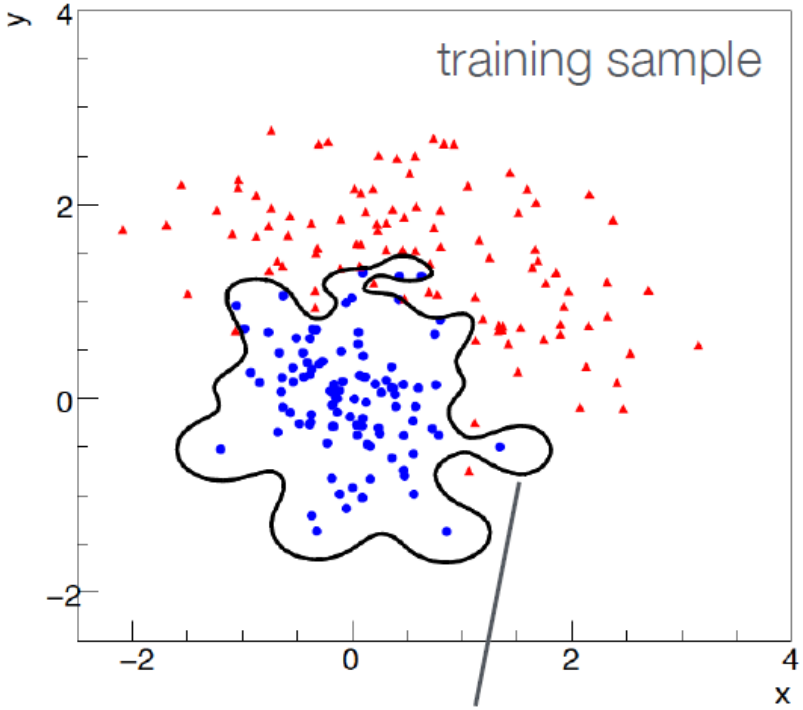


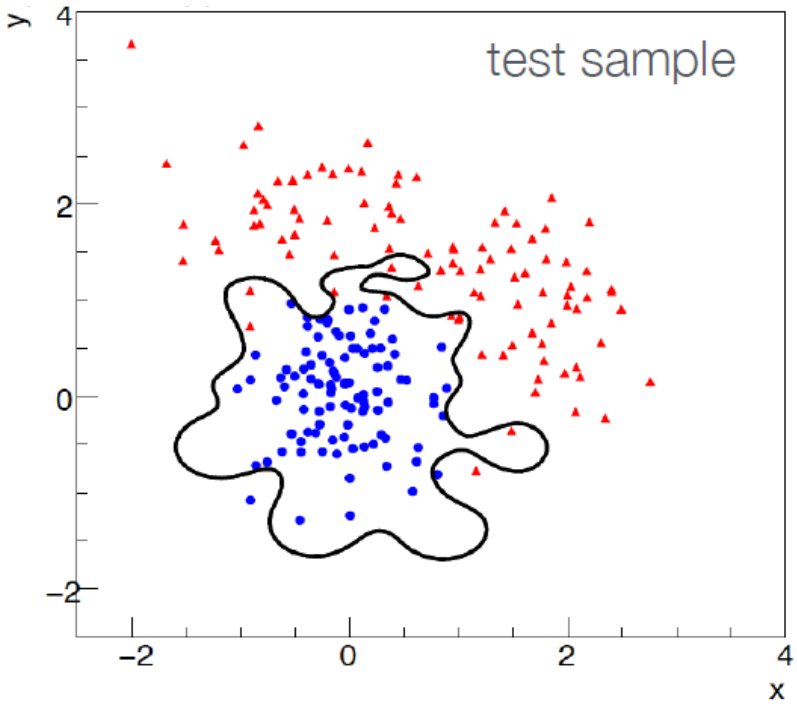


signal probability $p(s \mid x_1, x_2)$

Example of Overtraining

Too many neurons/layers make a neural network too flexible
→ overtraining

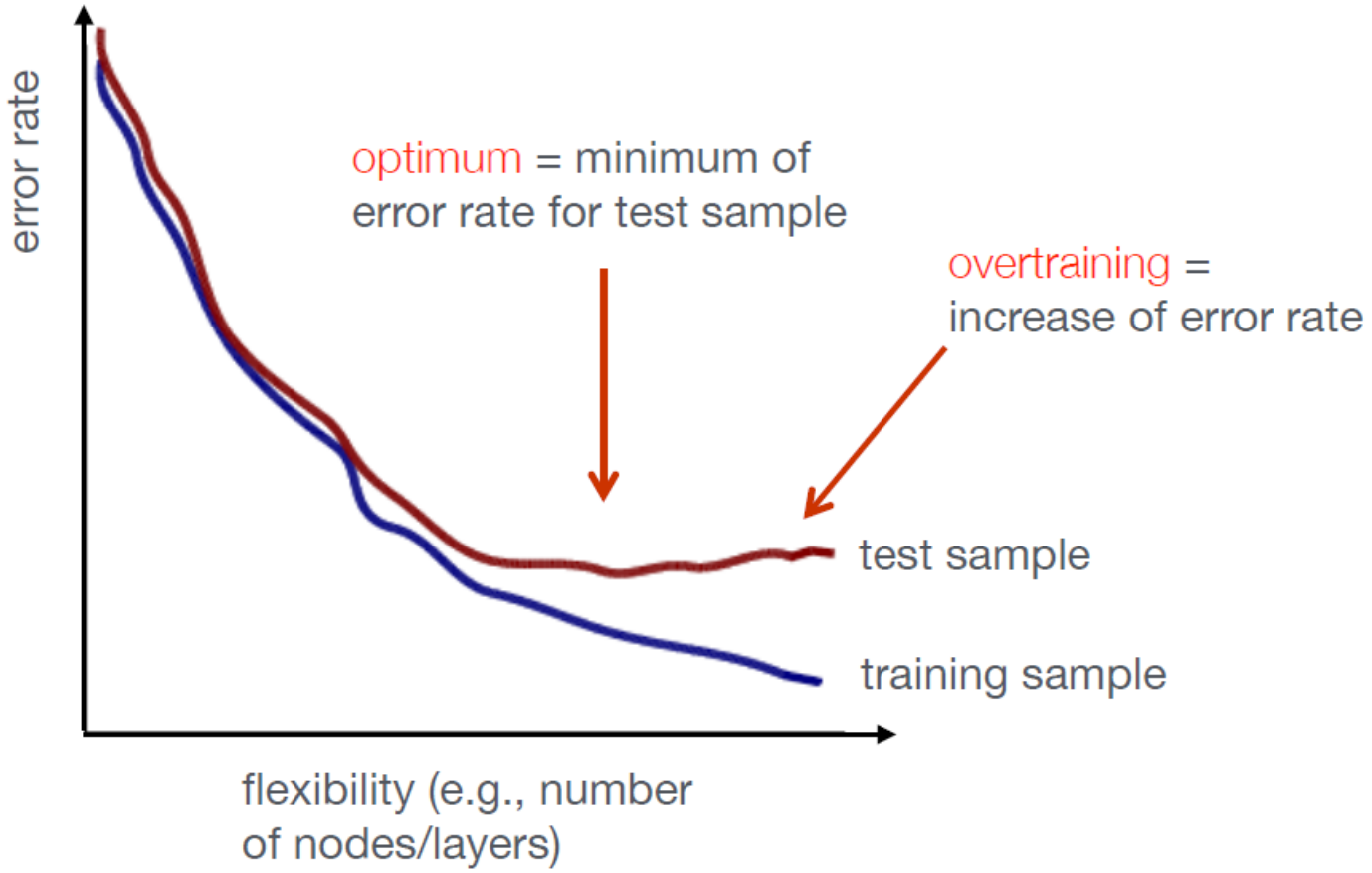




Network "learns" features that are merely statistical fluctuations in the training sample

Monitoring Overtraining

Monitor fraction of misclassified events (or error function:)



Deep Neural Networks

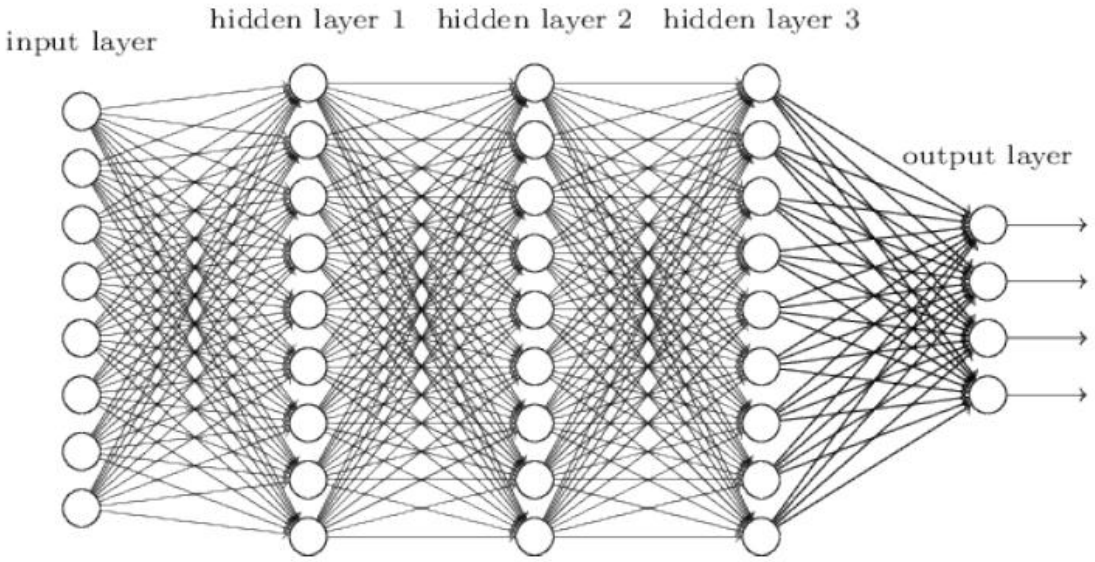
Deep networks: many hidden layers with large number of neurons

Challenges

- Hard too train ("vanishing gradient problem")
- Training slow
- Risk of overtraining

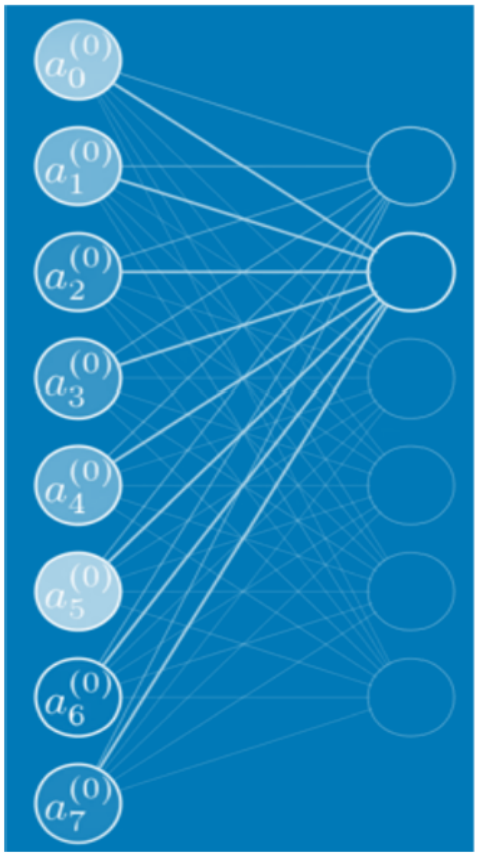
Big progress in recent years

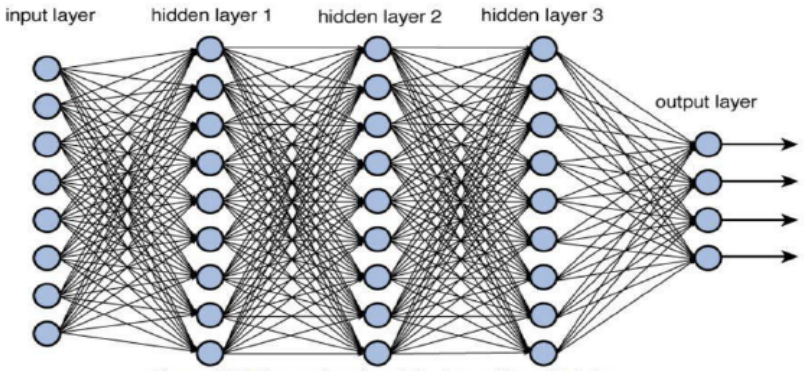
- Interest in NN waned before ca. 2006
- Milestone: paper by G. Hinton (2006): "learning for deep belief nets"
- Image recognition, AlphaGo, …
- Soon: self-driving cars, …



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?

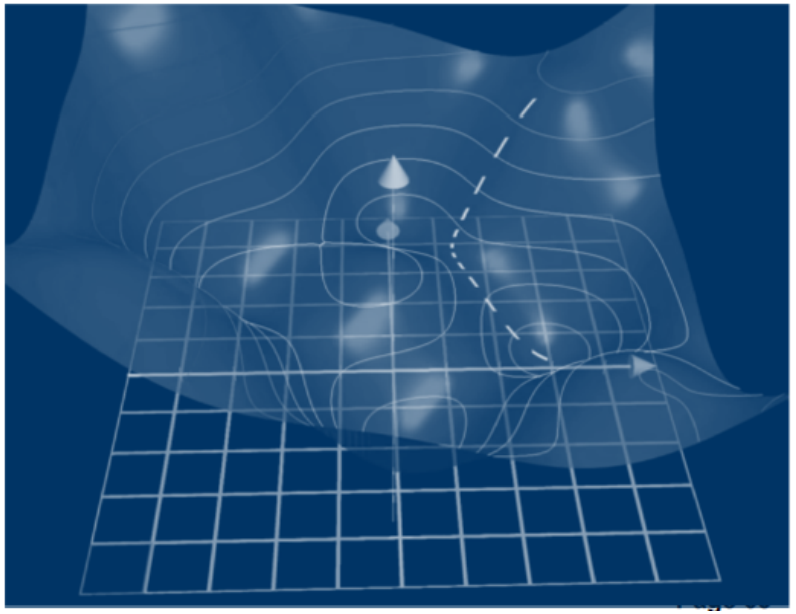
After we constructed a network, our task is to **assign proper weights** so neurons will react correctly to incoming signals.

- 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.

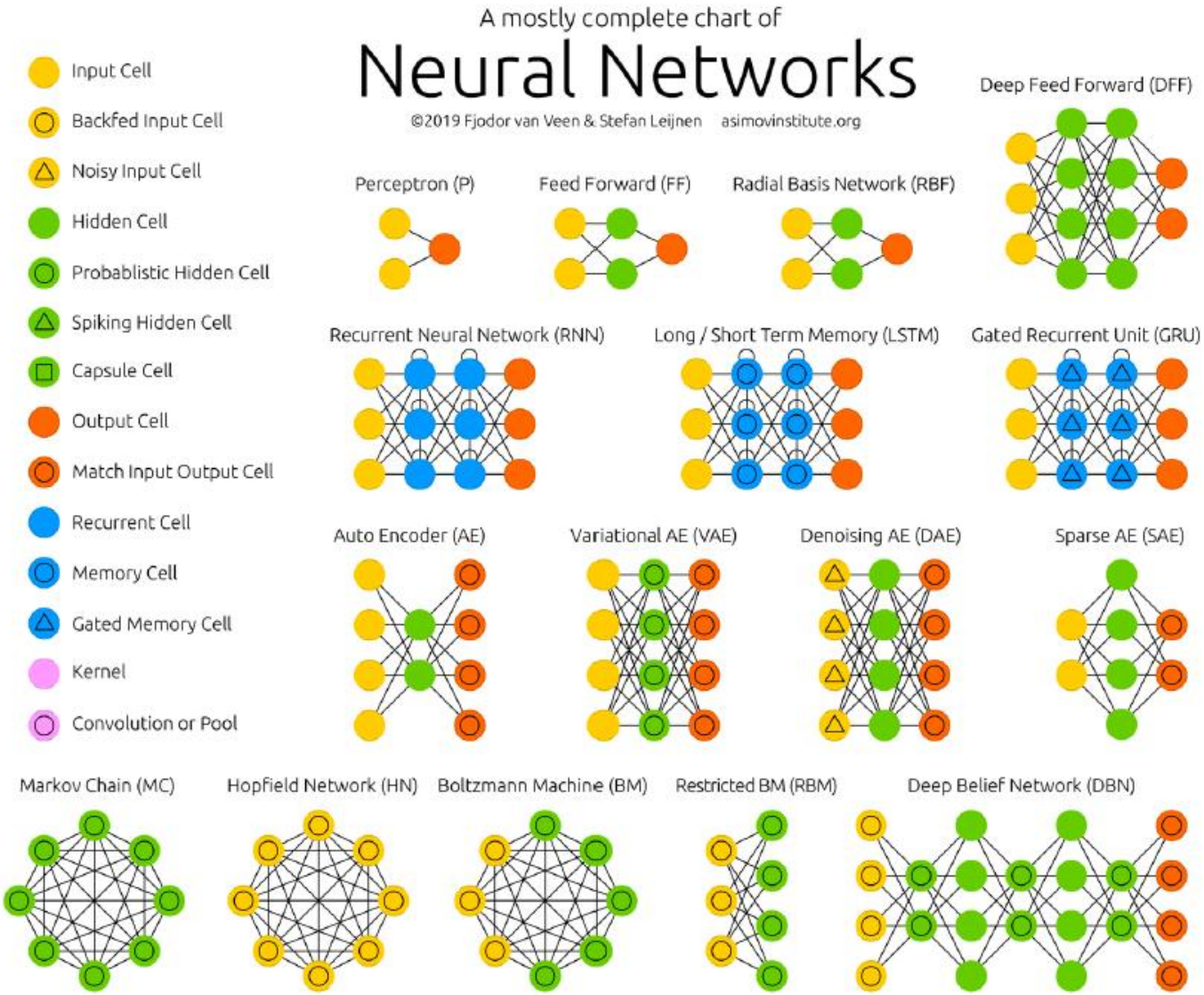
- To start with, all weights are assigned randomly.
- After evaluating the NN on the training dataset, we can compute all the per-neuron differences with respect to the correct result.
- Computing the gradient of the loss, gives us a direction in which to tune the weights towards a local minimum

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



DES

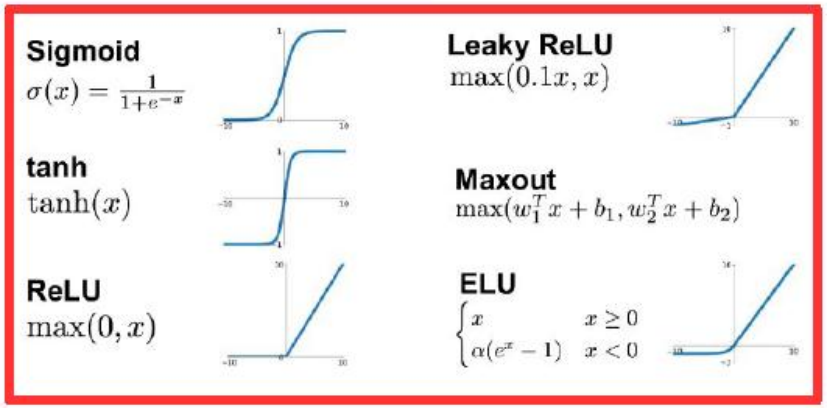
How do NNs learn?

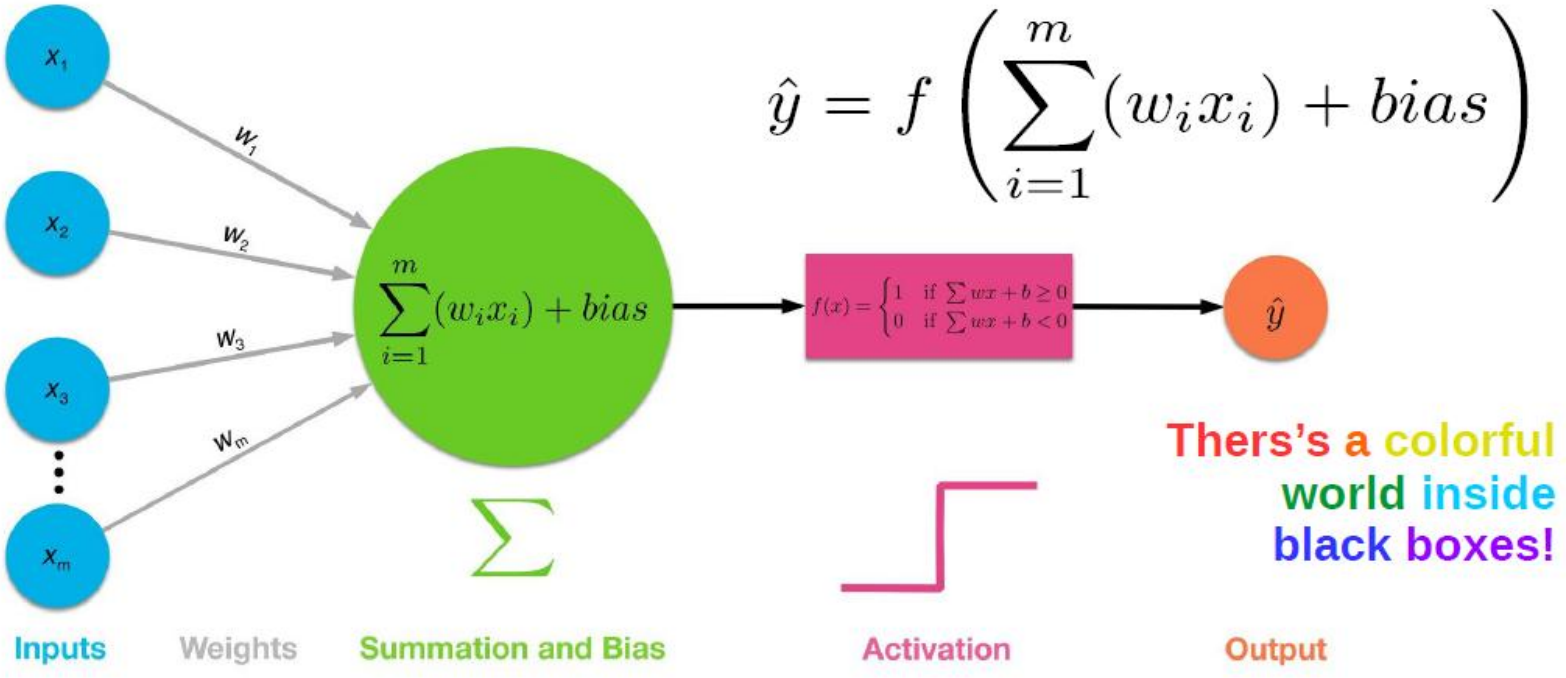


There are many more…

How do NNs learn?

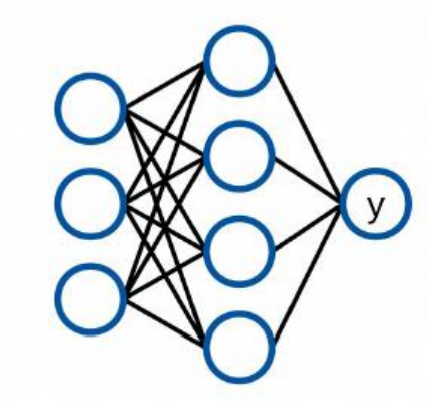
- Each input is multiplied by a weight.
- The weighted values are summed and a bias is added
- **The result is passed to an activation funciton (non liniarity).**

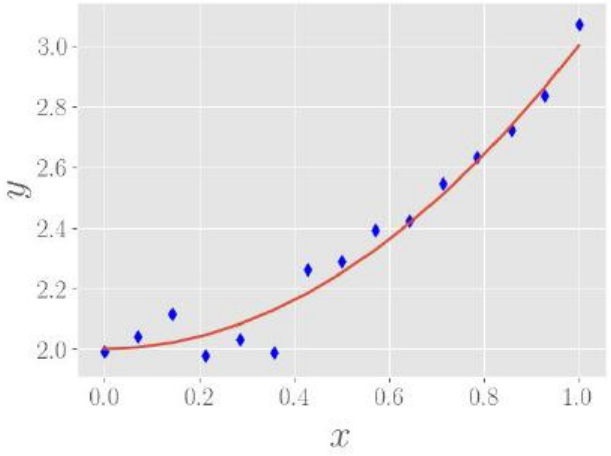




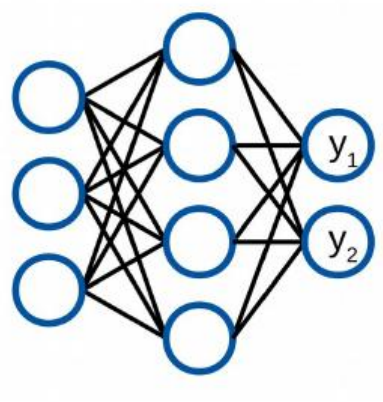
Typical Applications

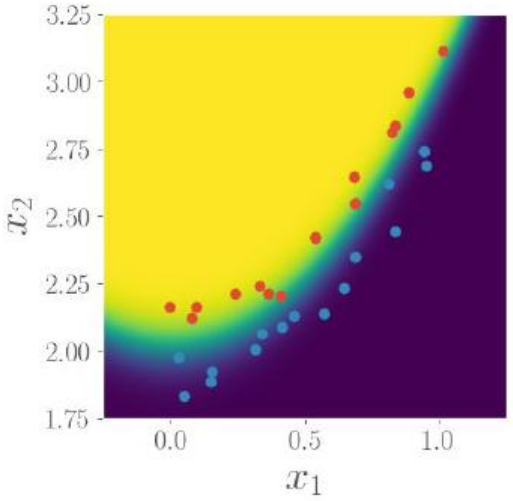
Regression:
Predict a continuous label.





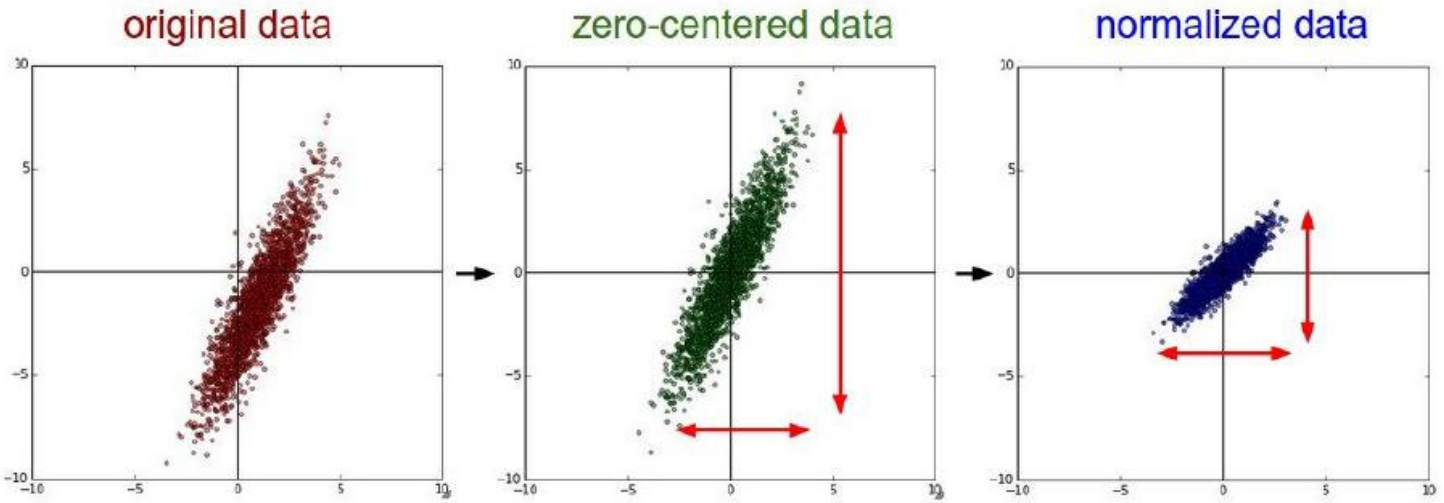
Classification:
Separate events into multiple categories.





Input Preprocesing

- Input features could have vastly different scales (e.g. p_T vs η).
  - Difficult to find optimal values.
- Basic strategy: Normalize to mean=0 and variance=1.

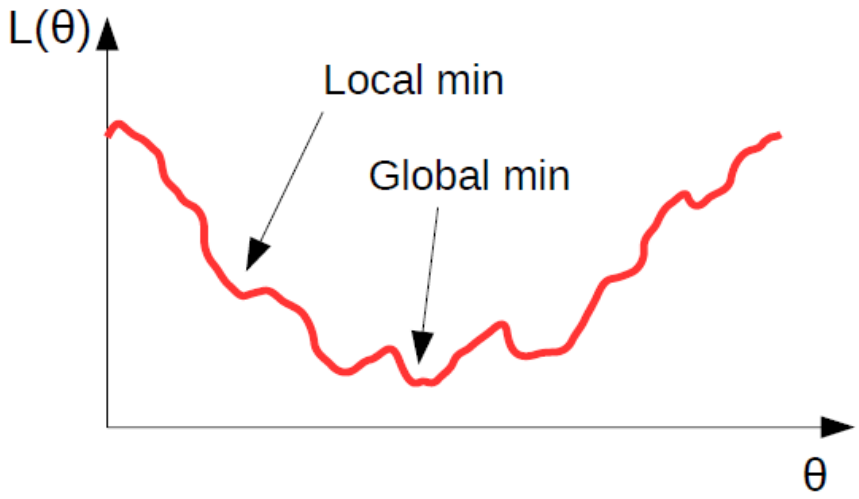


- Other options possible: decorrelation, non-linear transformation, …

Training

- Training starts specifying an input and a target dataset.
  - For each input set, the target is what the network should learn for that input.
- A loss function is required L(θ):
  - The loss funtion quantifies the mistakes the NN makes. E.g. *mean squared error* can be used for regression.

Training is the minimization of the loss function w.r.t. the NN parameters.



Training: (Stochastic) Gradient Descent

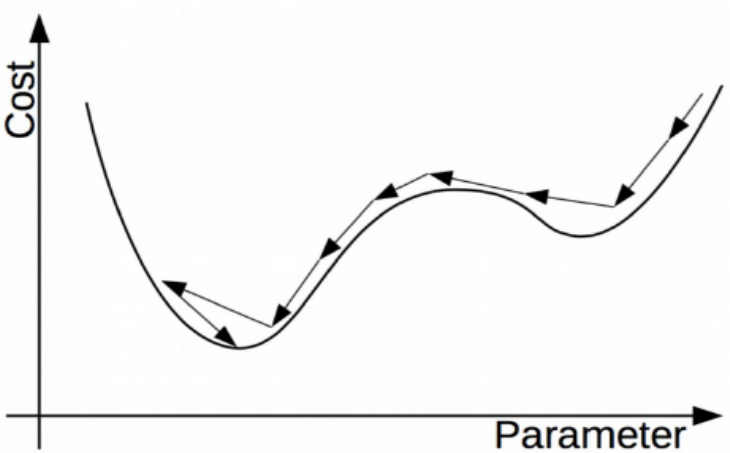
- Given the increasing size of datasets and parameters, it is no more possible to directly minimize the loss function.

Iterative minimization by updating θ in opposite direction of gradient.

$$\theta_i = \theta_{i-1} - \alpha \frac{\partial L}{\partial \theta}$$, where α is the so-called « learning rate ».

- Evaluation and derivation of the loss funciton for the full dataset is costly:

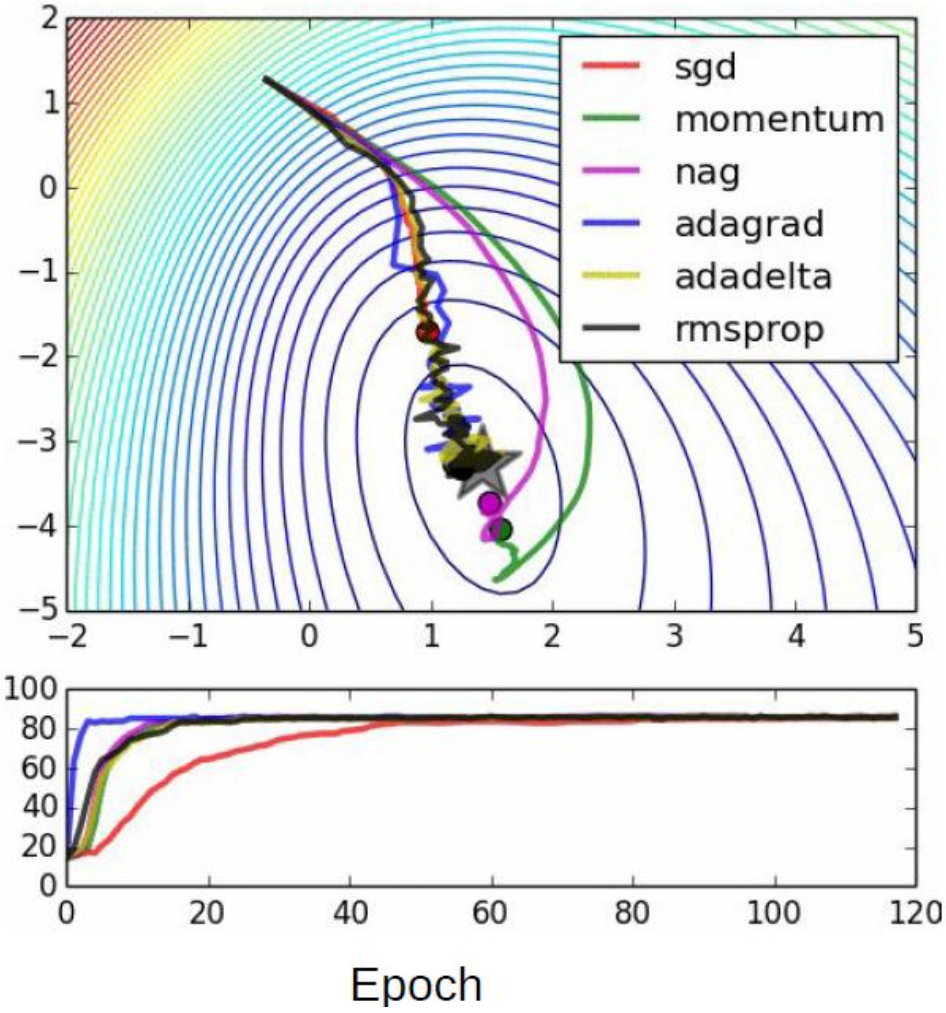
Stochastic gradient descent:
Calculate gradient for a small stochastic subset of the training sample (batch). → This also helps to avoid local minima!



One iteration over the full training dataset is called *epoch*.

Training: more optimisers

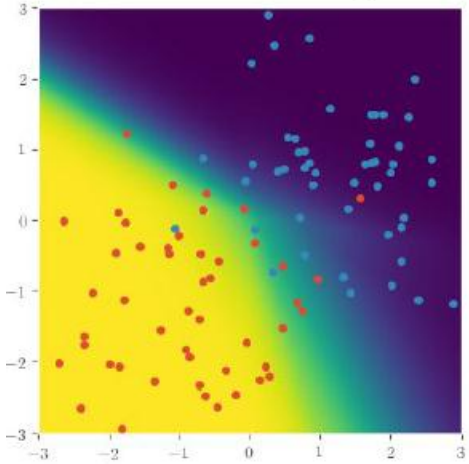
- More advanced options than fixed learning rate.
  - Momentum: past gradients used as « velocity »
  - Adaptive methods: different learning rates for each parameter and as a function of past gradients.

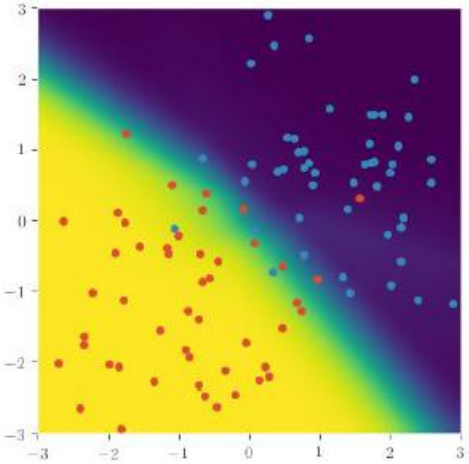


Underfitting and overtraining

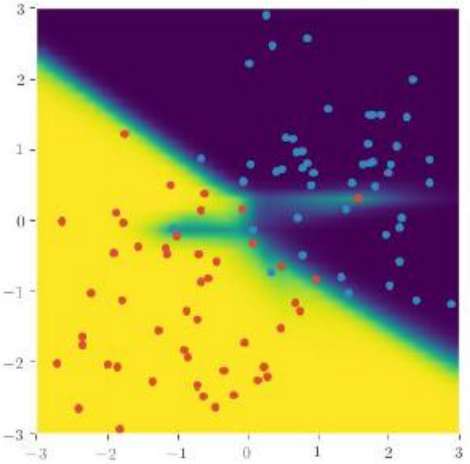
Underfitting: If model capacity is too low or if training is not enough
→ bad performance.

Underfitting





Overfitting

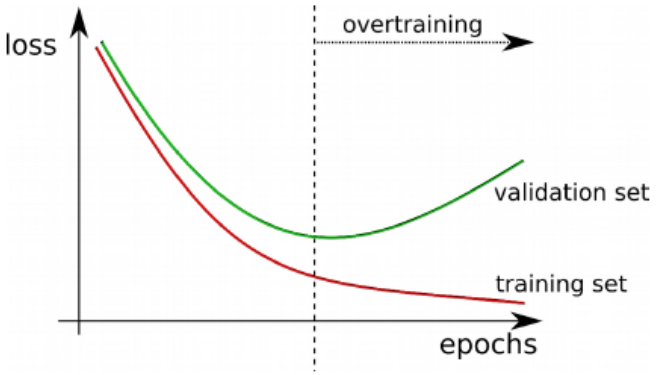


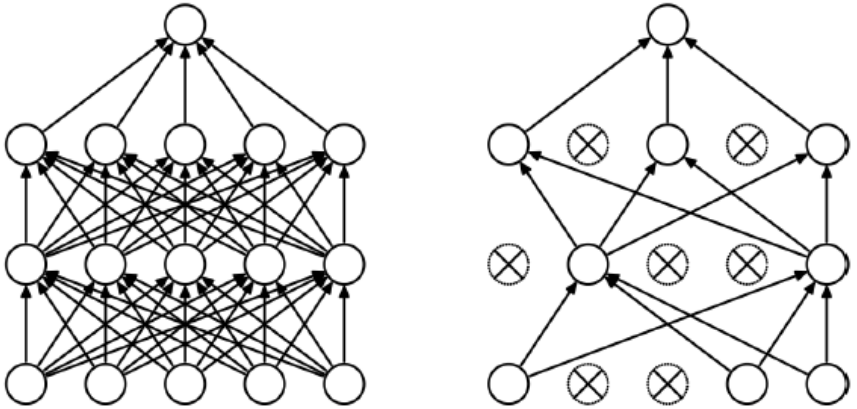
Overfitting: If model capacity is too high, network can « memorize » training samples
→ bad generalization.

Overtraining solutions

Early stop:
Evaluate the performance of the network on a validation dataset.
Stop when performance on validation set decreases

Dropout:
Randomly drop a percentage of nodes at each training step.
Learn redundant representations, hence giving a more robust model.





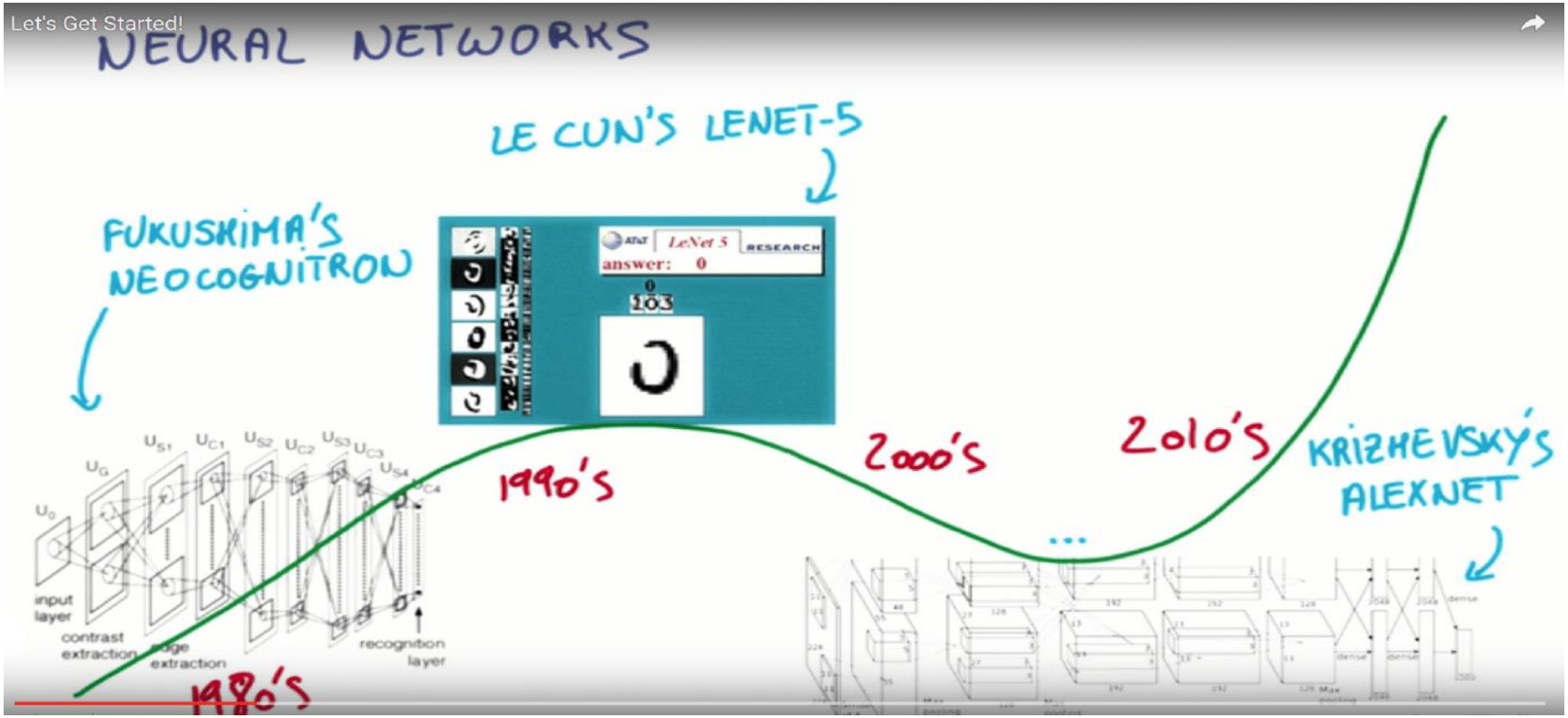
Deep-learning Neural Network

TensorFlow™

MNIST example

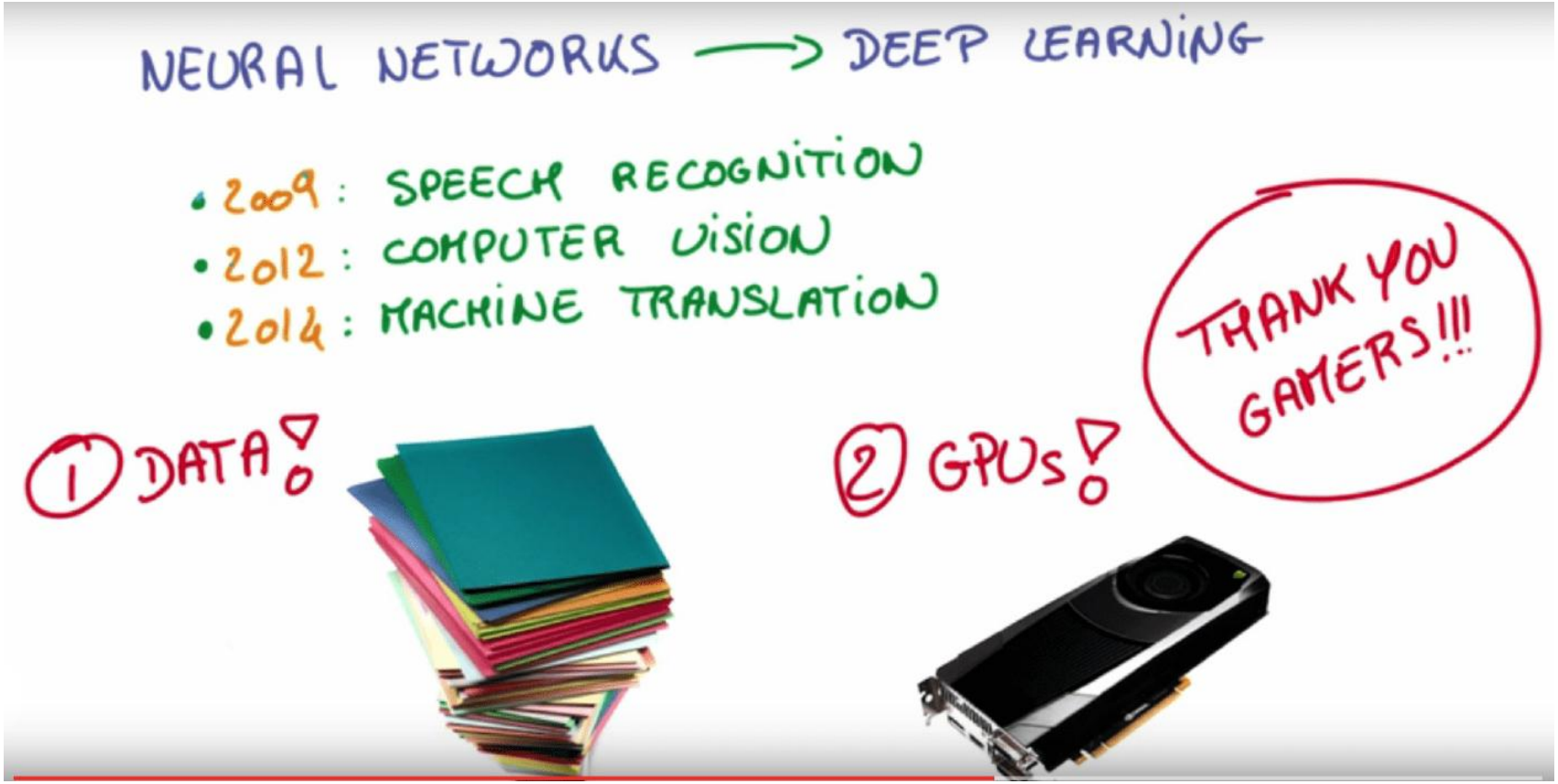
Neural network

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



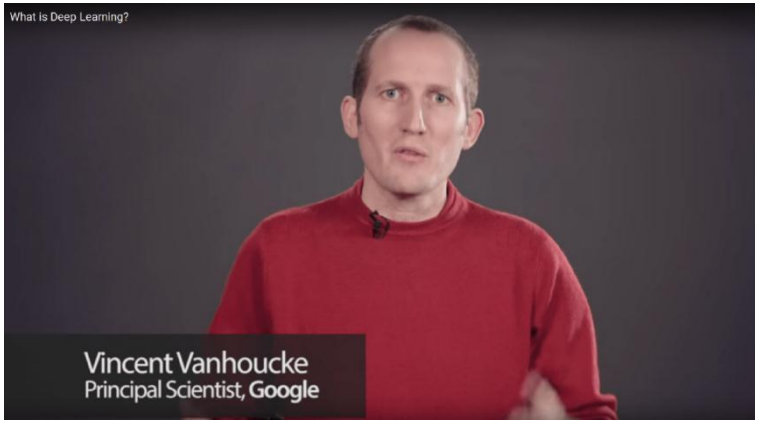
Neural network

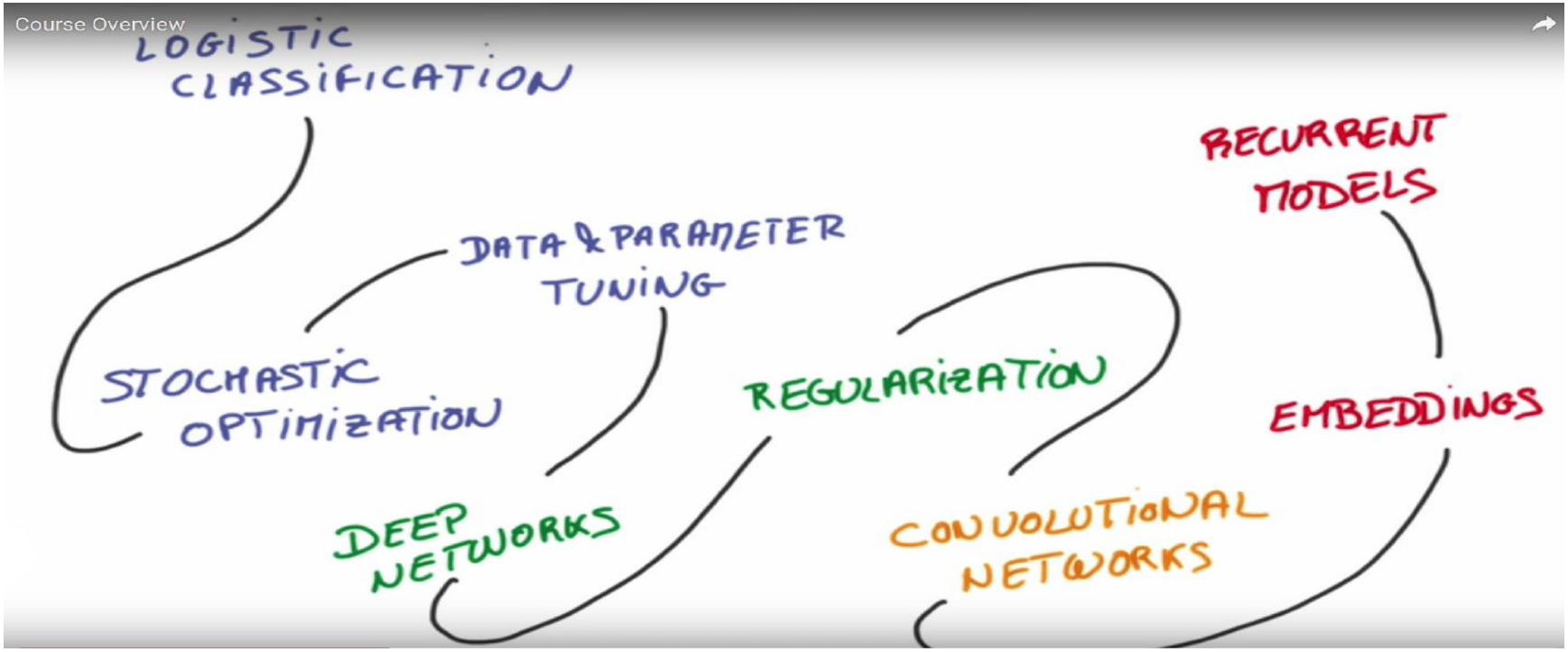
Since 2010 new era: rapidly increasing areas of applications



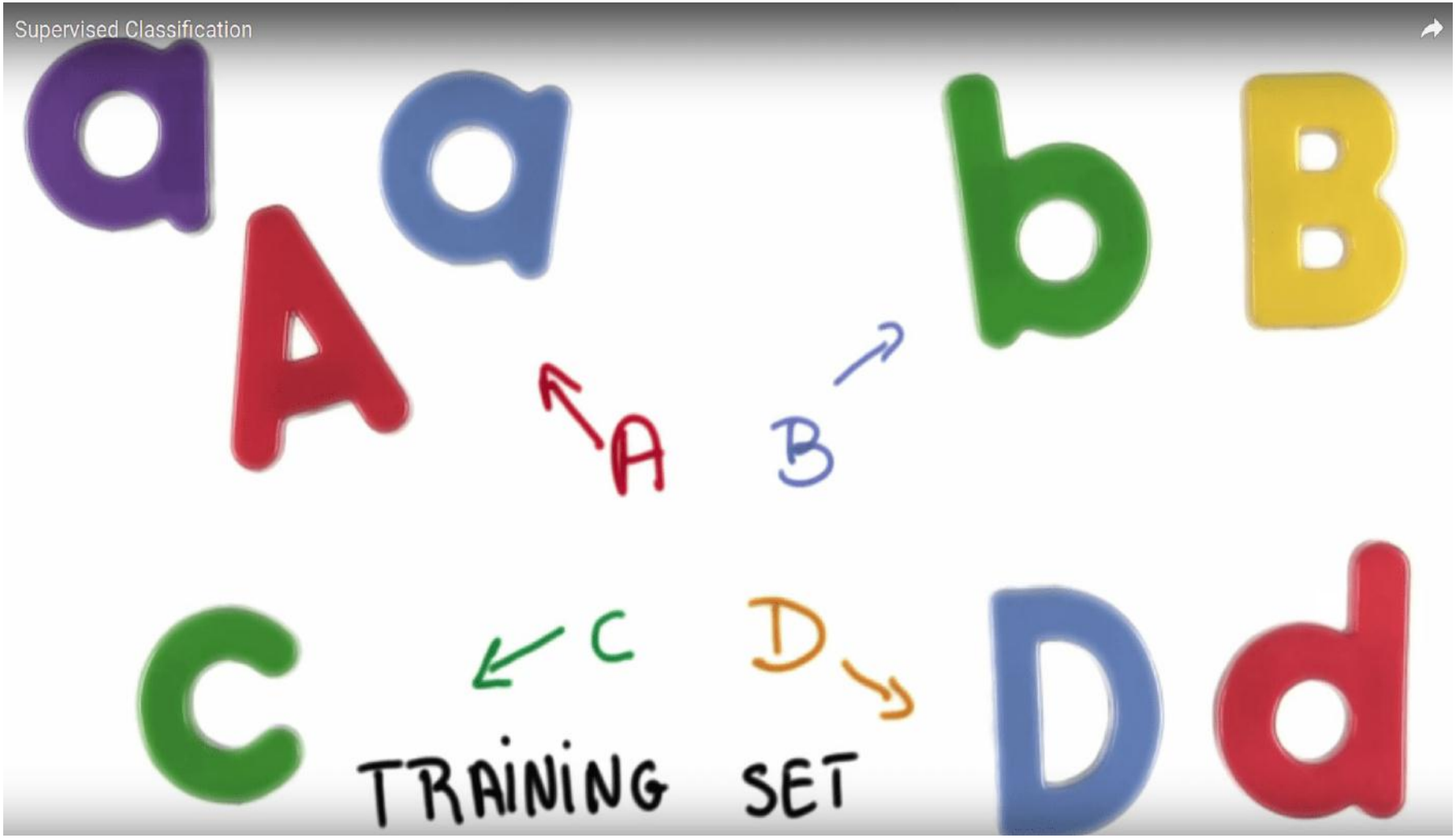
Deep-Learning tutorial @ udacity

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

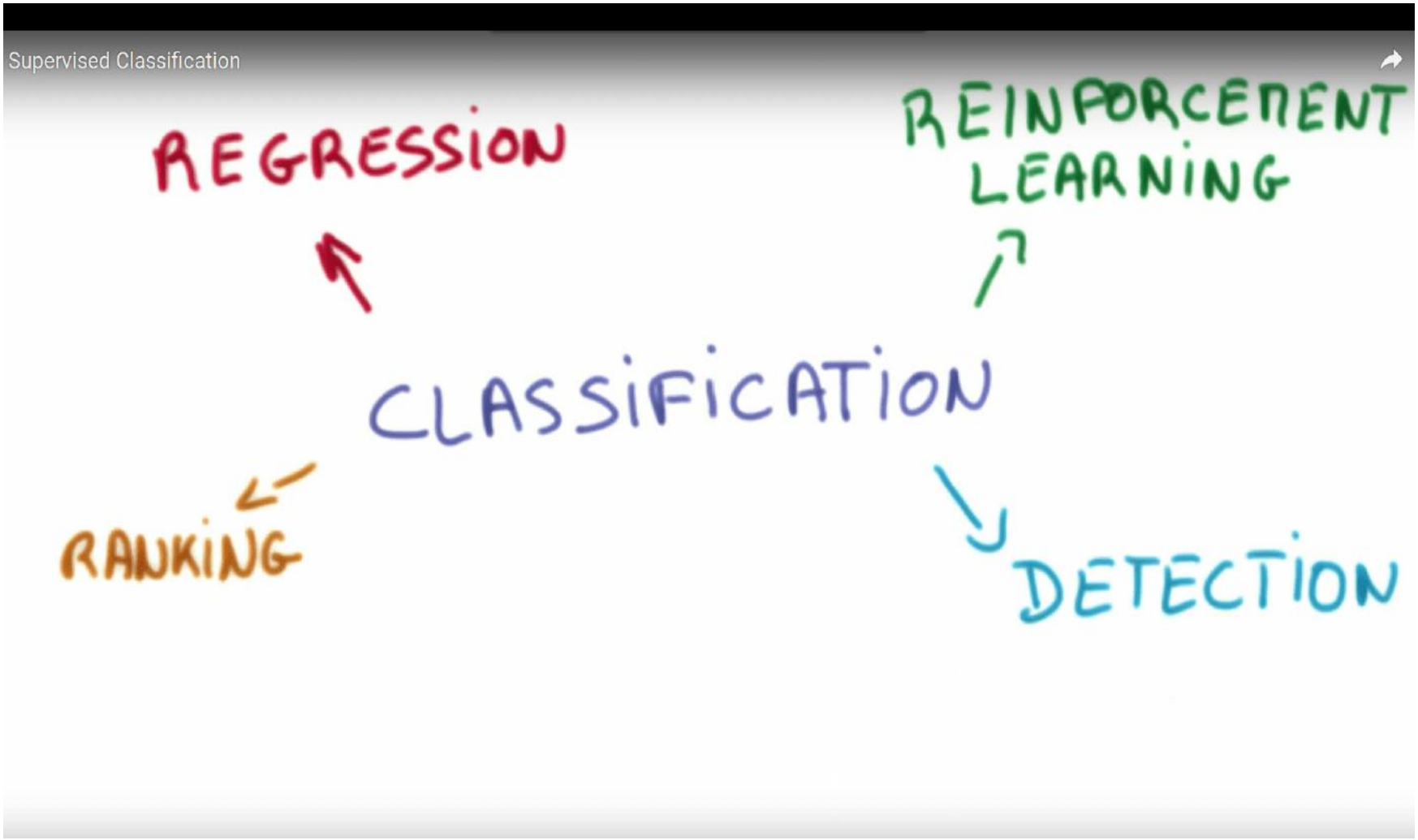




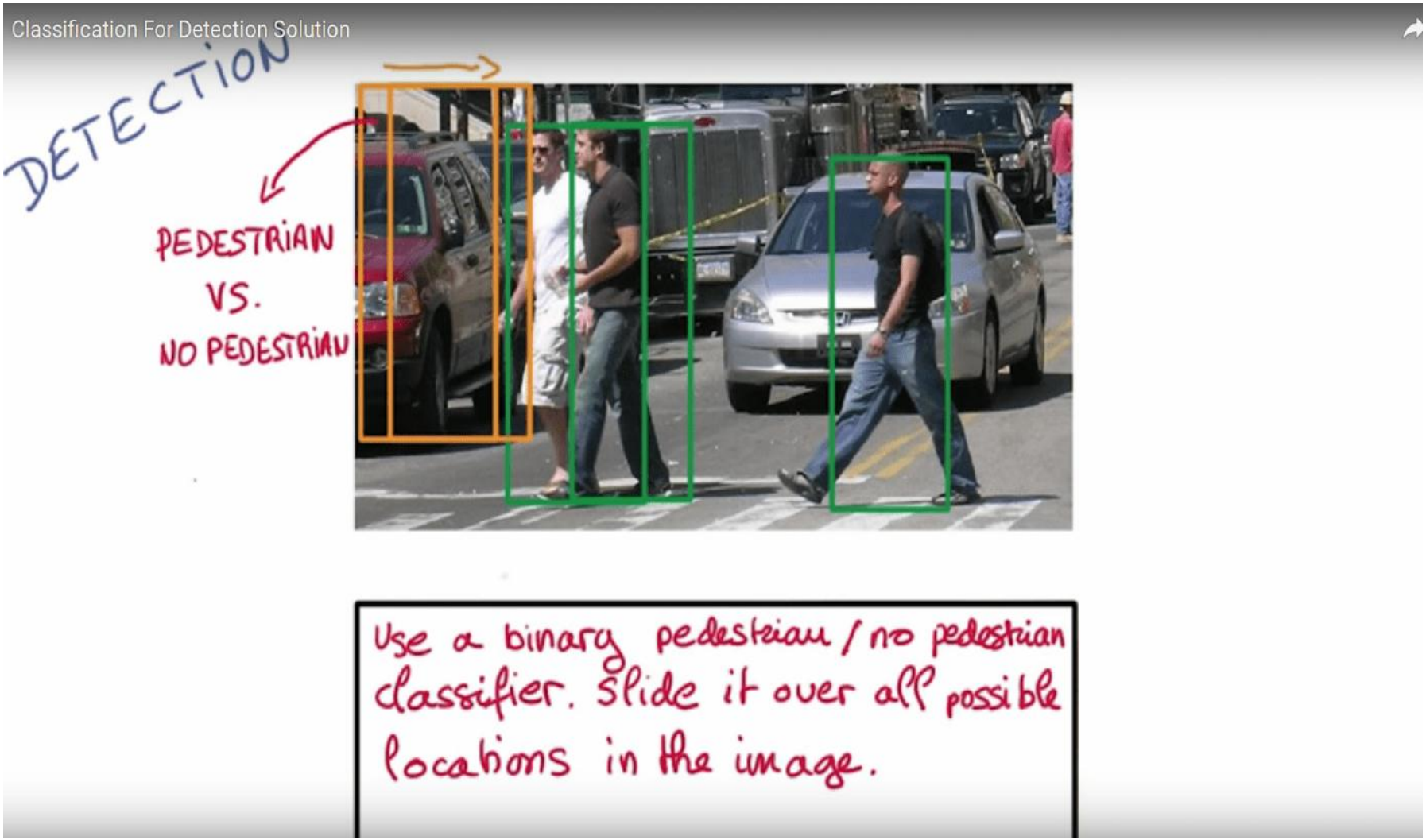
Supervised Classifications



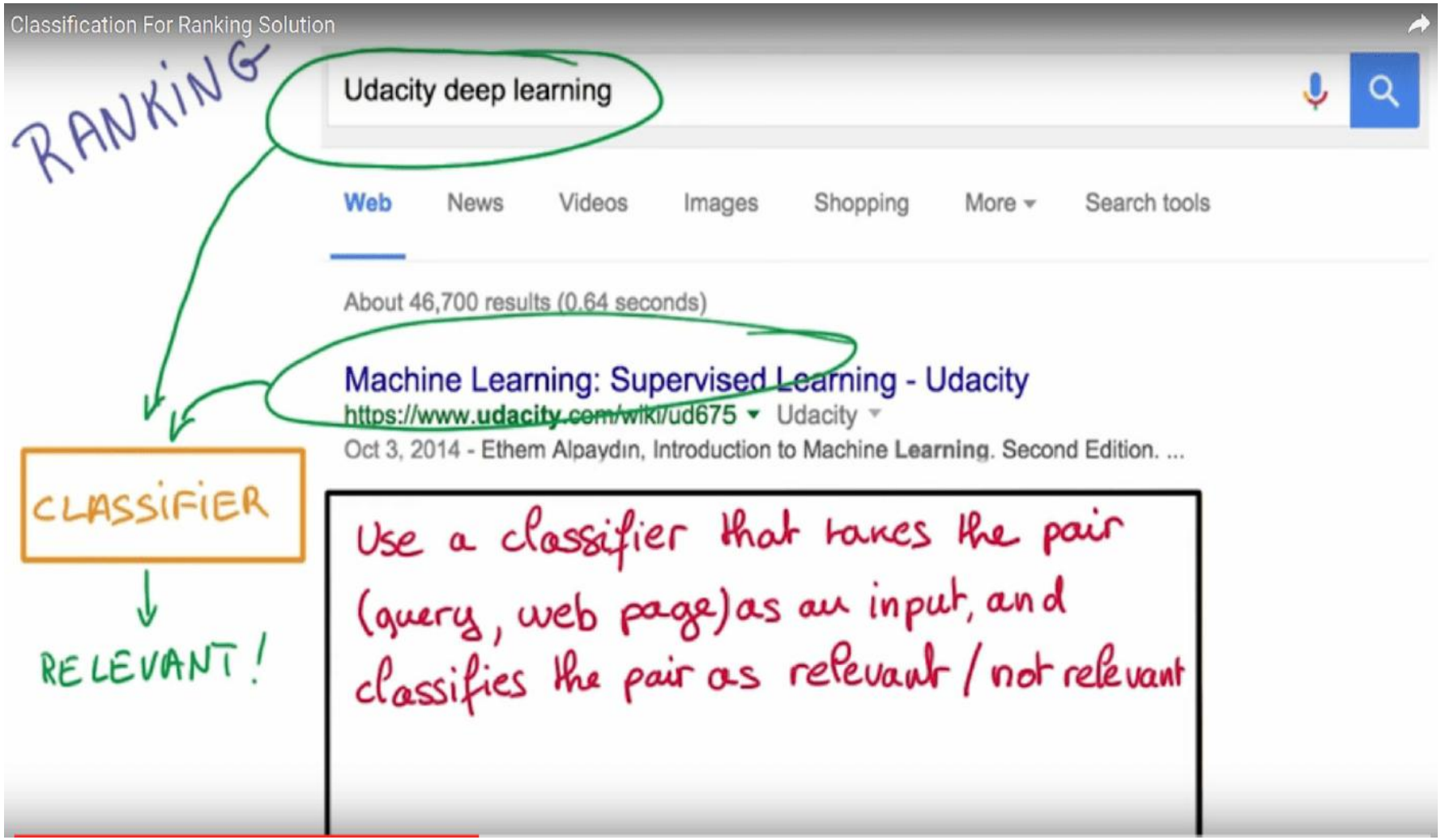
Supervised Classifications



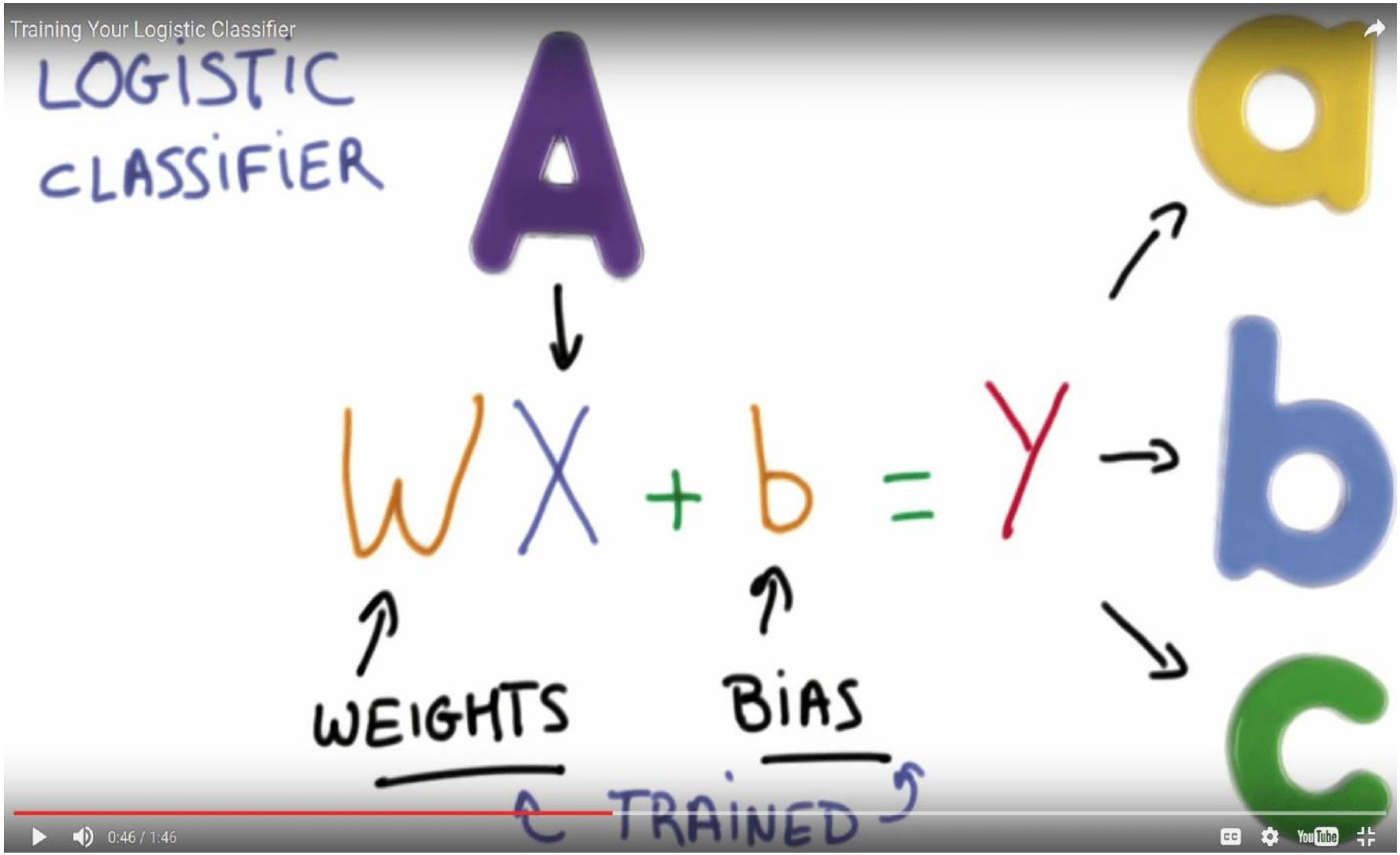
Classifications for Detection



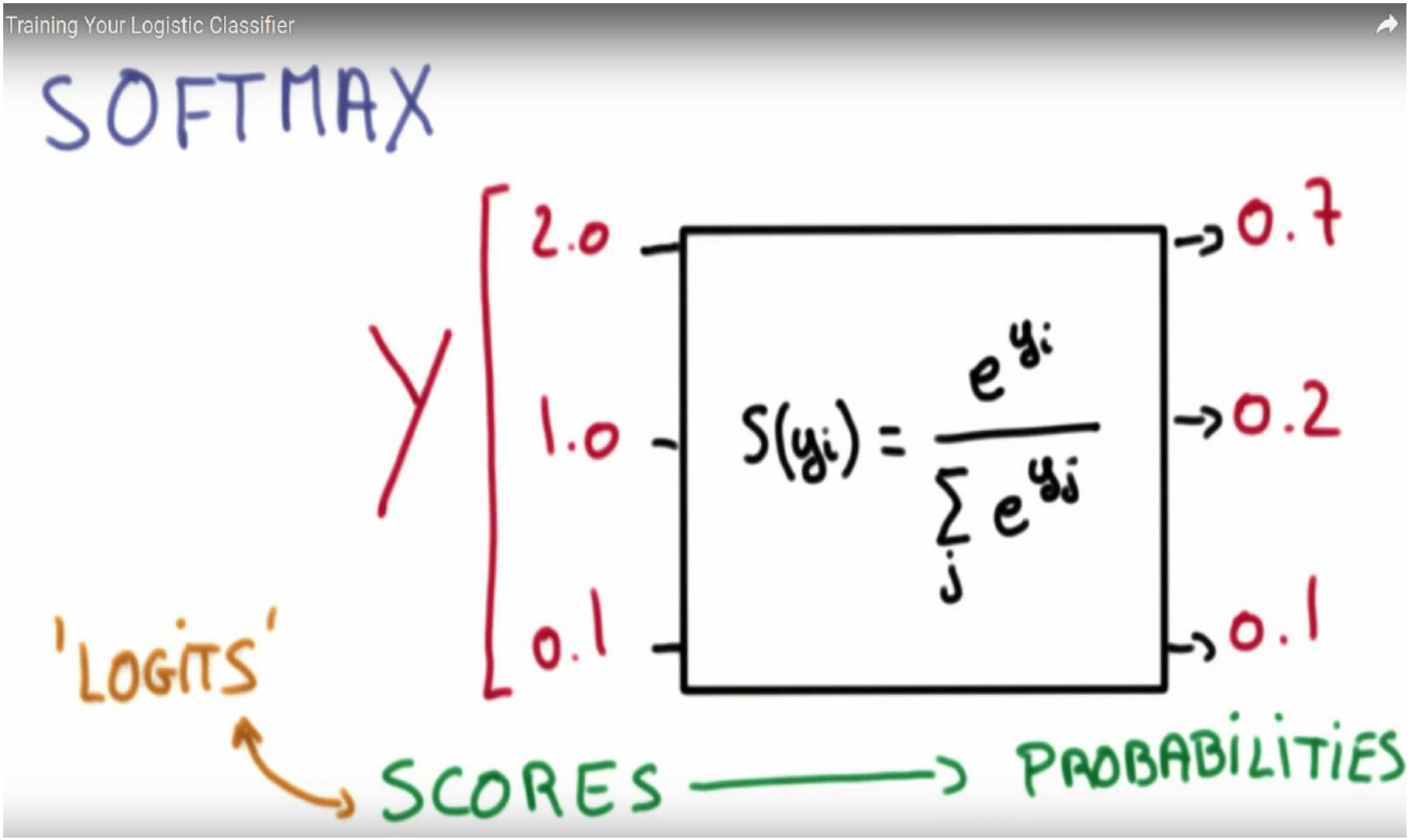
Classifications for Ranking



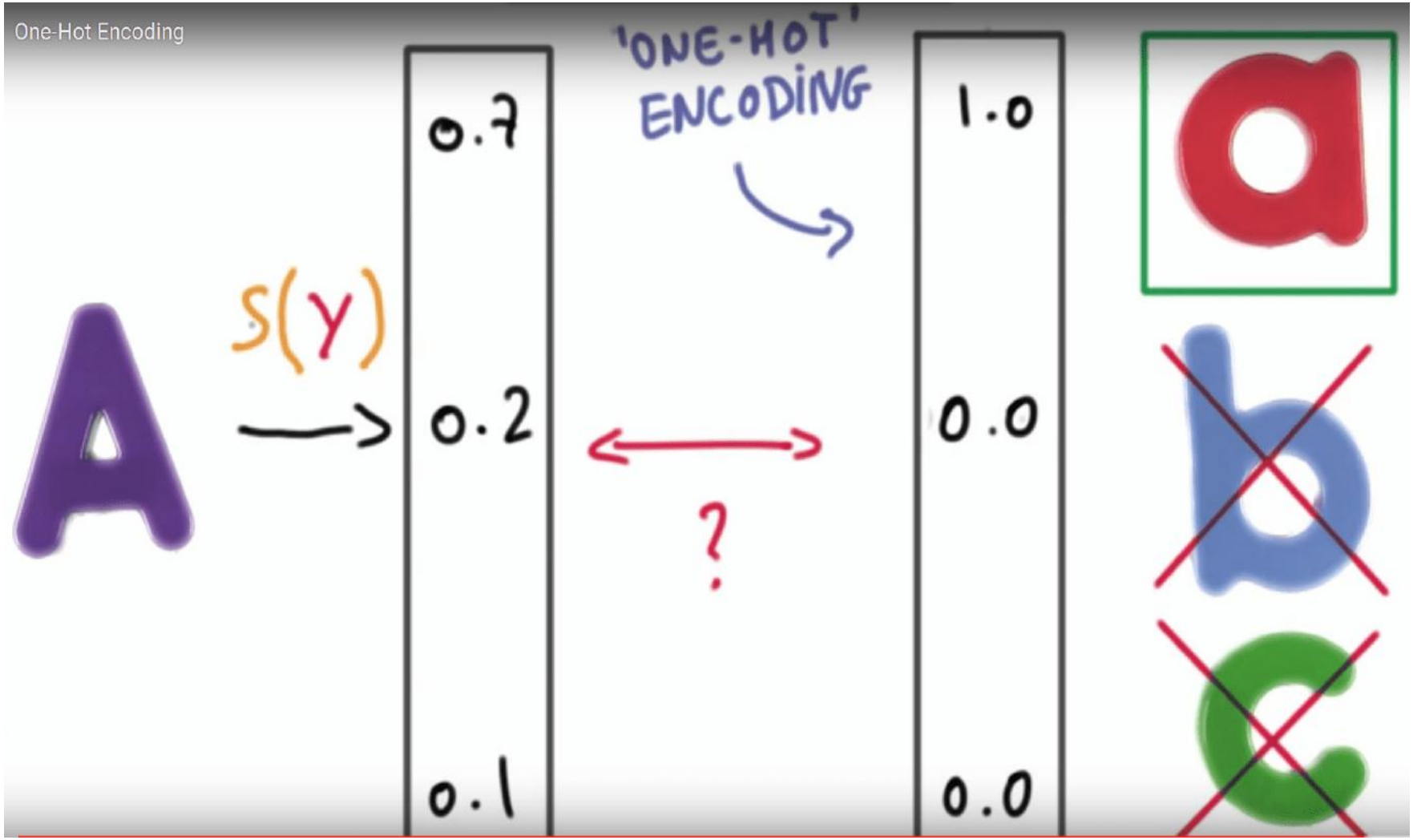
Logistic classifier: Linear model



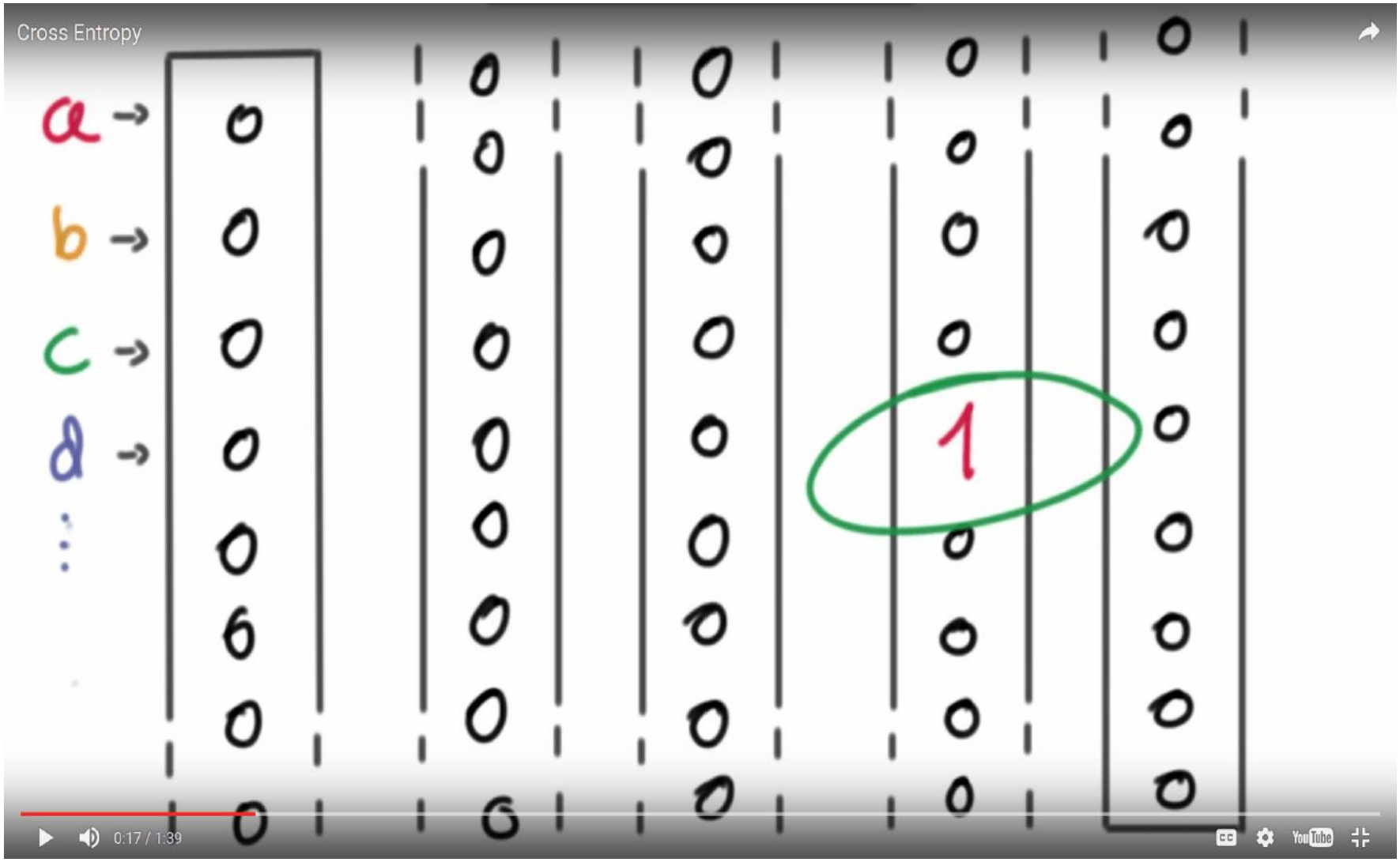
Softmax


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

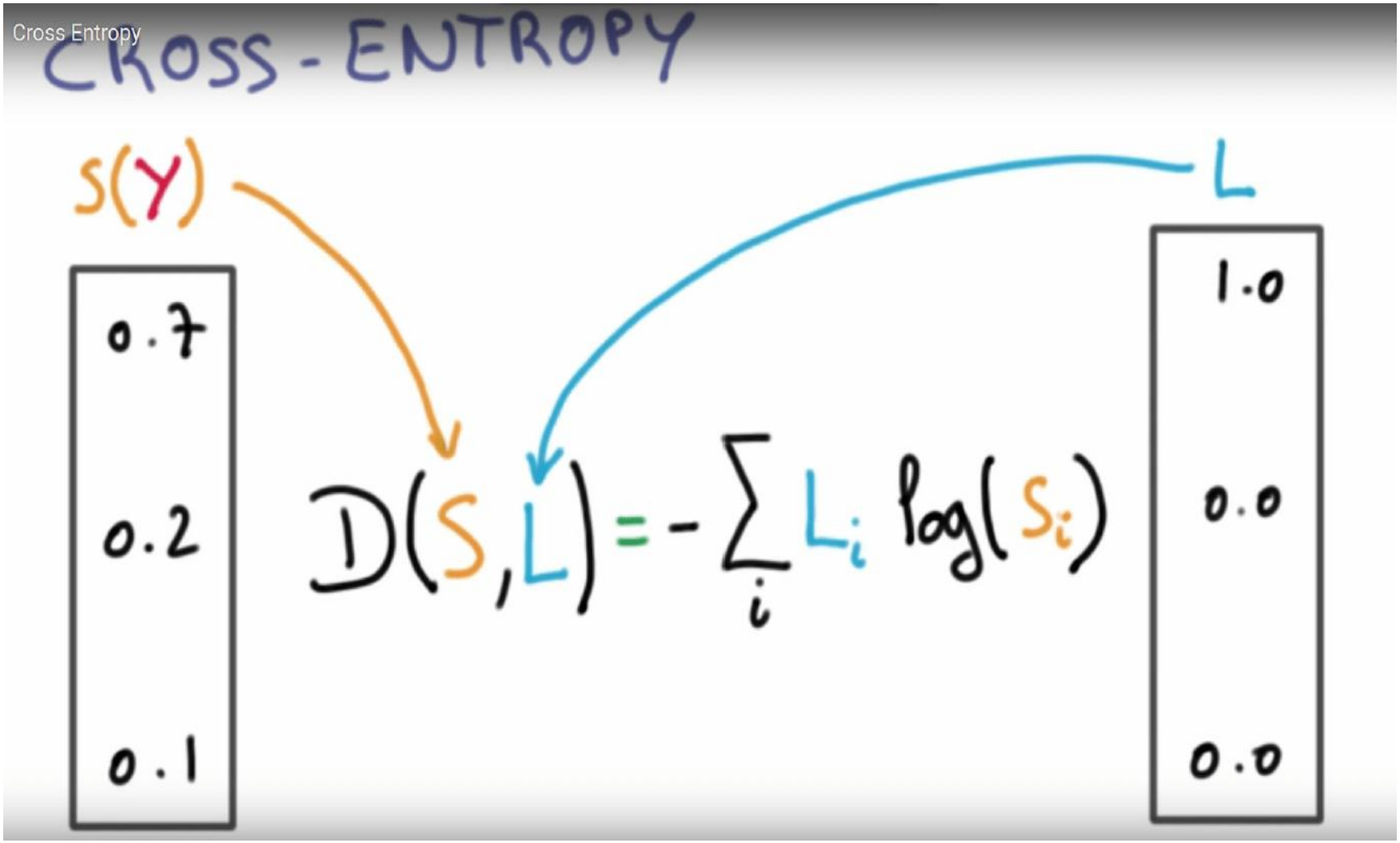
„One hot" encoding



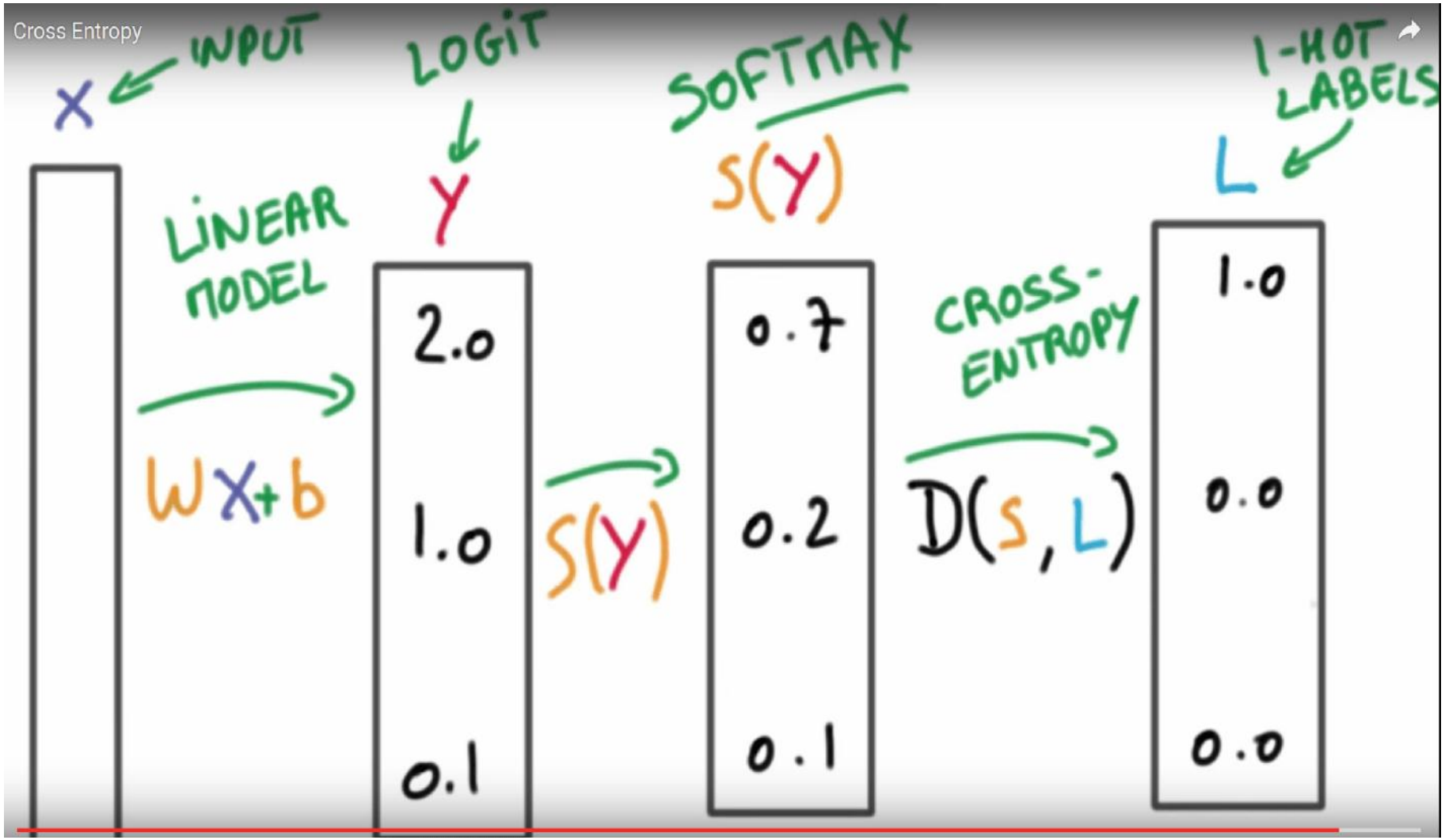
„One hot" encoding



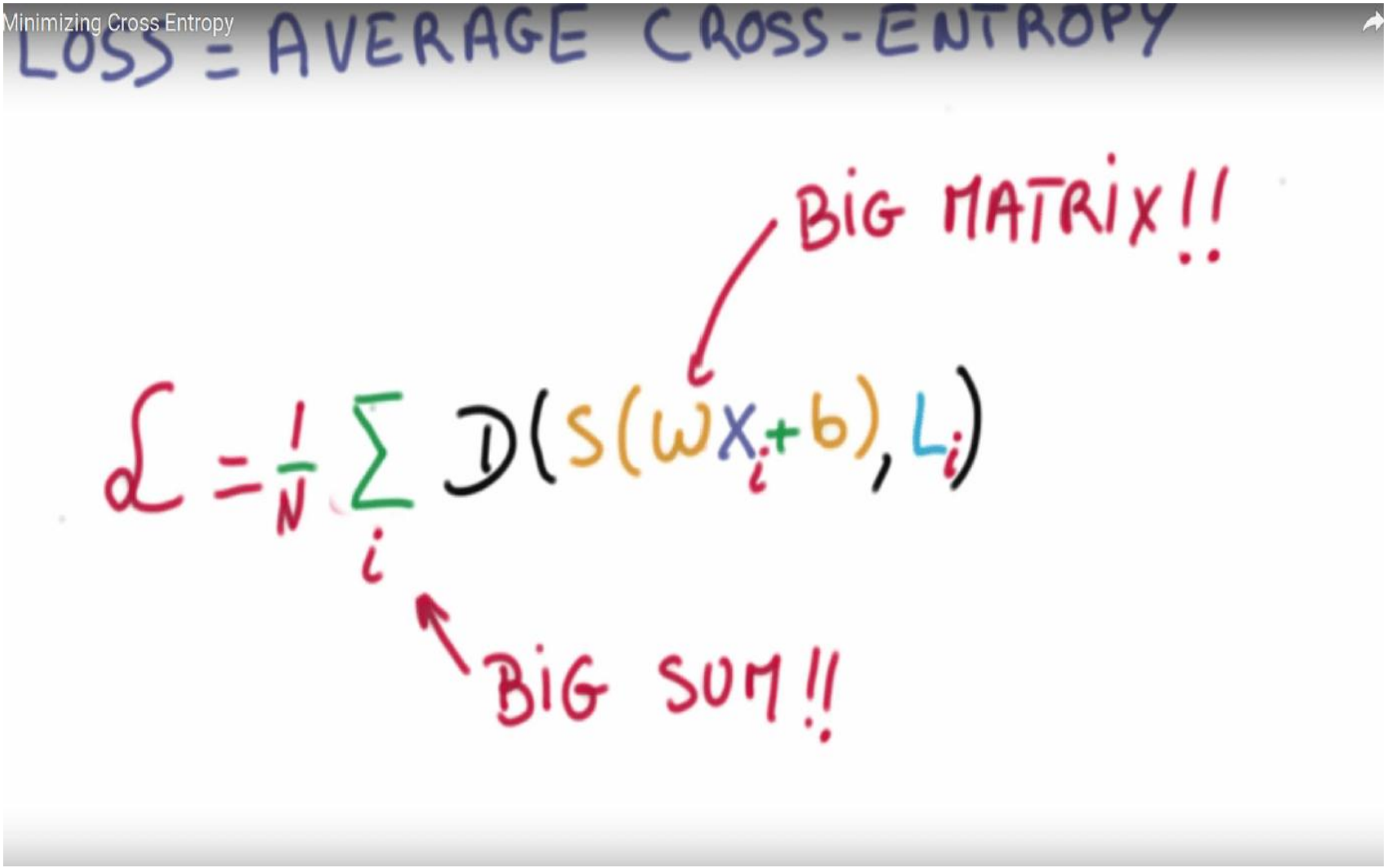
Optimisation: Cross-Entropy



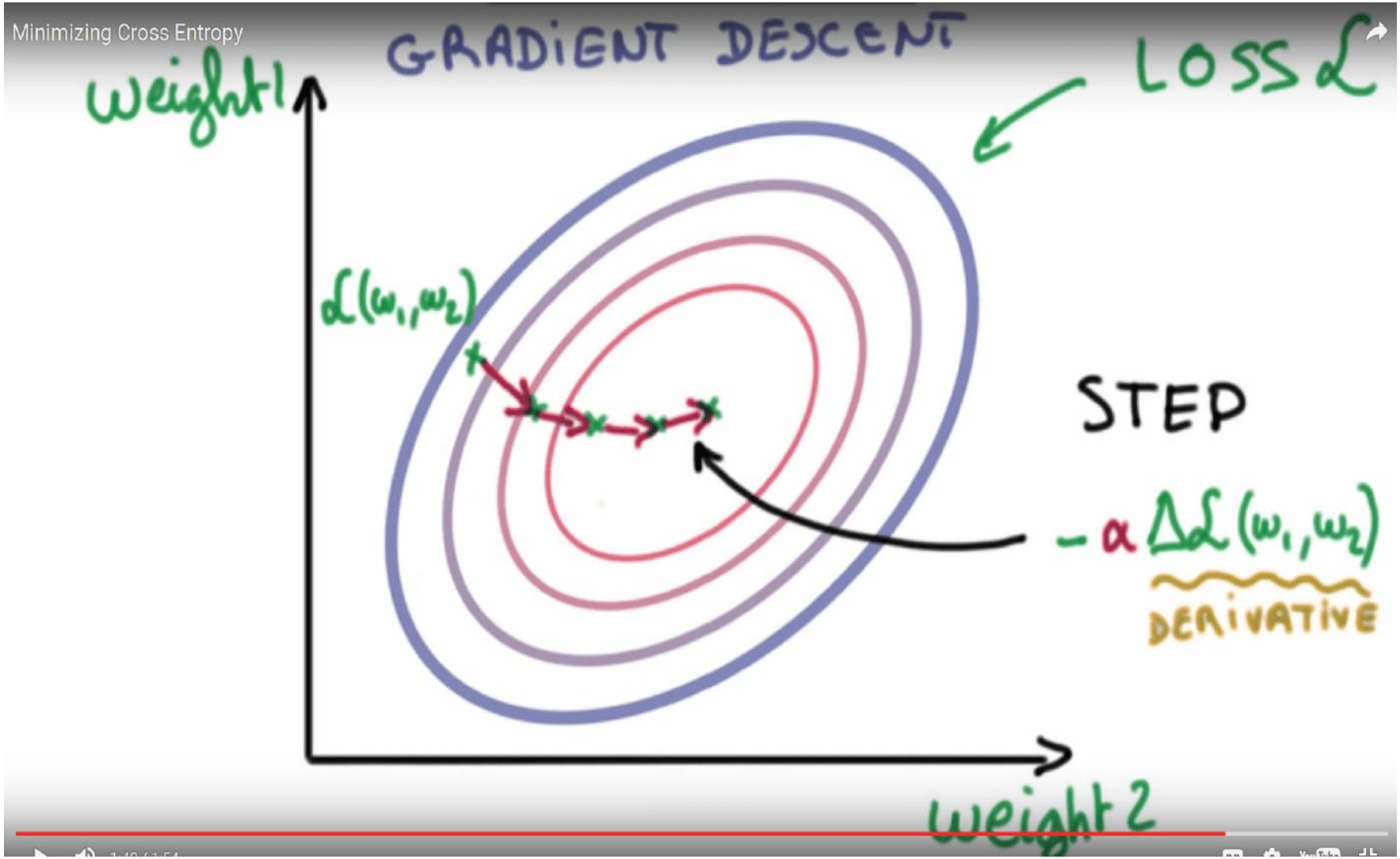
Multinomial logistic classification



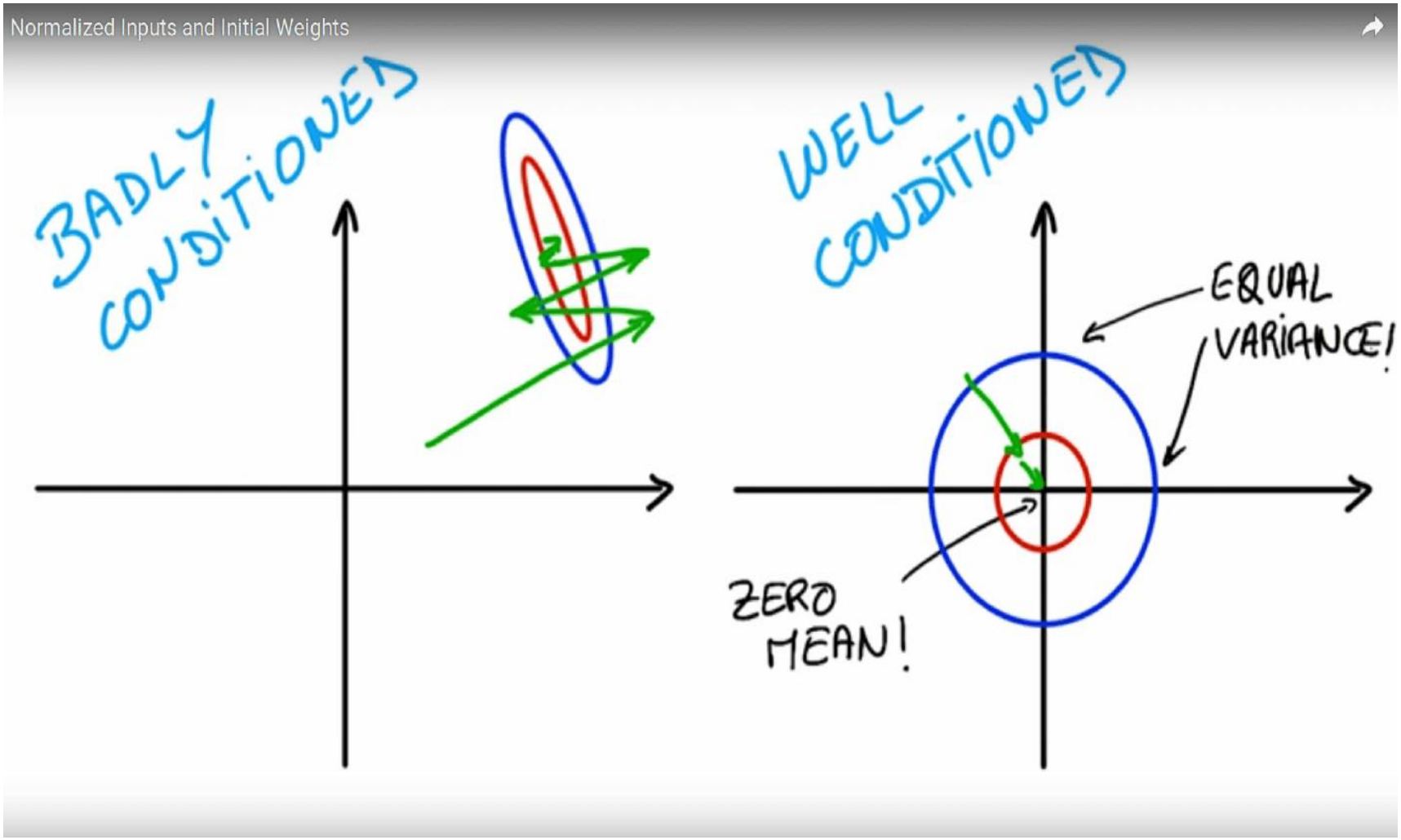
Optimisation of average loss



Gradient decent



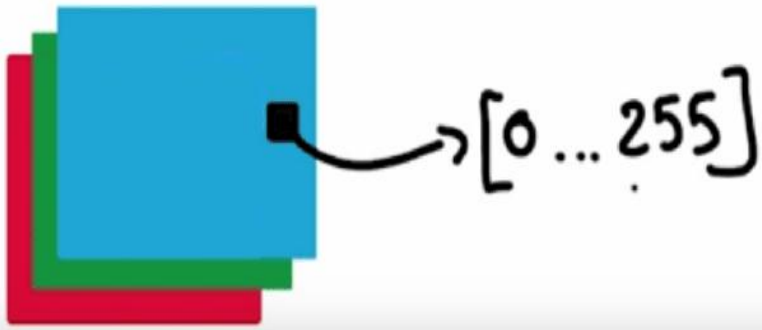
Normalised input and output



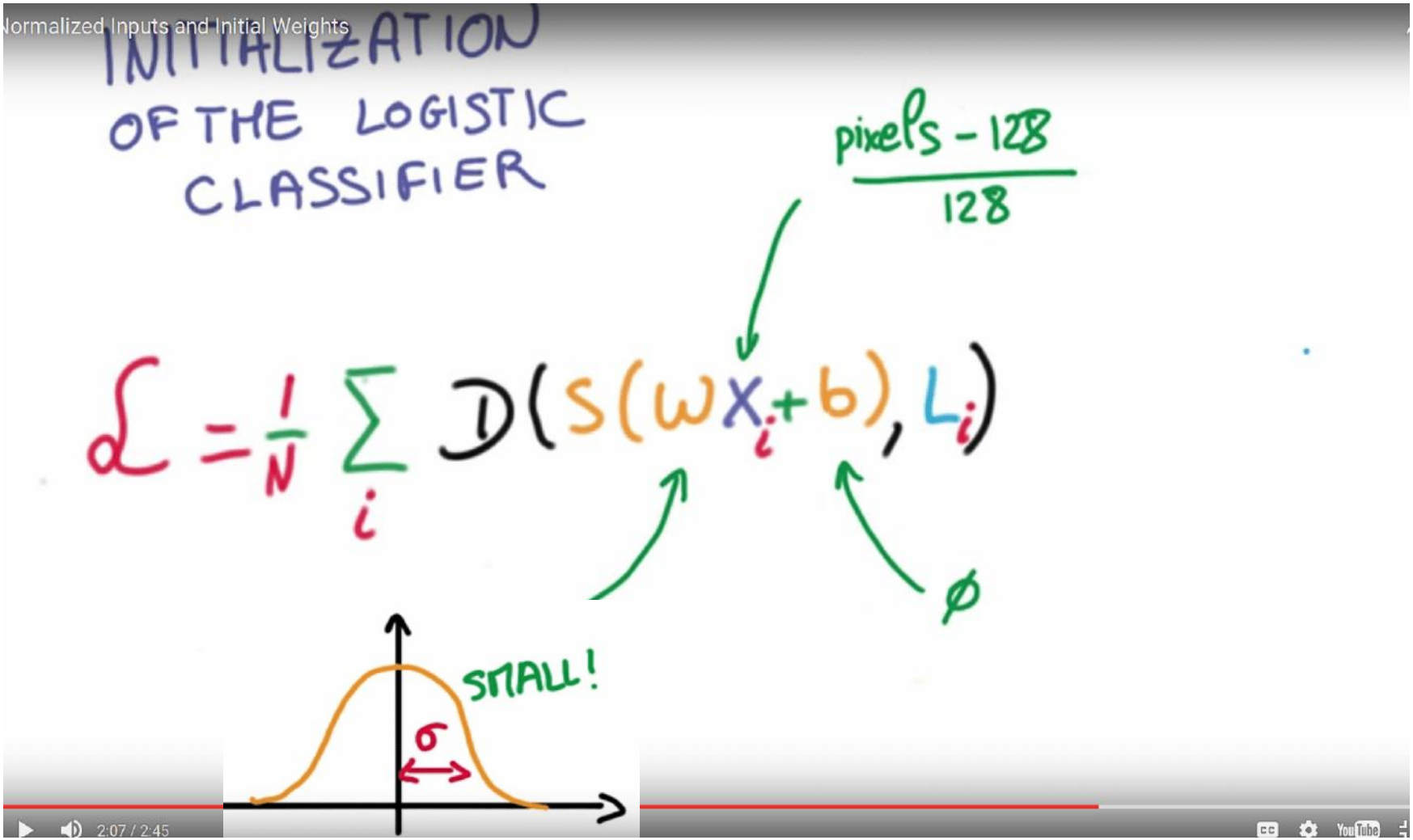
Normalised input and output

IMAGES

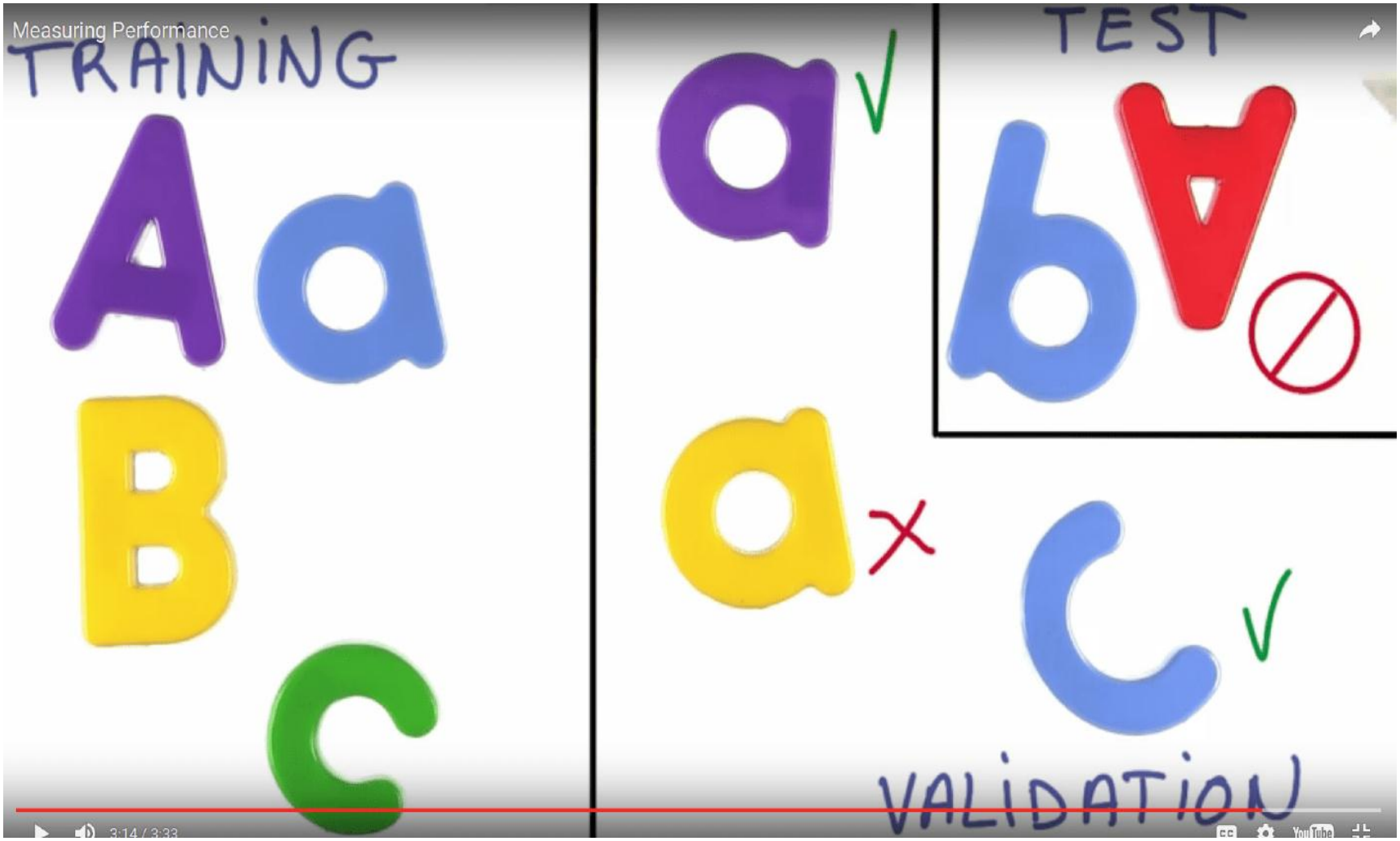
$$\frac{R-128}{128} \qquad \frac{G-128}{128} \qquad \frac{B-128}{128}$$



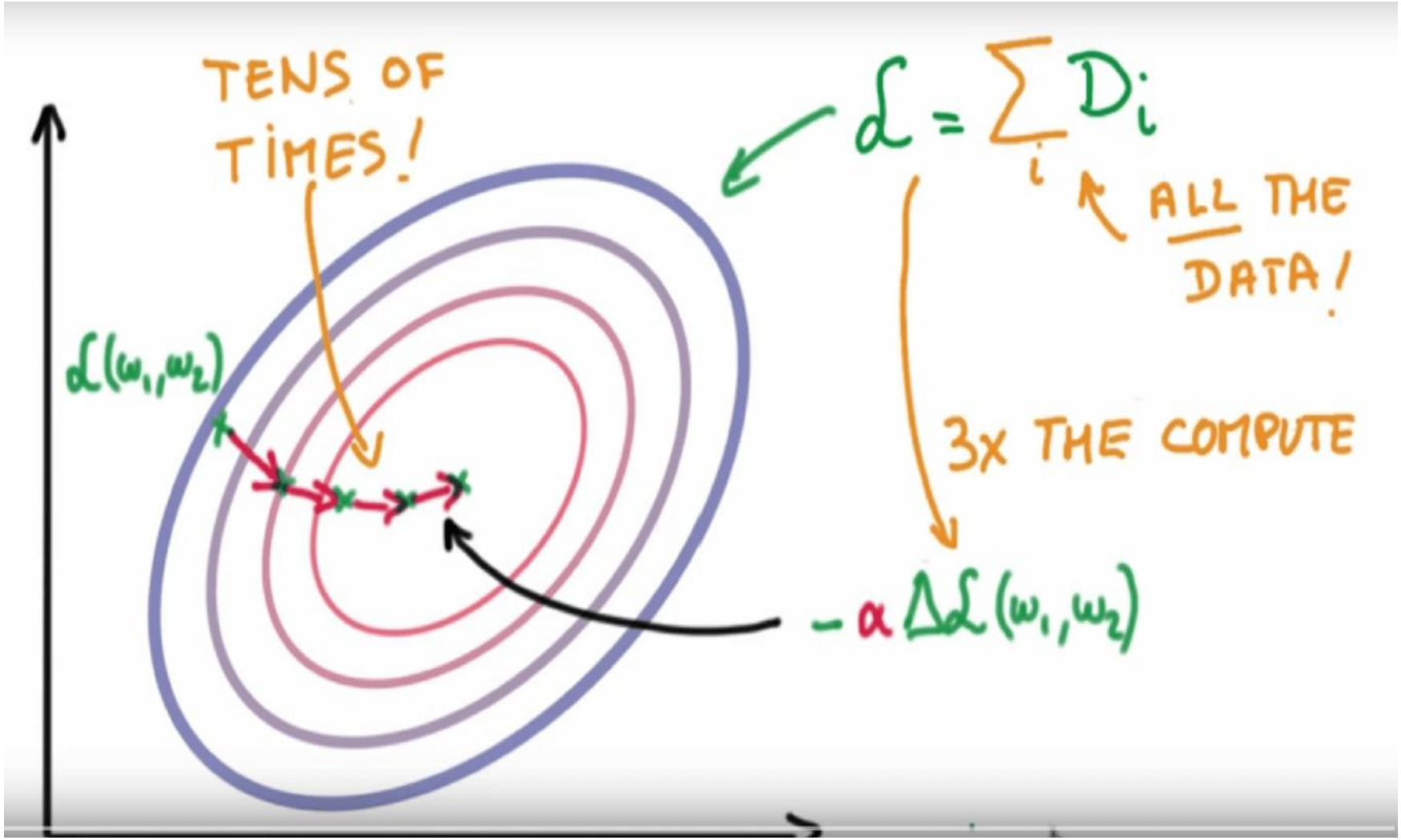
Initialisation



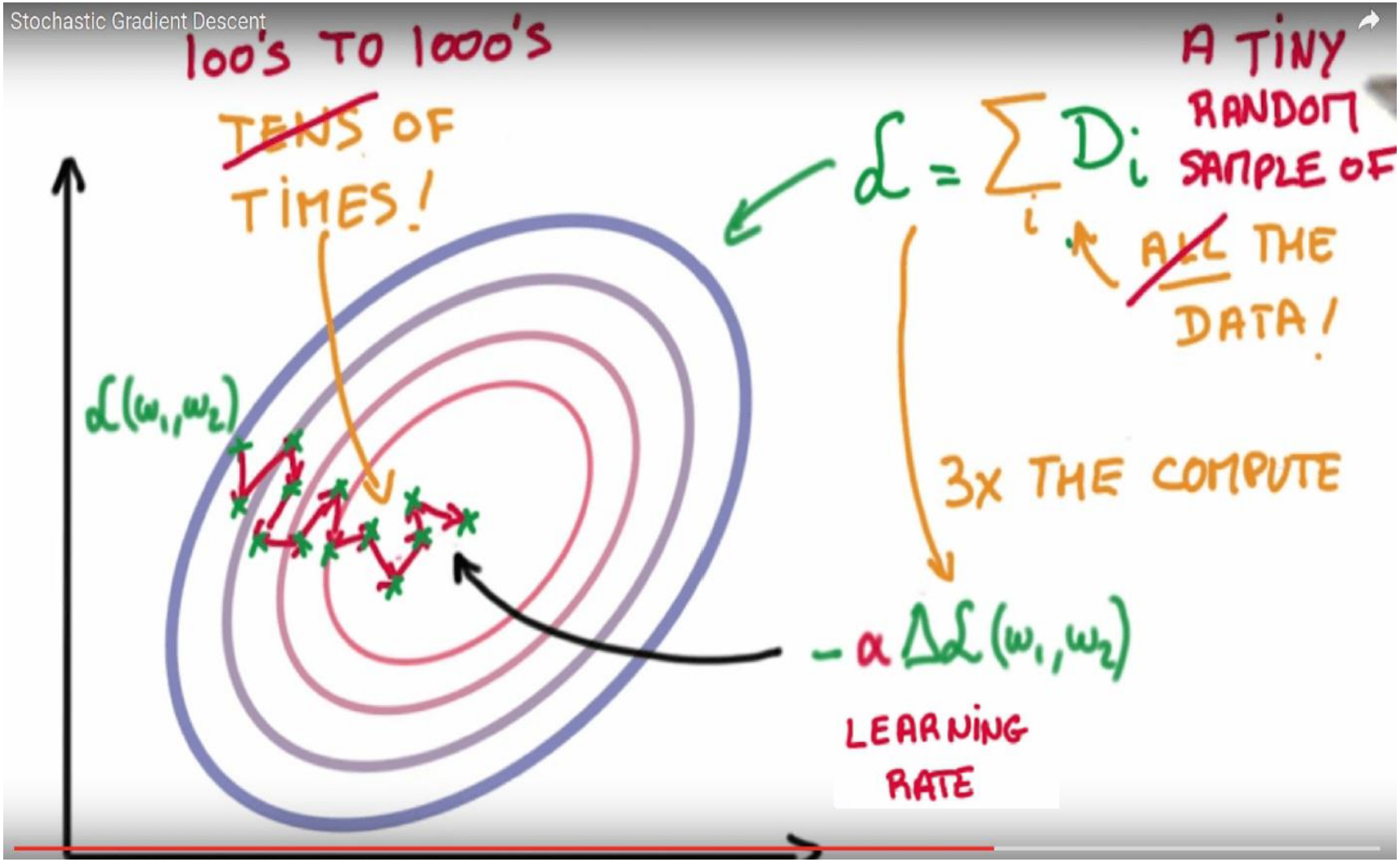
Training, validation, testing



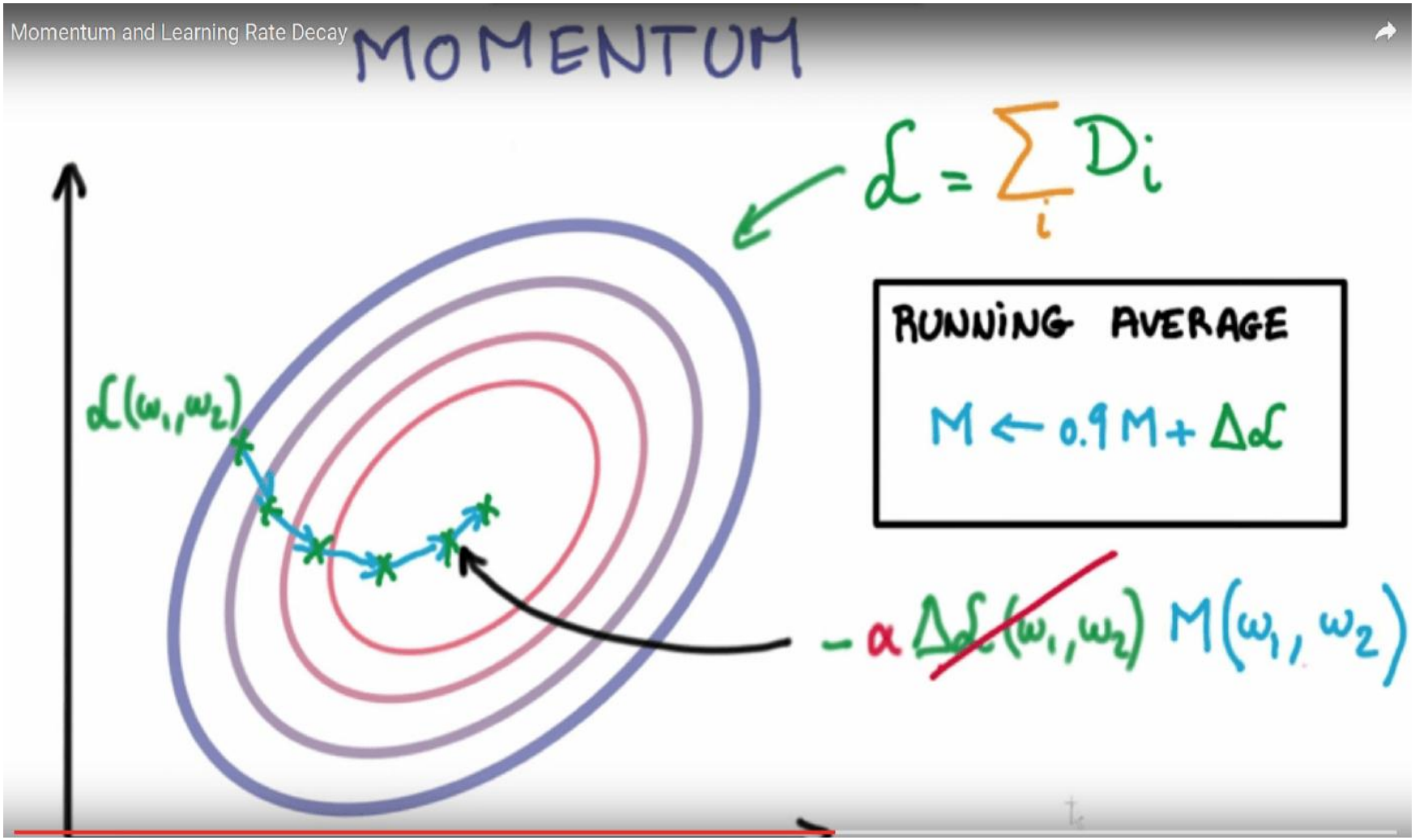
Gradient Descent



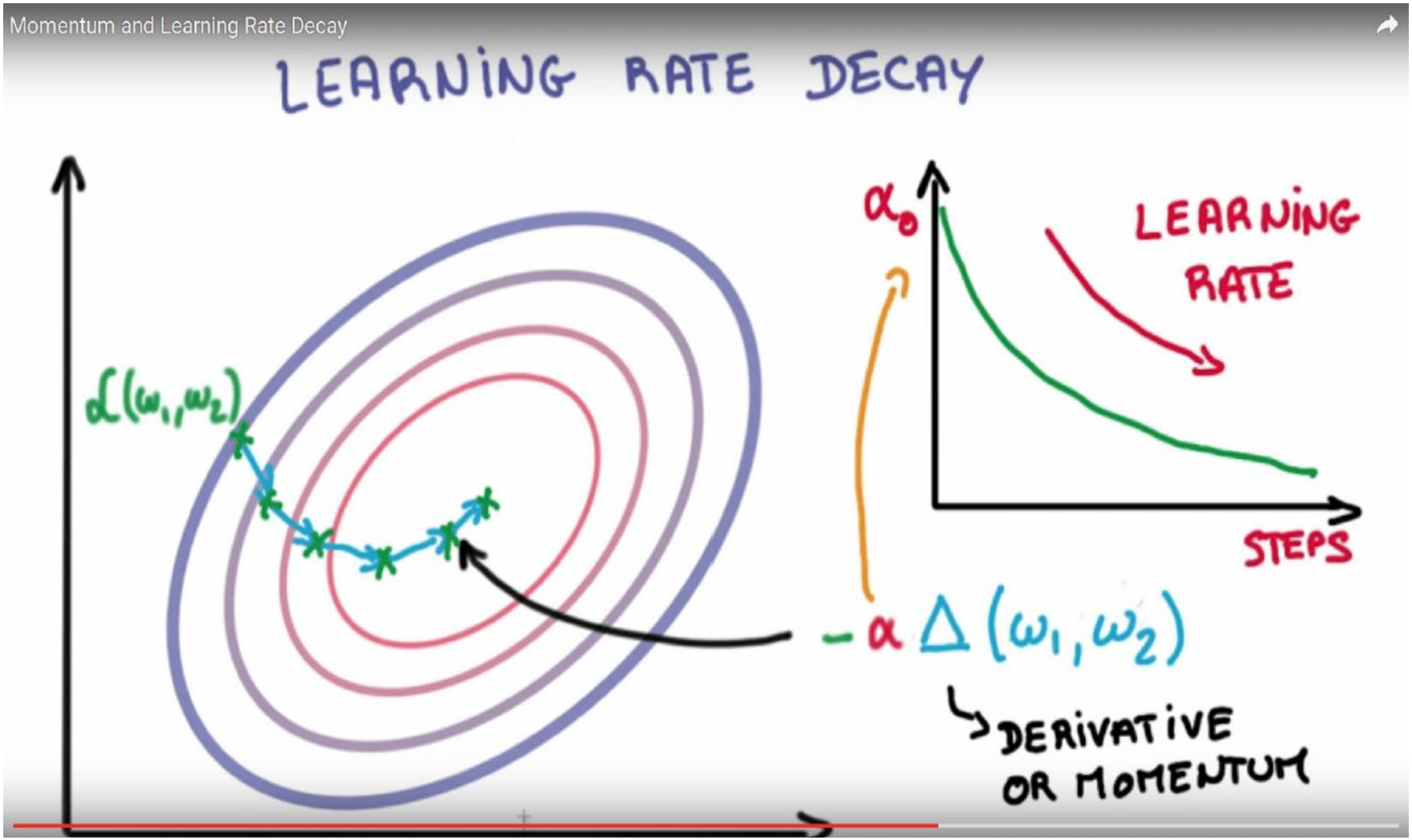
Stochastic Gradient Descent



SDG: optimising with momentum



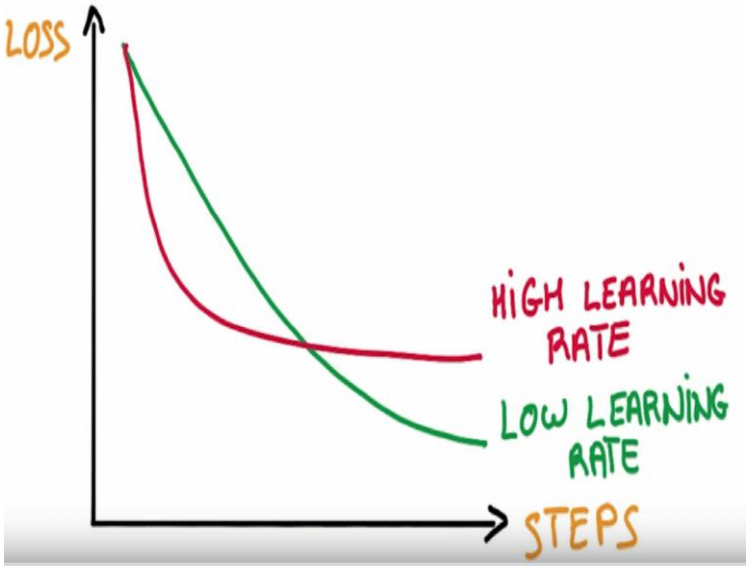
SDG: learning rate



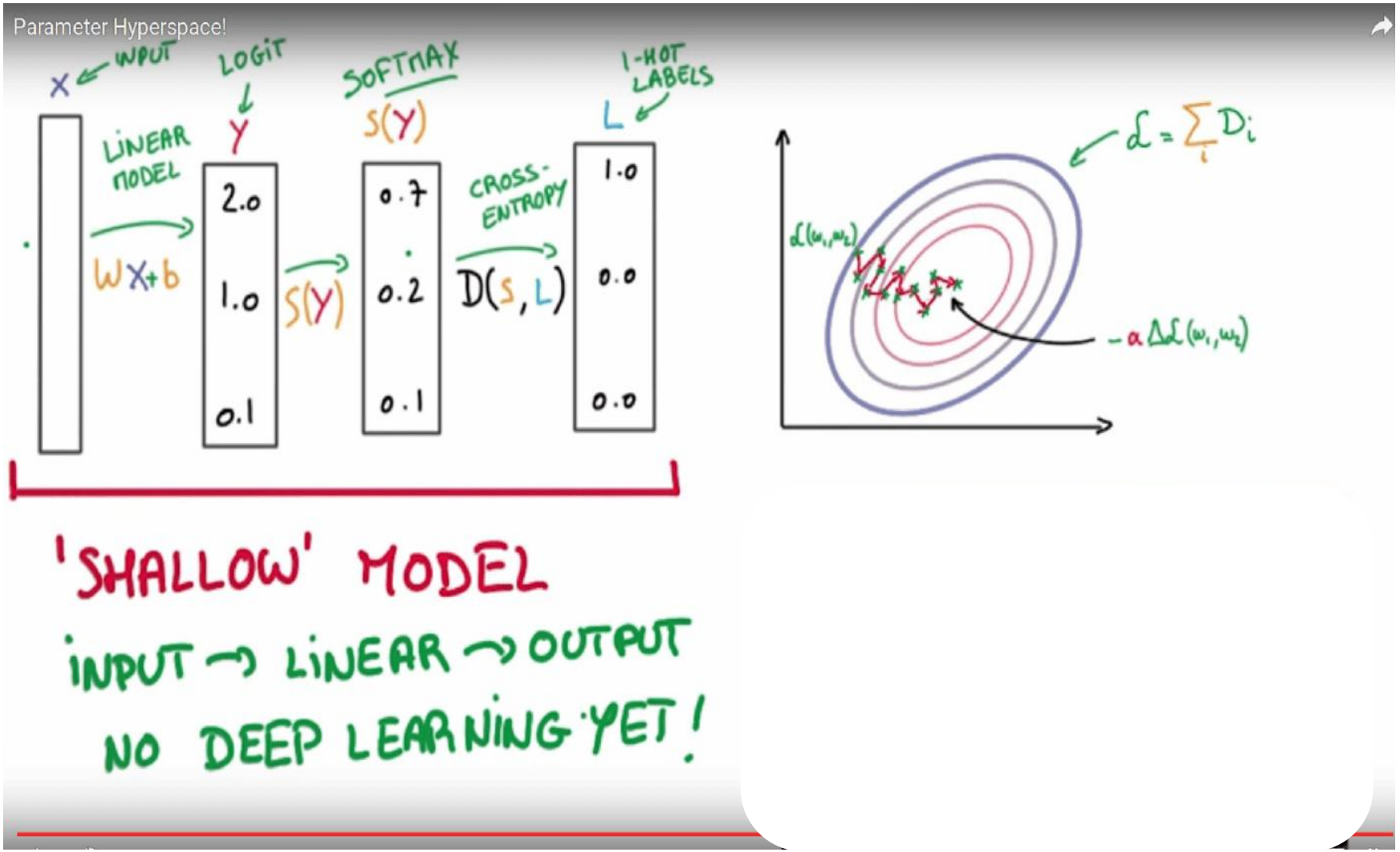
SDG: „black magic”

MANY HYPER-PARAMETERS

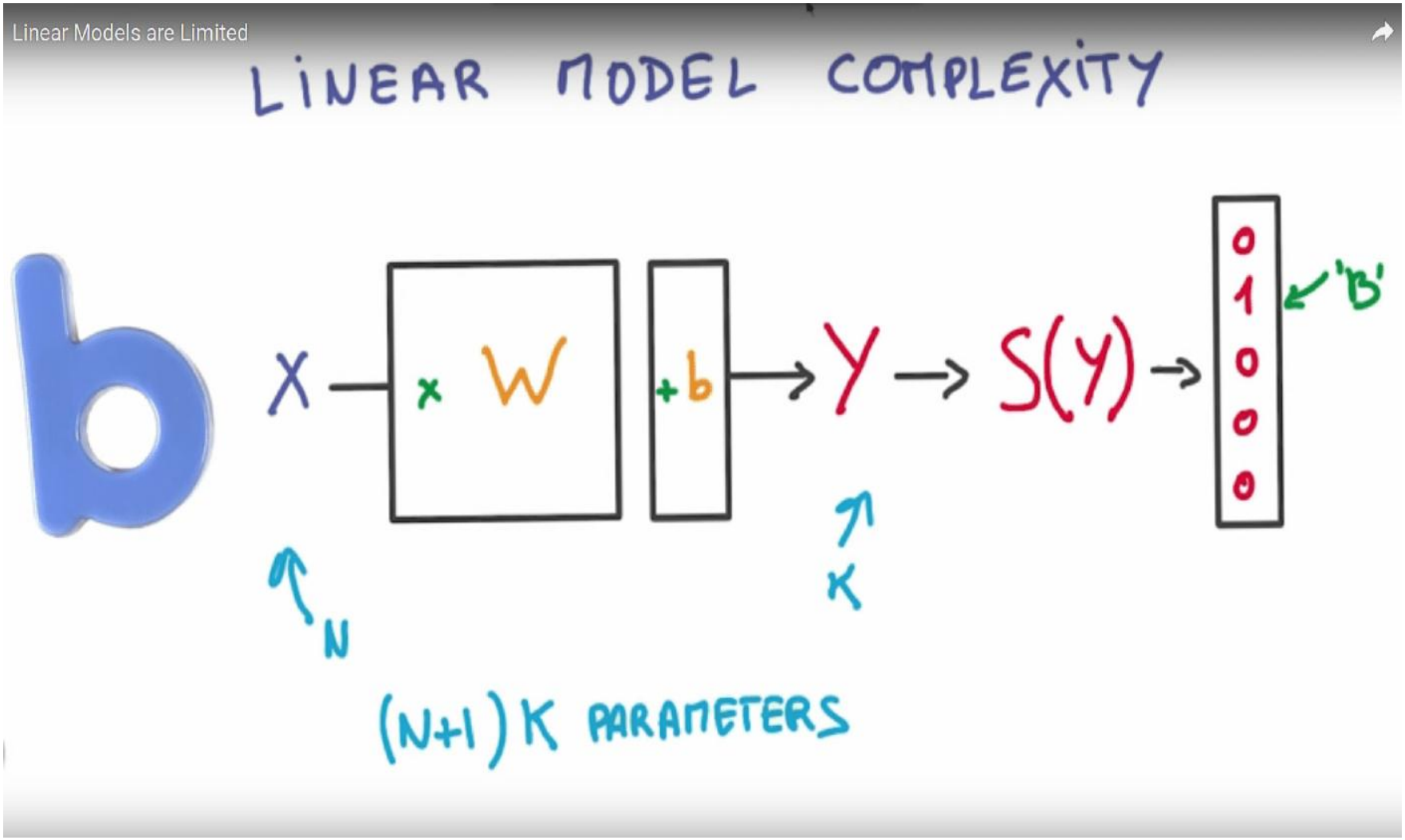
- INITIAL LEARNING RATE
- LEARNING RATE DECAY
- MOMENTUM
- BATCH SIZE
- WEIGHT INITIALIZATION



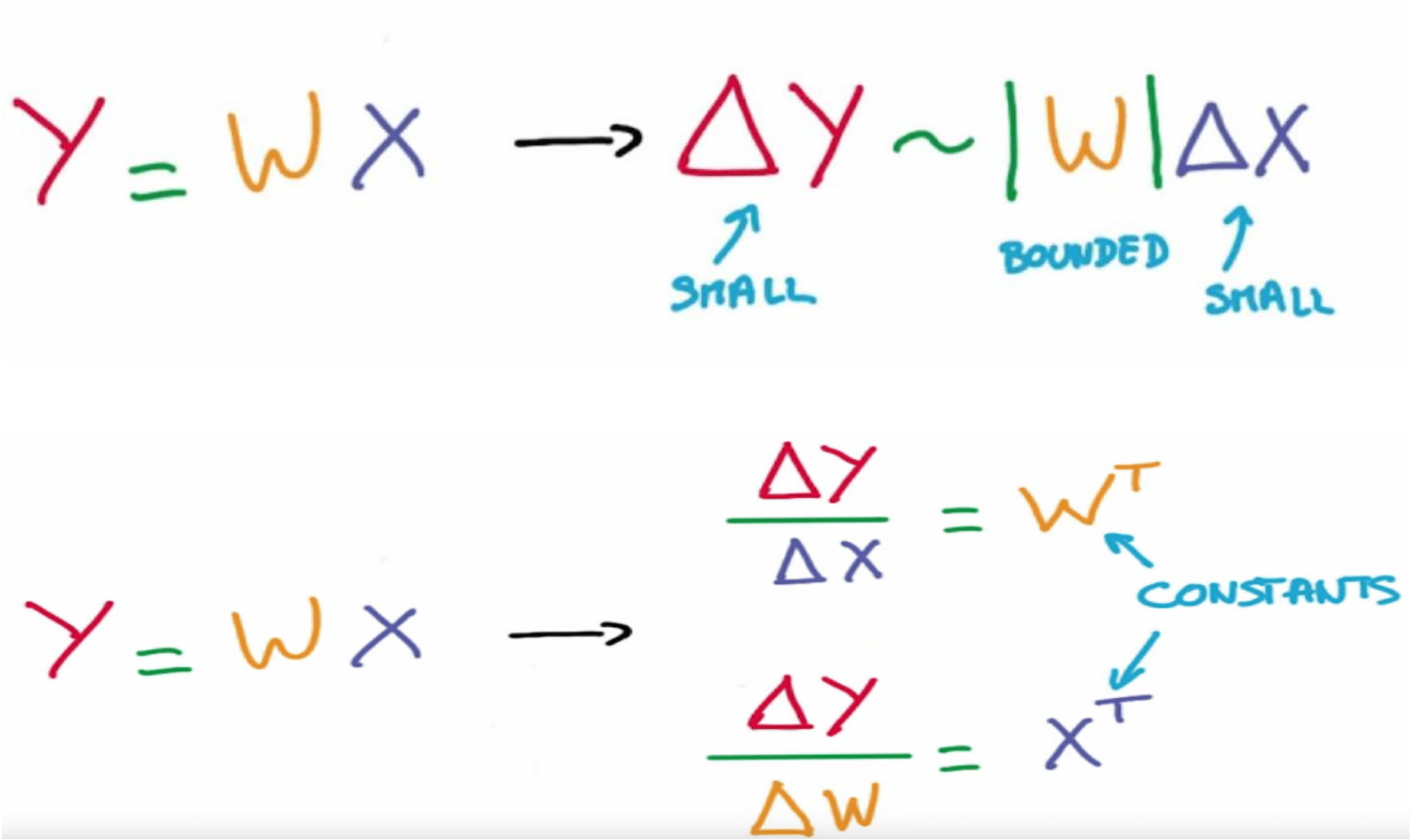
Input – linear - output



Linear models are linear



Linear models are stable



Linear models are here to stay

- **This is still linear**

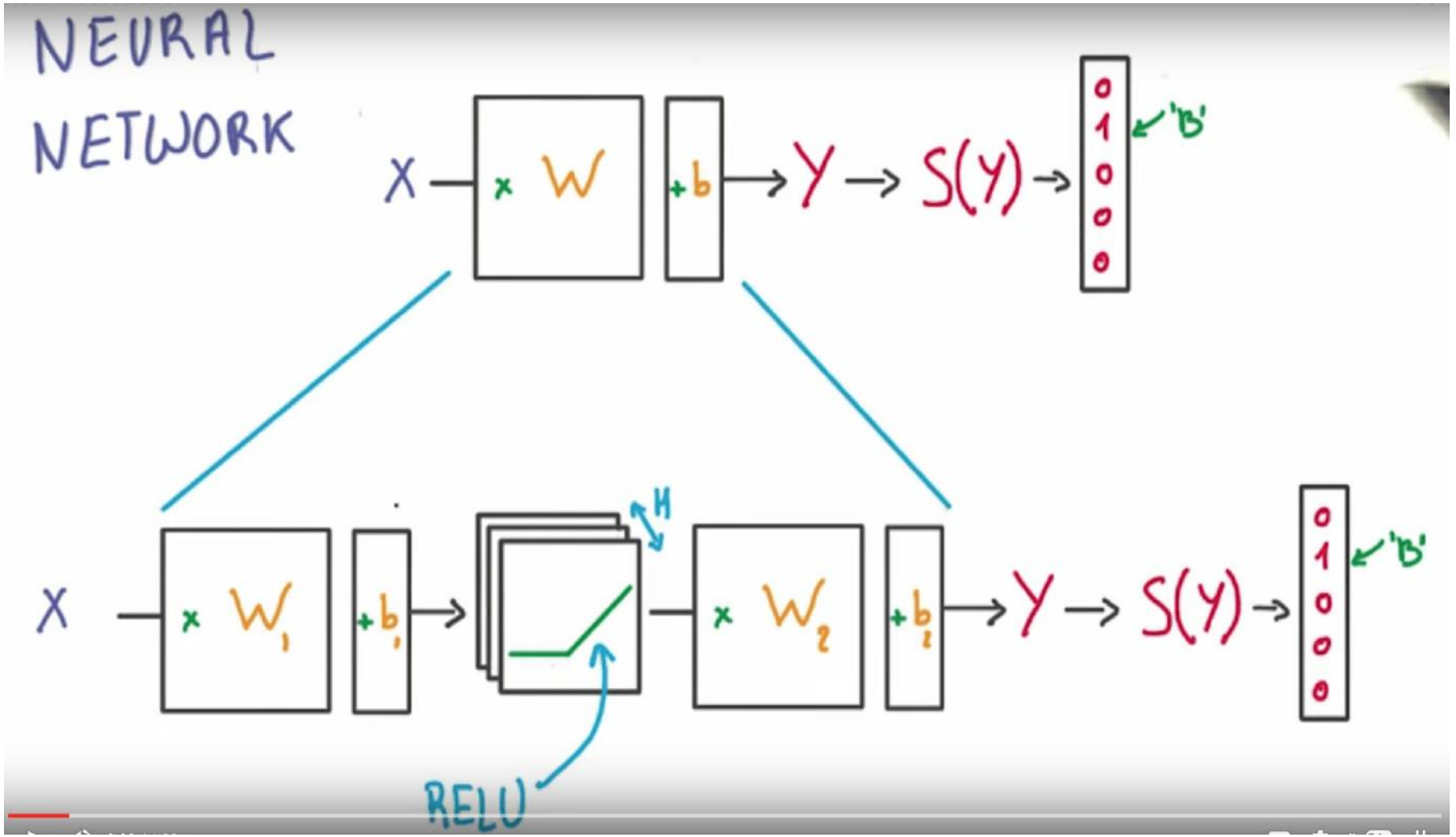
$$Y = W_1 W_2 W_3 X = W X$$

- **Lets introduce non-linearity**

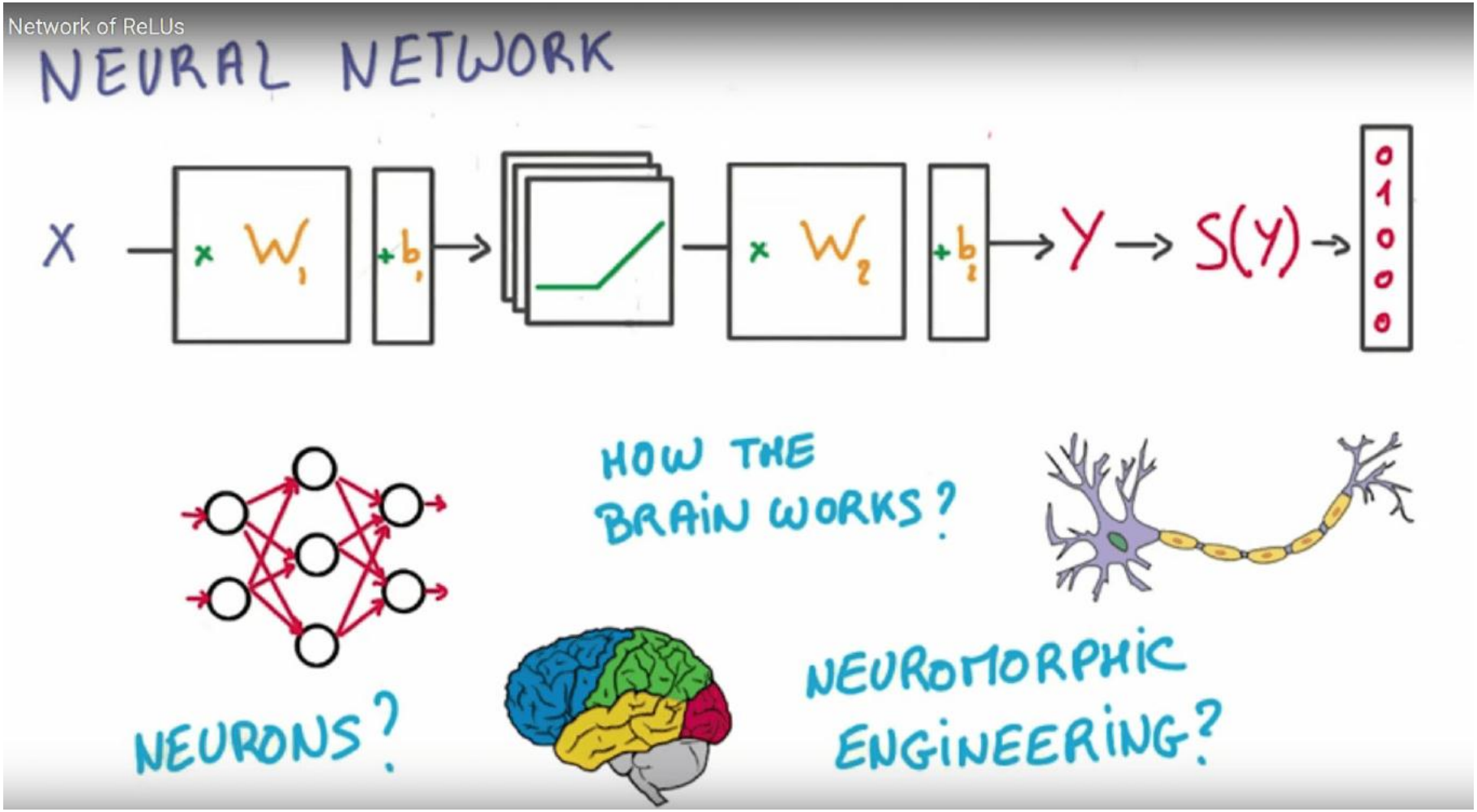
$$Y = W_1 W_2 W_3 X = [illegible] X$$

NON-LINEARITIES

RELU: Rectified Linear Unit



Networks of RELU



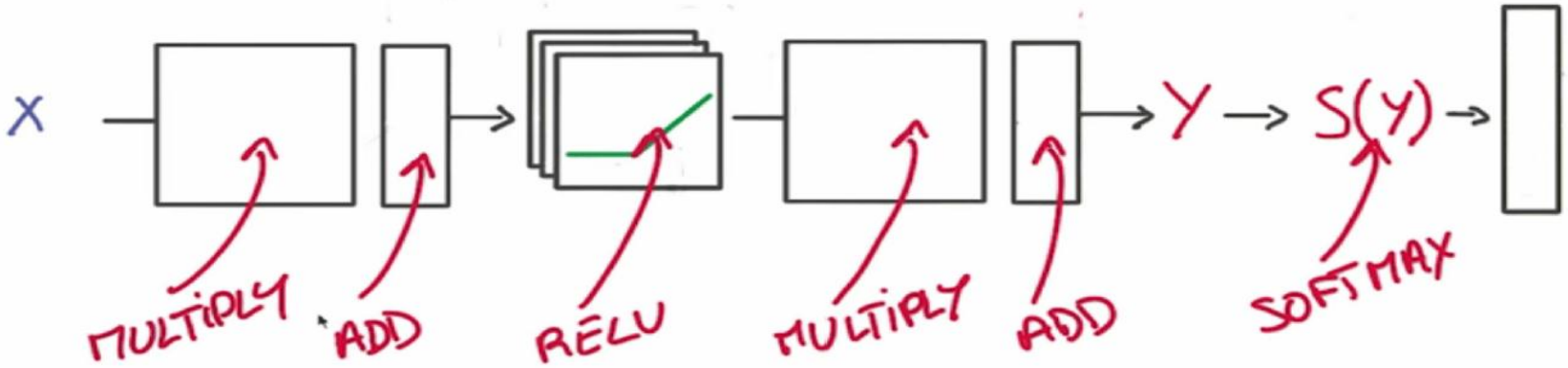
The Chain Rule

$$[g(f(x))]' = g'(f(x)) \cdot f'(x)$$

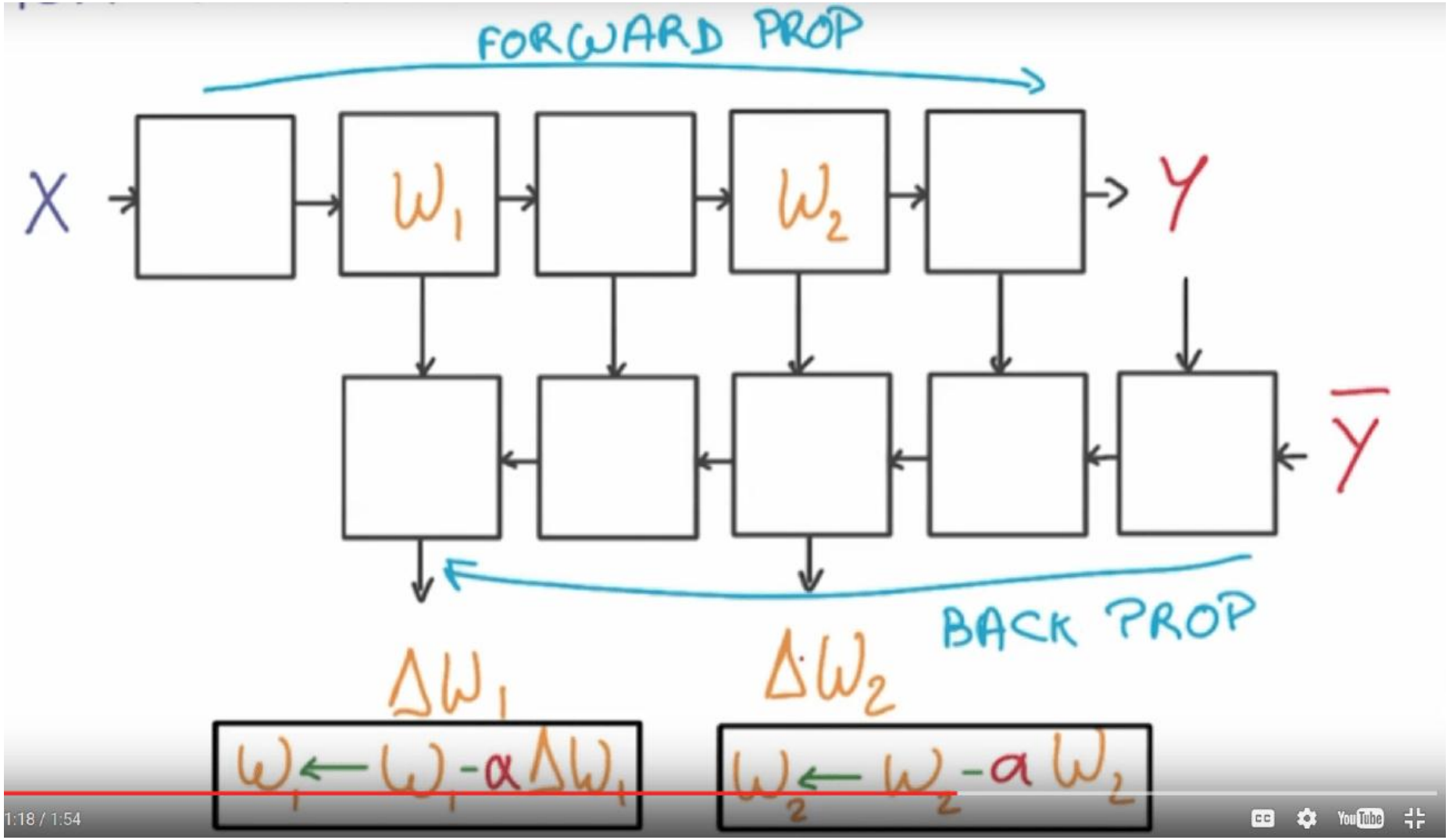
DERIVATIVE

PRODUCT

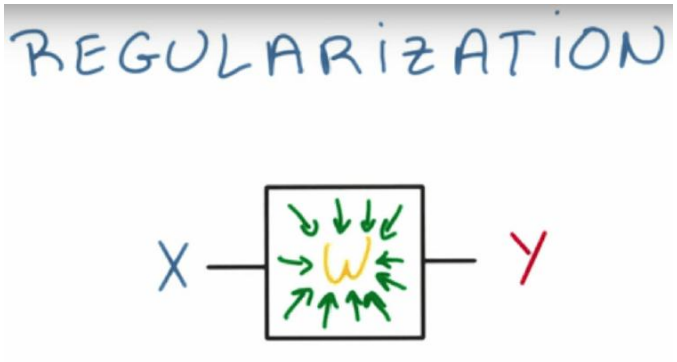
STACKING UP SIMPLE OPERATIONS

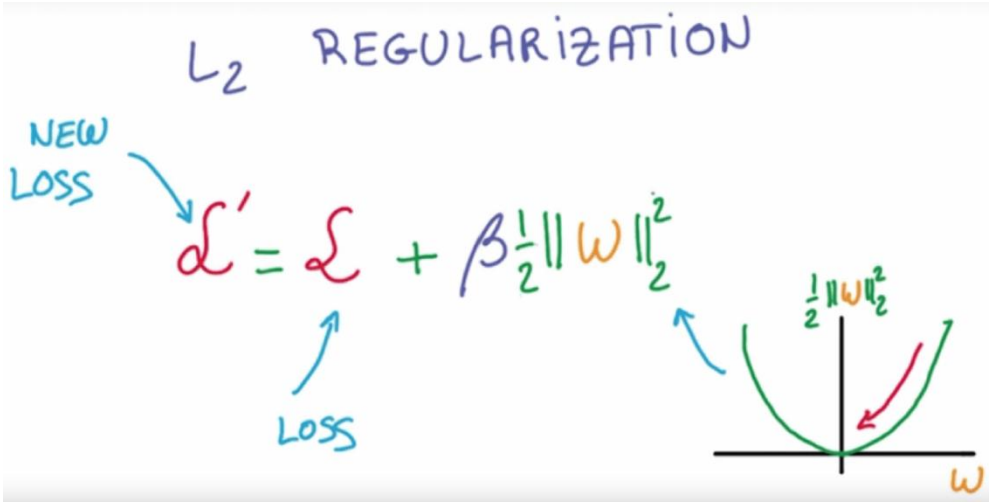


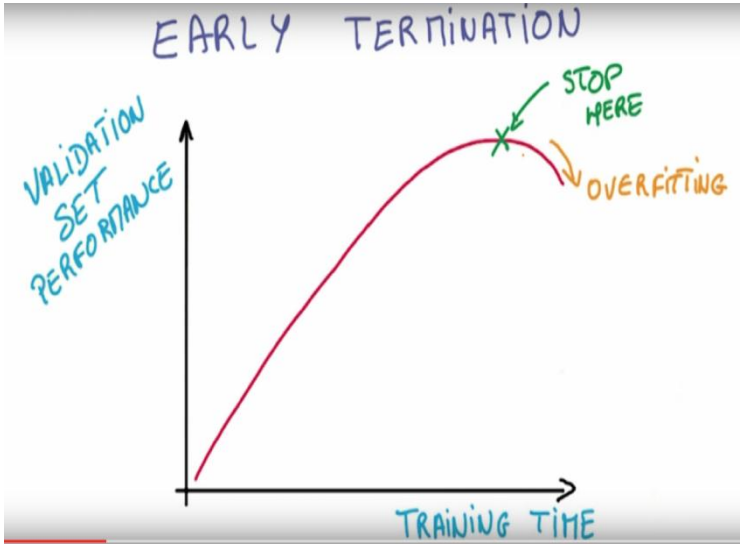
Back - propagation

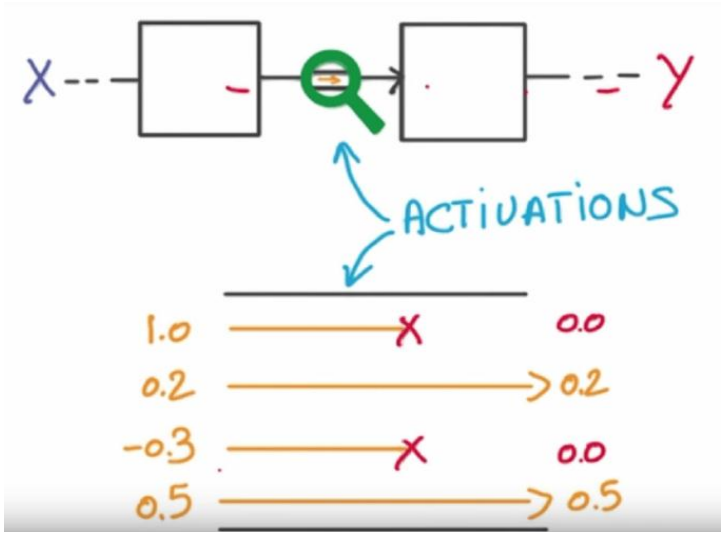


Optimisation tricks

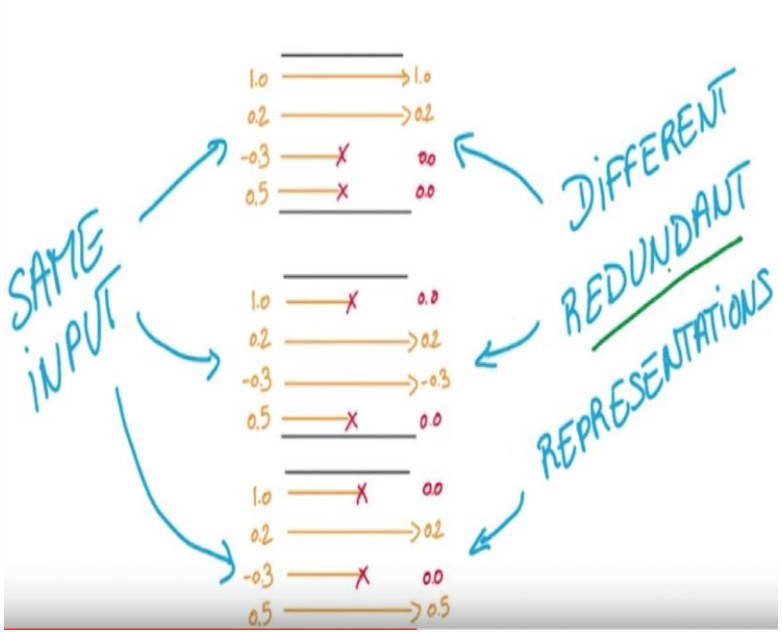


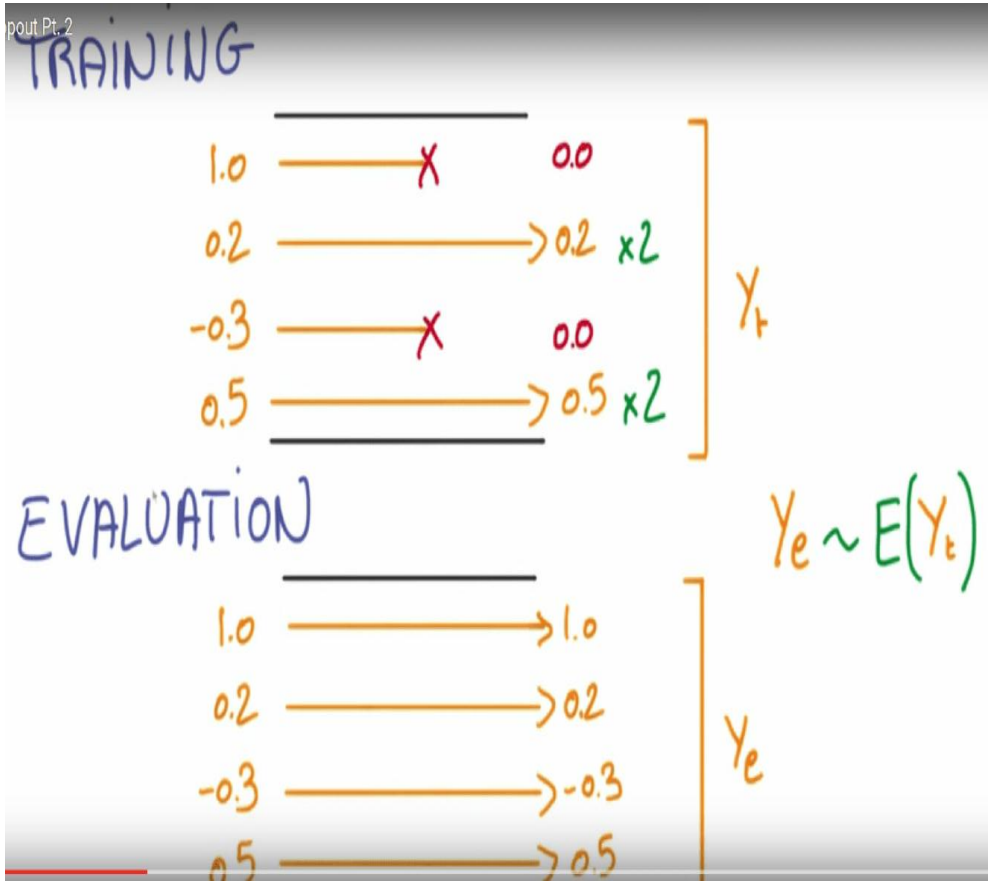




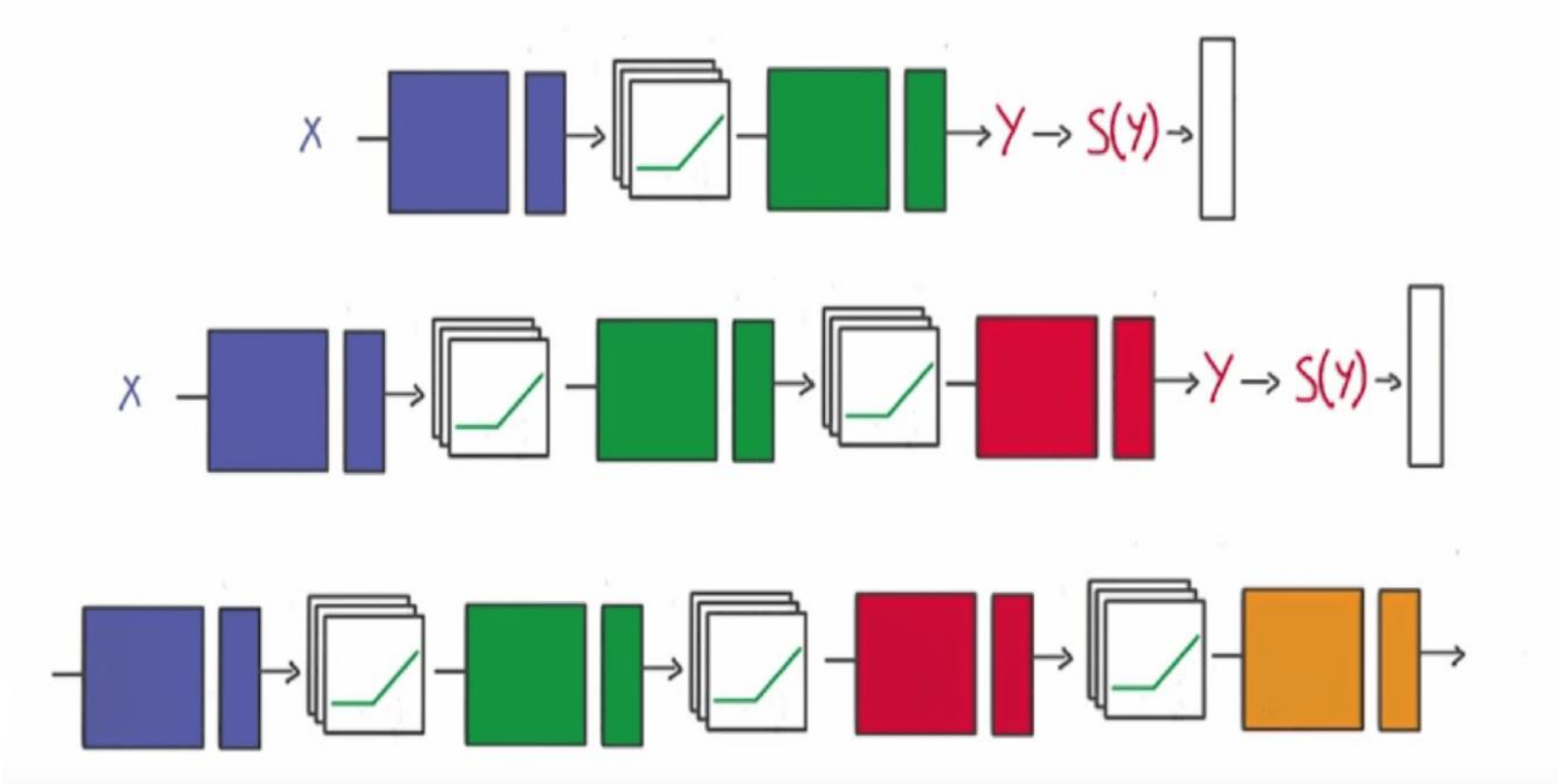


Optimisation trick: dropout

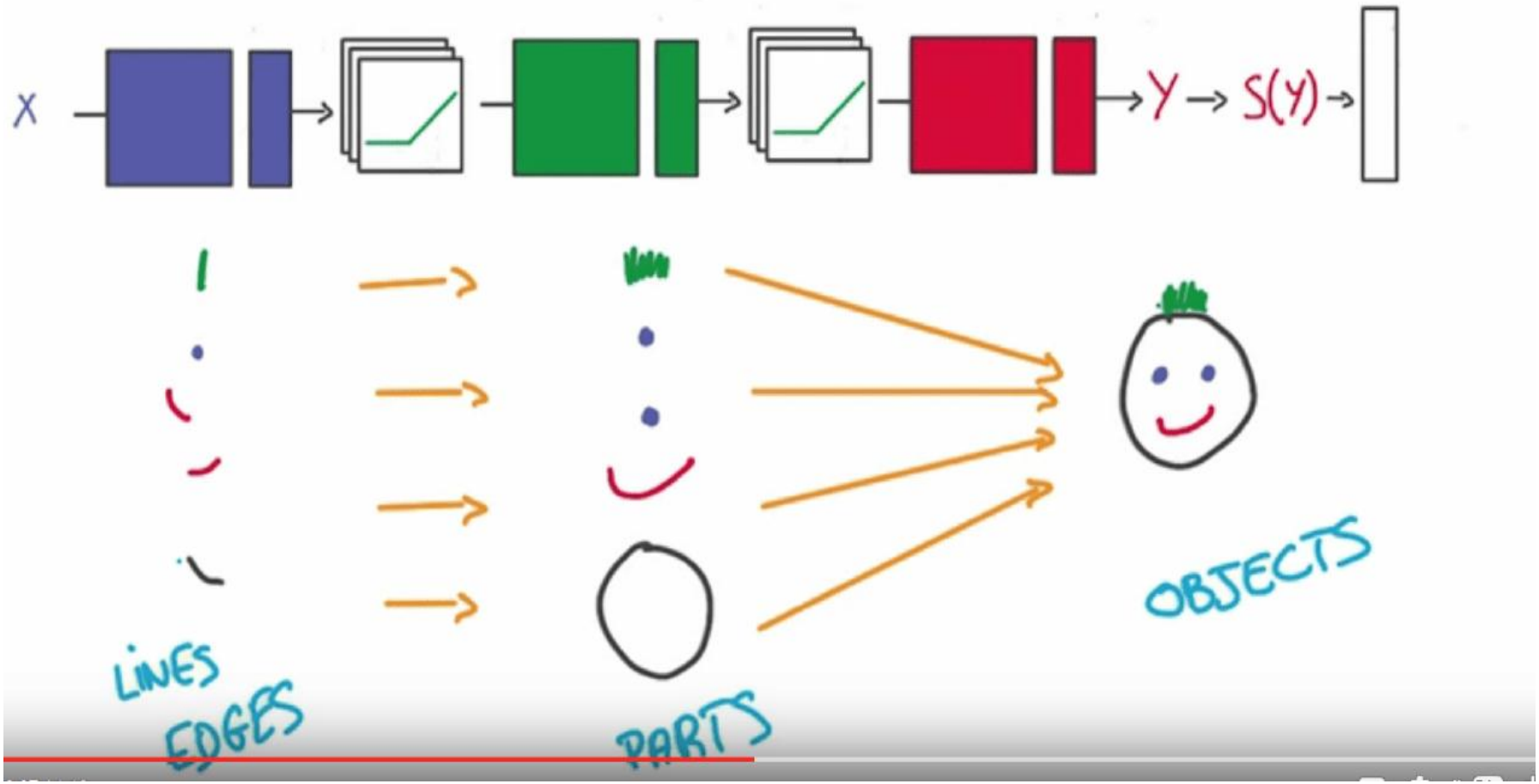




Deep networks



Deep networks



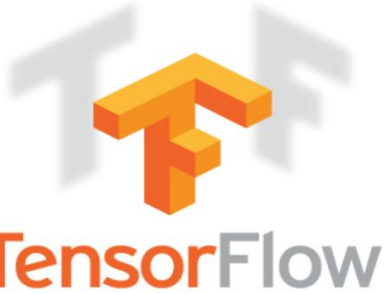
tensorflow.org/paper/whitepaper2015.pdf

TensorFlow:
Large-Scale Machine Learning on Heterogeneous Distributed Systems
(Preliminary White Paper, November 9, 2015)

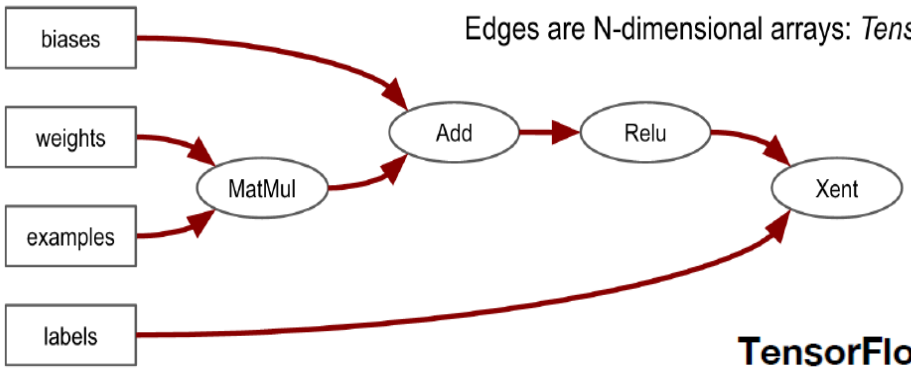
Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro,
Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow,
Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser,
Manjunath Kudlur, Josh Levenberg, 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

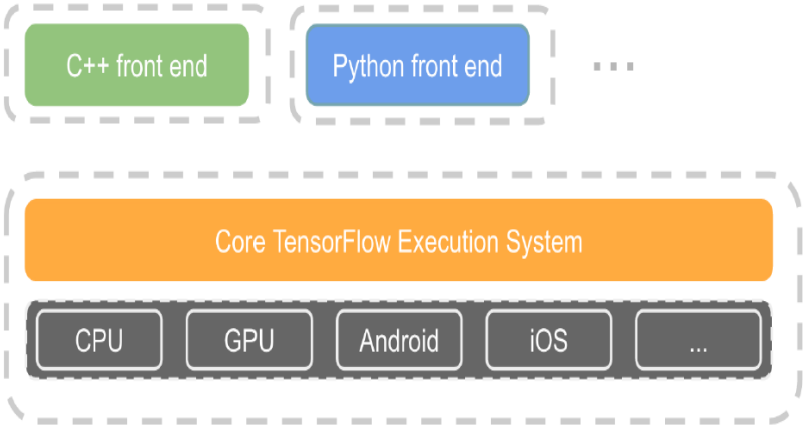
What is a Data Flow Graph?

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

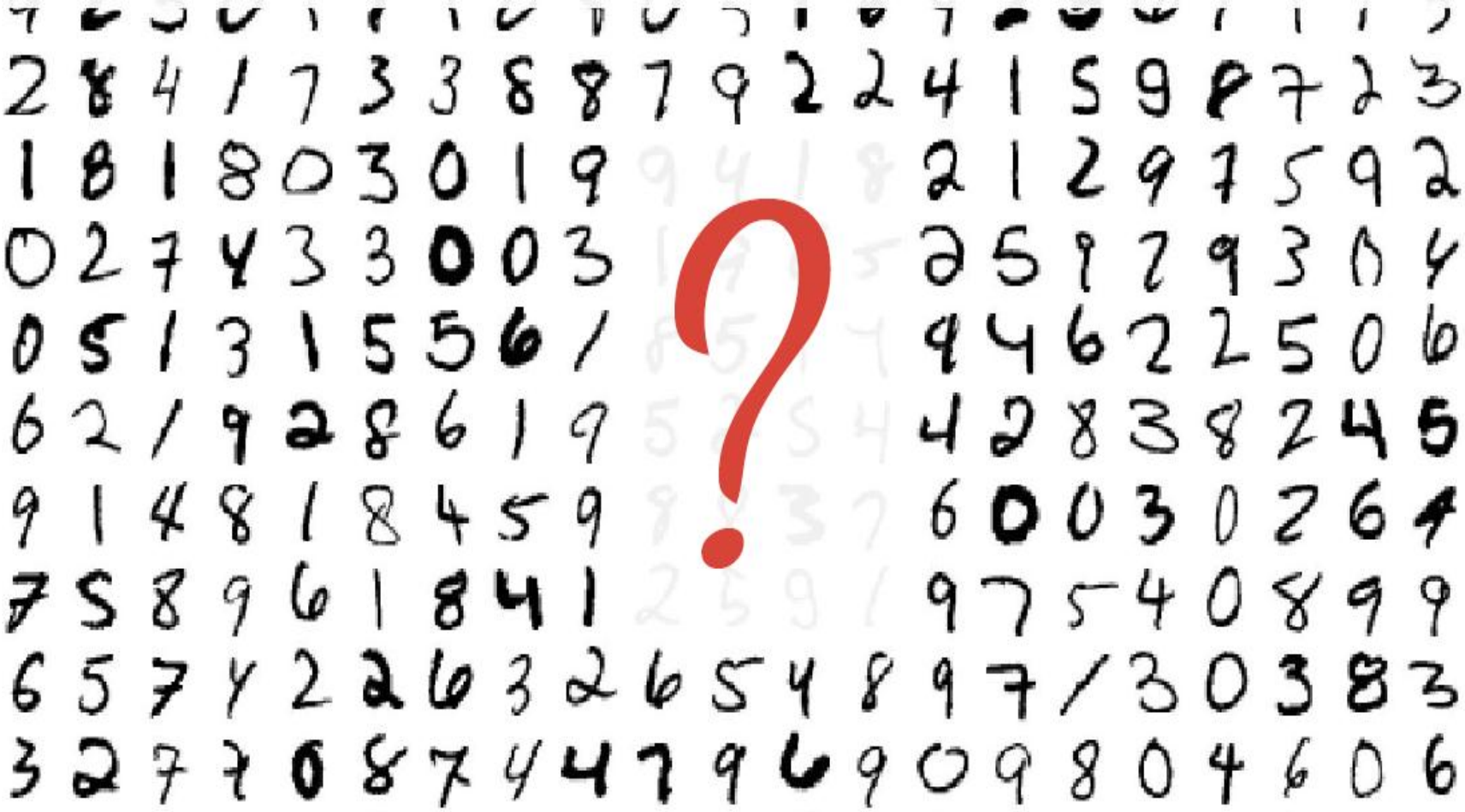
There are hundreds of predefined Ops (and easy to add more)

- 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

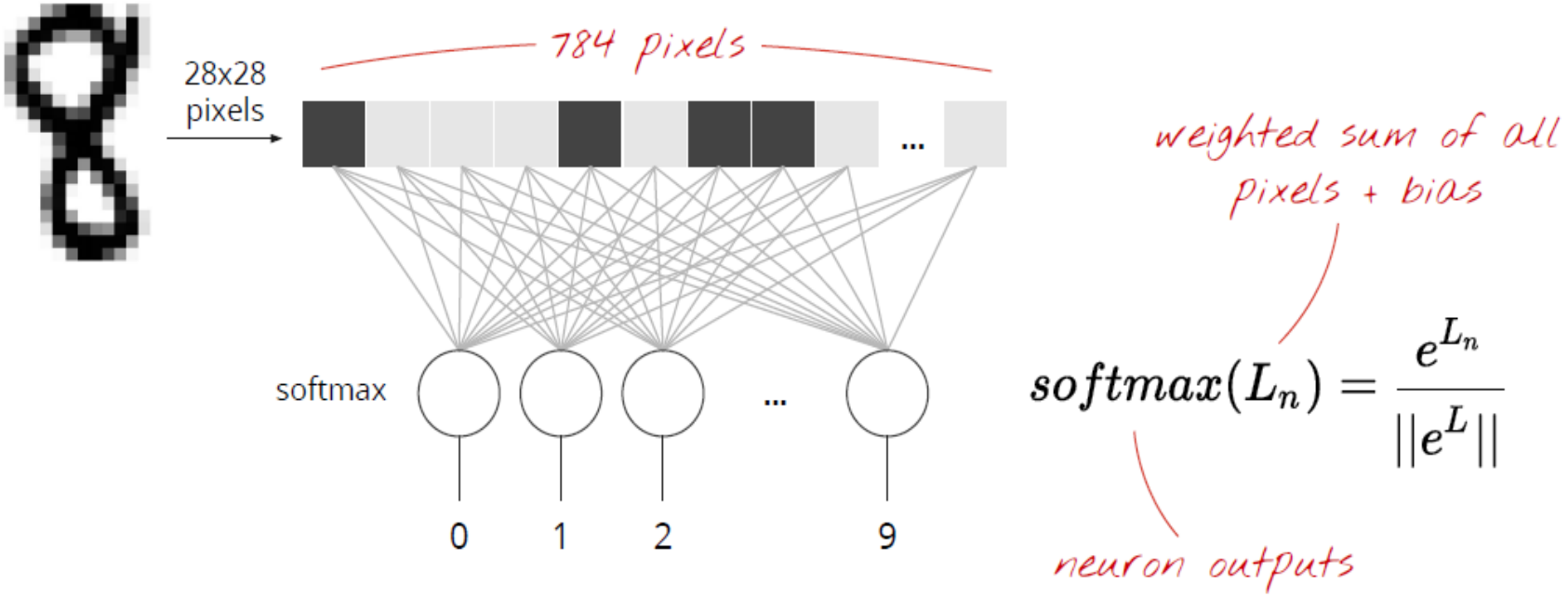
- Activations: sigmoid, tanh, relu, dropout, ...
- Pooling: avg, max.
- Convolutions: with many options.
- Normalization: local, batch, moving averages.
- Classification: softmax, softmax loss, cross entropy loss, topk.
- Embeddings: lookups/gather, scatter/updates.
- Sampling: candidate sampler (various 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



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

TensorFlow full python code

initialisation

```python
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

```python
# 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])
```

success metrics

```python
# 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))
```

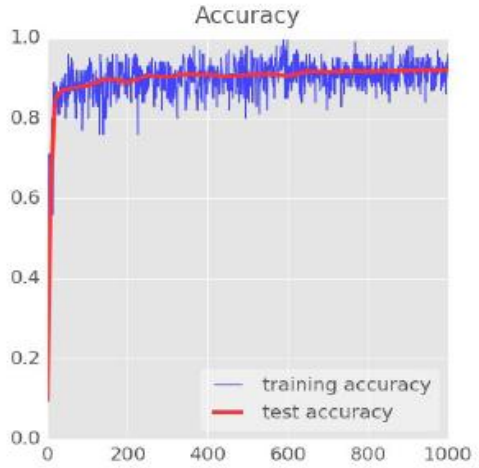
training step

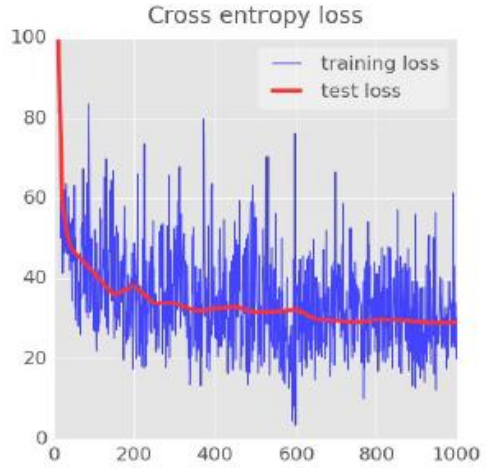
```python
optimizer = tf.train.GradientDescentOptimizer(0.003)
train_step = optimizer.minimize(cross_entropy)

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)
```

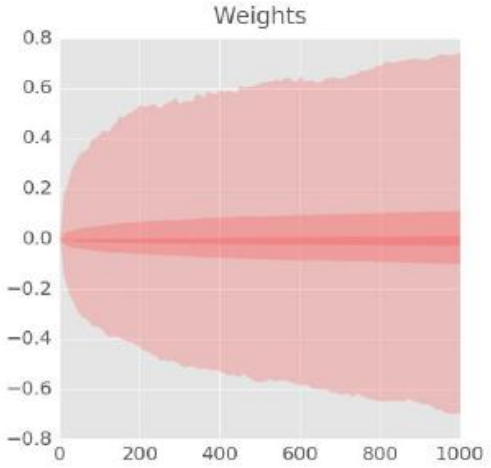
Run

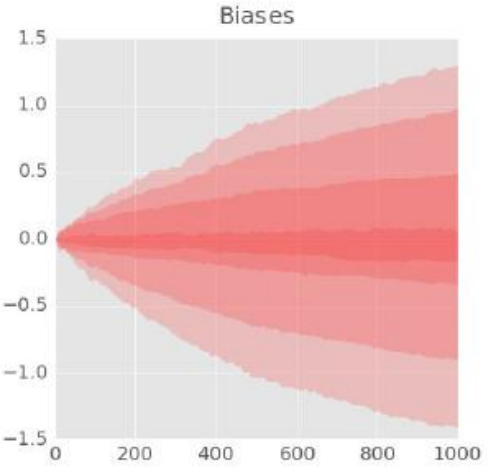
Slides from M. Gorner@youtube

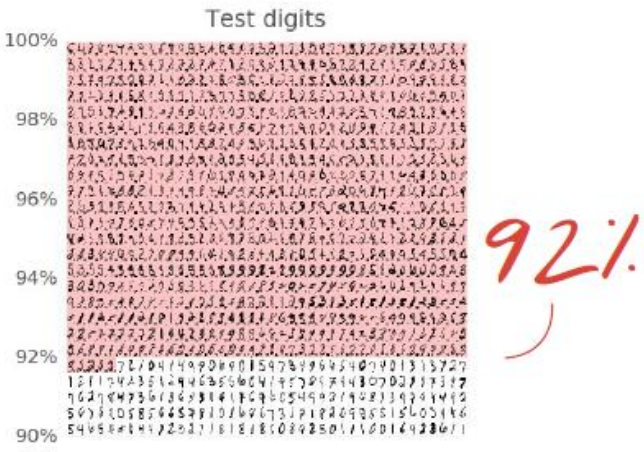
Simple linear model





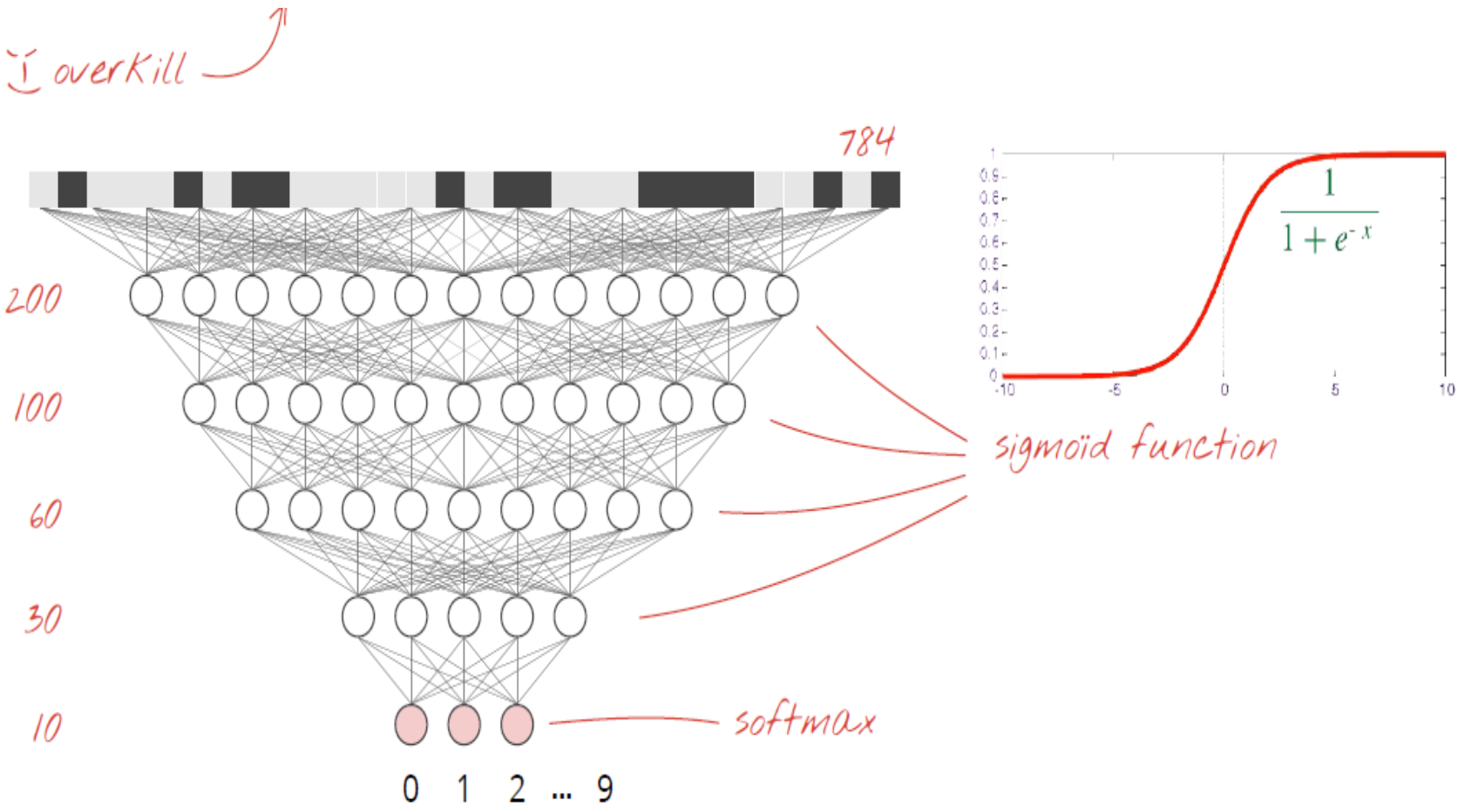






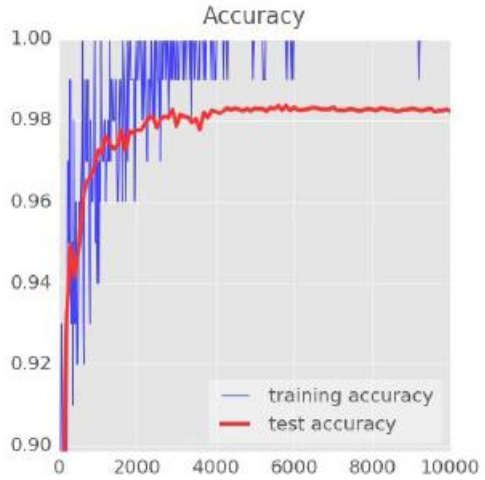
Slides from M. Gorner@youtube

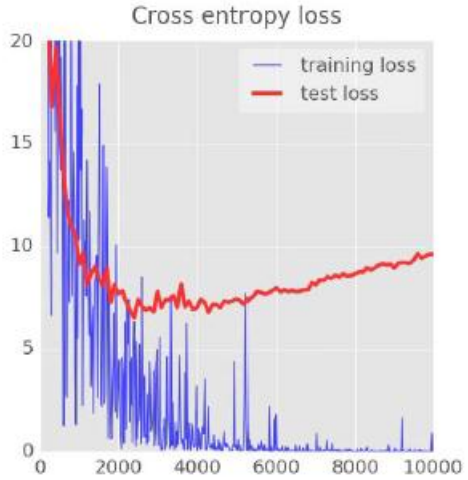
Multi-layer connected network

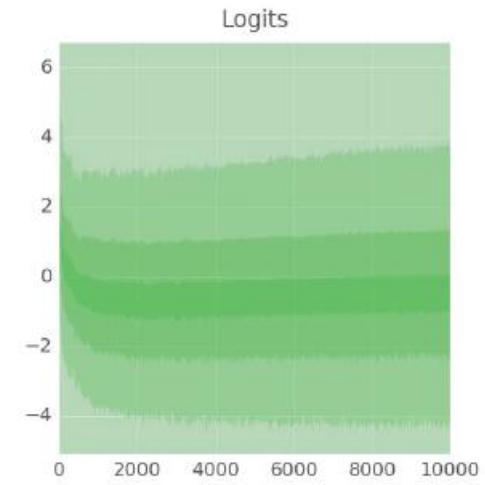


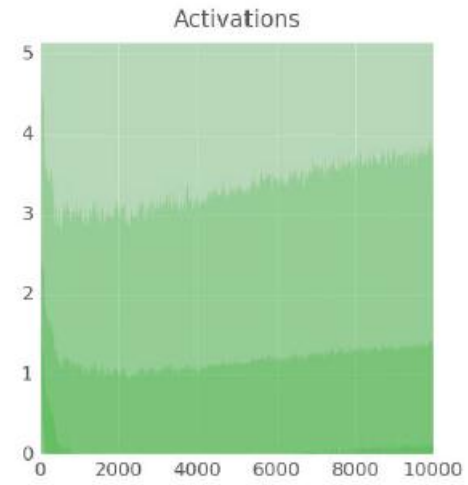
Slides from M. Gorner@youtube

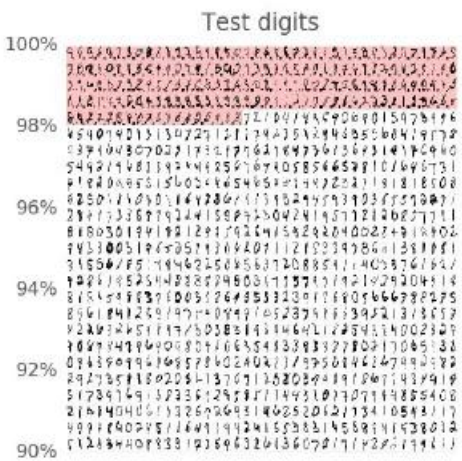
Multi-layer connected network







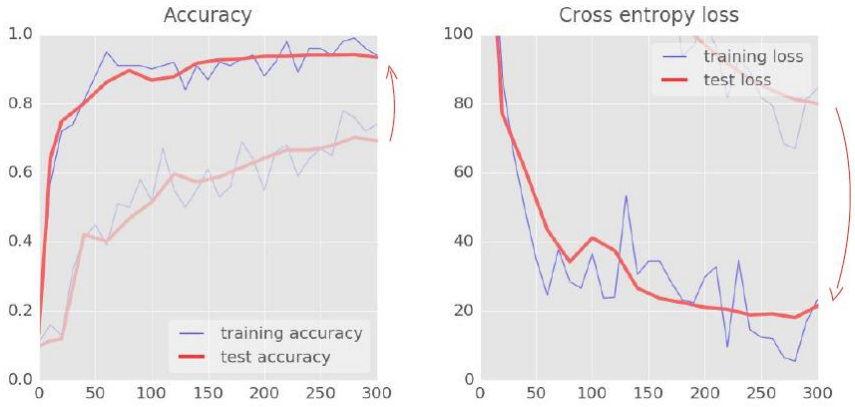




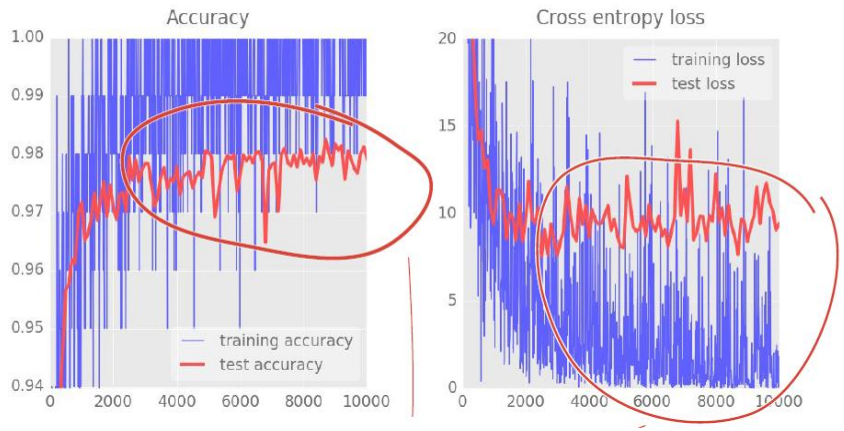
98%

All tricks count

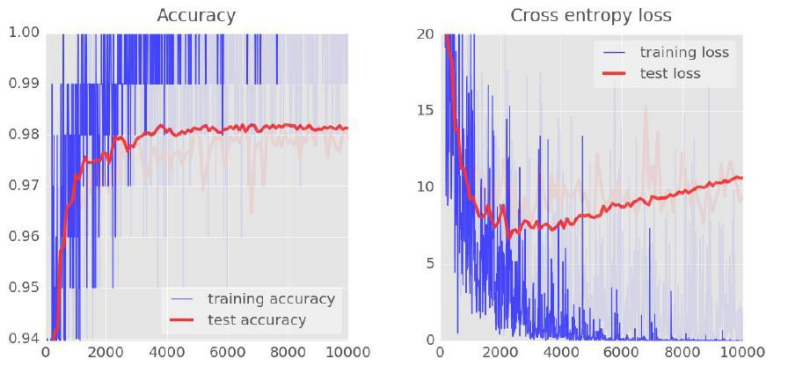
Use RELU



But noisy accuracy

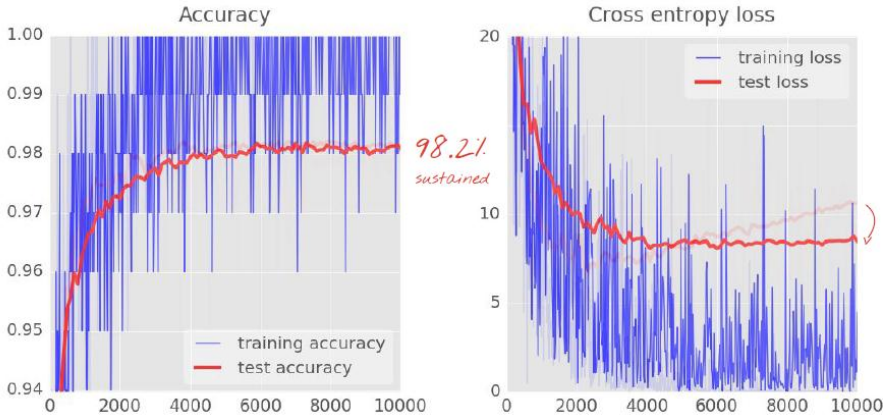


Exponentialy reduce learning rates



Learning rate 0.003 at start then dropping exponentially to 0.0001

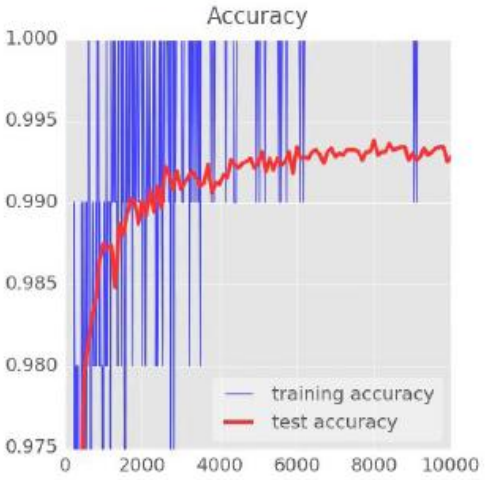
Add drop-out

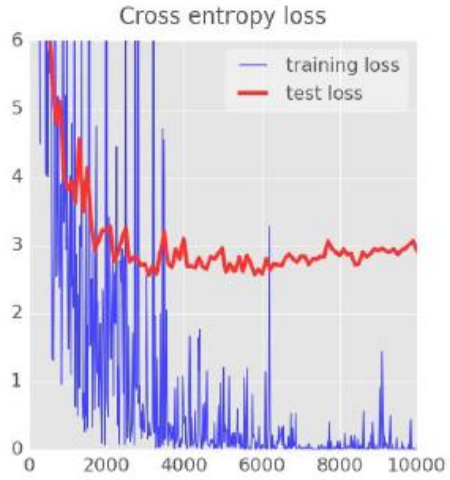


RELU, decaying learning rate 0.003 -> 0.0001 and dropout 0.75

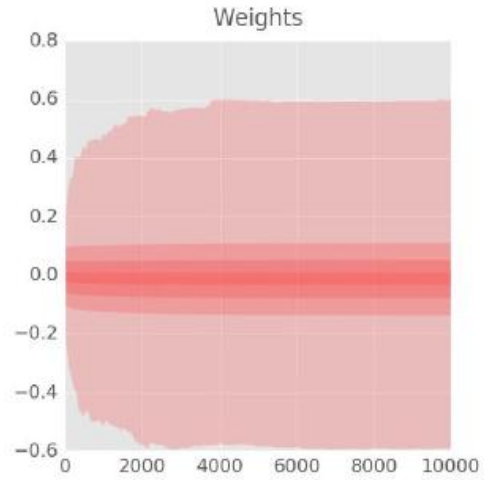
Slides from M. Gorner@youtube

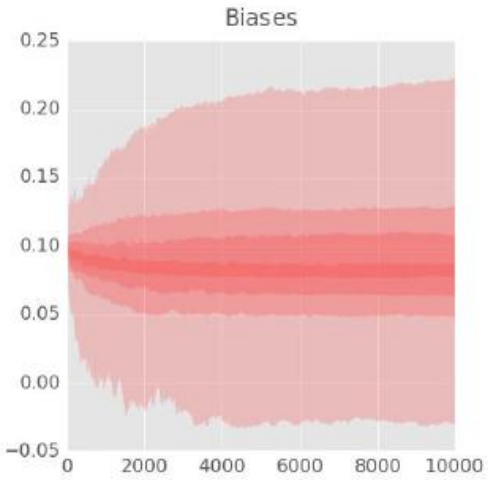
Can do better with conv network





99.3%





References

http://www.deeplearning.book.org

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

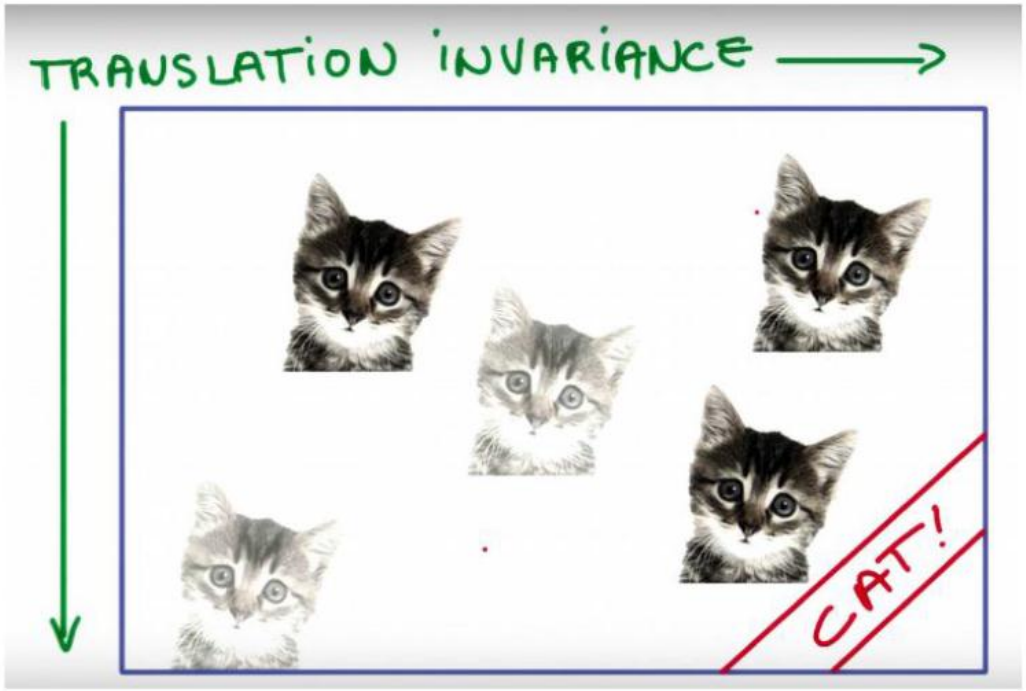
https://www.tensorflow.org/

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

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

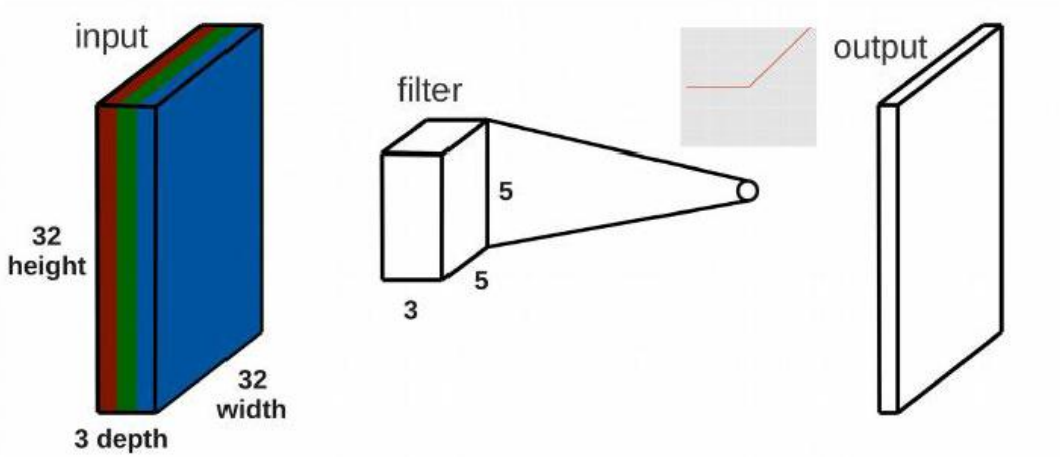
Convolutional Neural Network

- Convolutional NNs are made to exploit local correlation and translation invariance.
- Typical application are image processing and computer vision.



Convolutional Layer

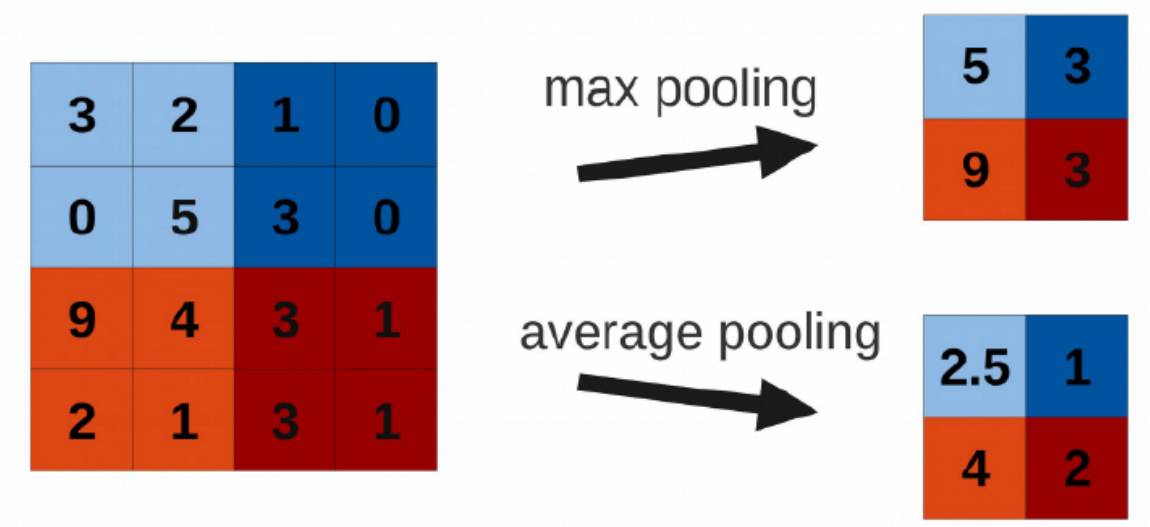
- A small filter (weight tensor) slides over the image to create a feature map.



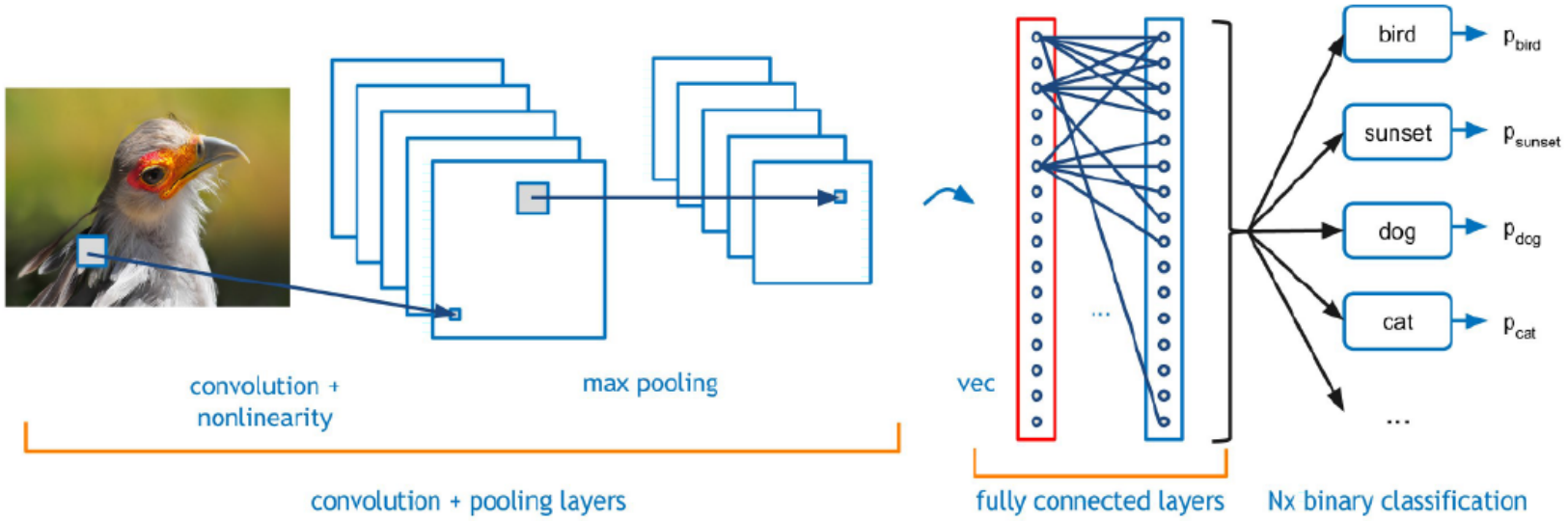
- Several filters could be stacked depth-wise.
- Several convolution can be applied one after the other to extract higher level features.

Average and Max pooling layers

- When output size reduction is required:
  - Max pooling: takes the maximum of each patch.
  - Average pooling: takes the average of each patch.
- Eg: 2x2 filter with stride=2



Convolutional NN architecture

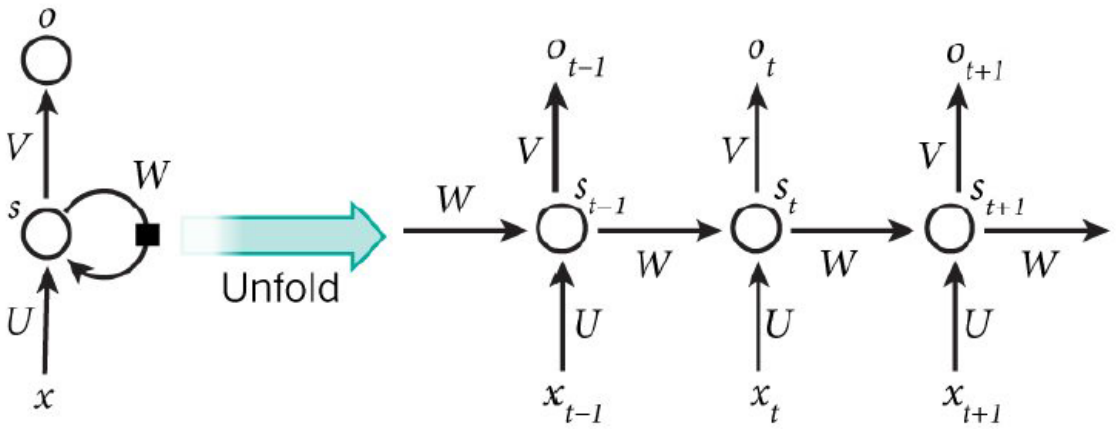


- Convolution and pooling layers to extract features.
- Fully connected layers used at the end to combine features.

- Applications in HEP: PID for neutrino experiments, jet tagging, reduction of seeds for tracking, etc..

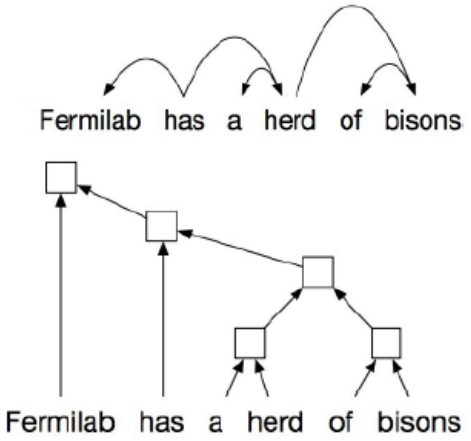
Recursive NN

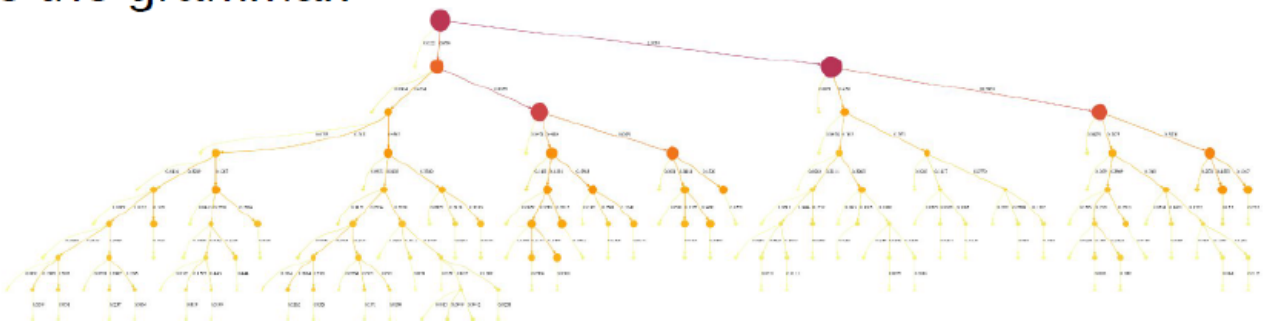
- Recursive NNs are deep NN created by applying the same set of weights recursively over a structured input of variable size.
- They are called recurrent because they performe the same task for every element of a sequence, with the output being depended on the previous computations.
- Typical applications in natural language processing: apply the recursive NN to each word in a sentence for text generation (predict the next word in the sequence), translation, etc..



Recursive NN: HEP application

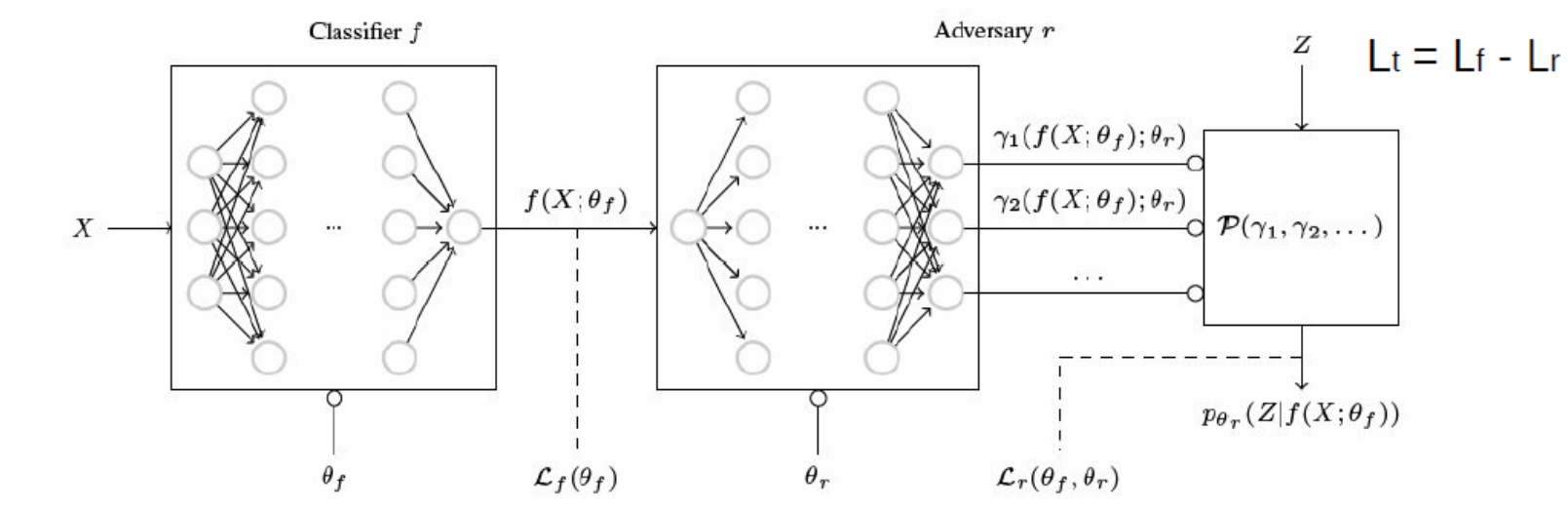
- With particle-flow, collision raw data is converted in a list of particles.
- Complex objects (e.g. jets) are reconstructed by combining particles from this list.
- Image-processing approaches might not be the best in this case.
- Recursive NNs can be better suited.
  - Particles are like words in a sentence.
  - QCD is the grammar.





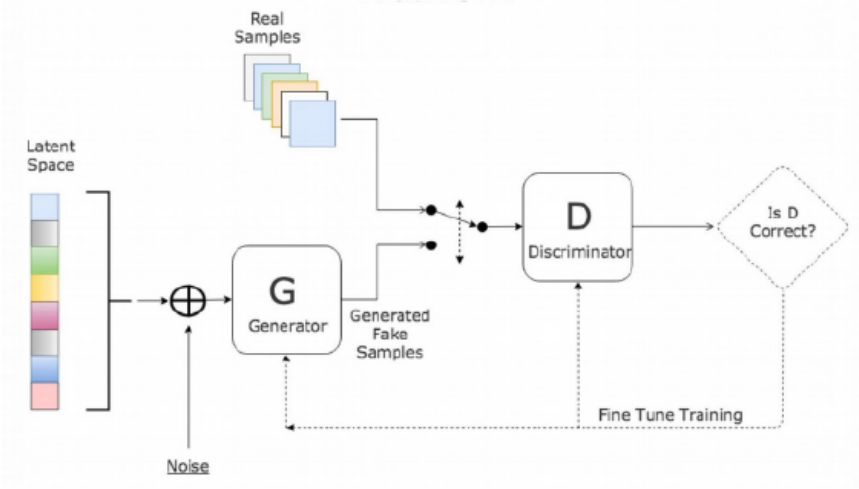
Adversarial NN

- Two deep NNs in competition with each other.
- The first NN can be used to maximize the classification performance of signal against background events.
- The second NN can be trained to identify dependency on systematic uncertanty of the output of the first NN.
- The minimization of the global loss function guarantees optimal classification performance with reduced systematic dependence.



Generative adversarial NN

- Adversarial NNs can also be used for image generation.
- A generator and a discriminator are trained in competition.
  - Generator: creates images starting from noise.
  - Discriminator: tries to distinguish between true and generated images.



- The global loss function is given by Loss(gen) – Loss(discr).
- **The generator learns to make images that it has never seen simply by fooling the discriminator!**
- Application: gen of calo images, jets, and even high-level variables!

Lorentz boost network: motivation

- Deep learning methods using high-level and low-level variables are outperforming shallow learning methods using high-level variables.
  - There is some information in low-level variables that high-level variables is not using.

- Deep learning methods using low-level variables only are not able to reproduce the same results as deep learning methods using also high-level variables.
  - Need of a new NN architecture to fully exploit low-level information and automatize the design of high-level variables.

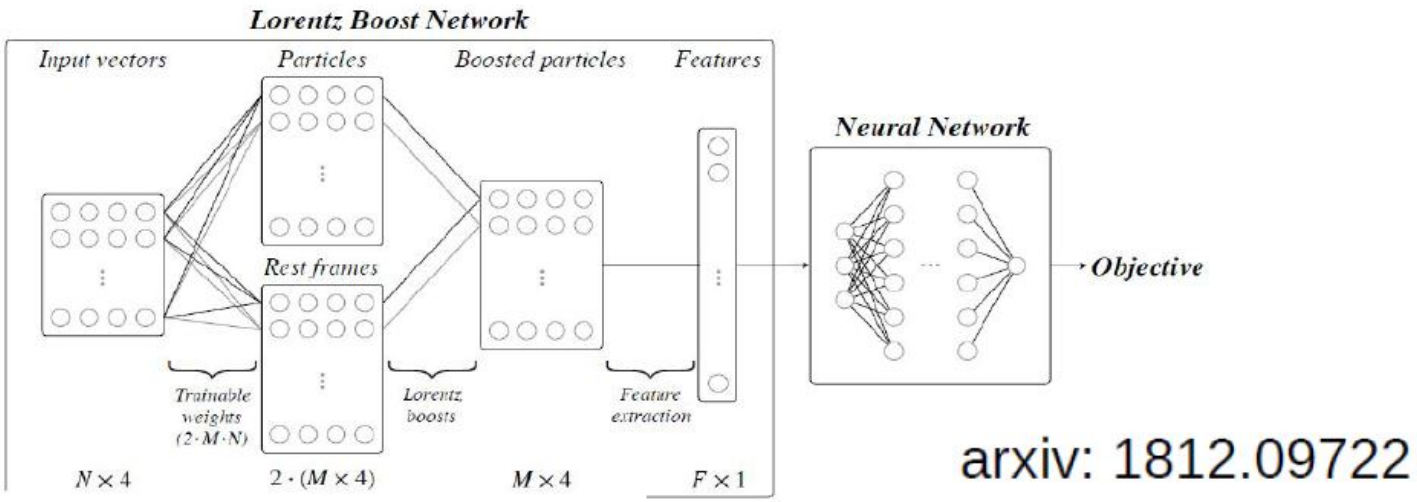
arxiv: 1812.09722

LBN: network architecture

- Two stages approach:

  combines them to form composite particles and rest frames. Composite particles are boosted in the rest frames where features are extracted.

  - An application specific NN uses LBN features as input.

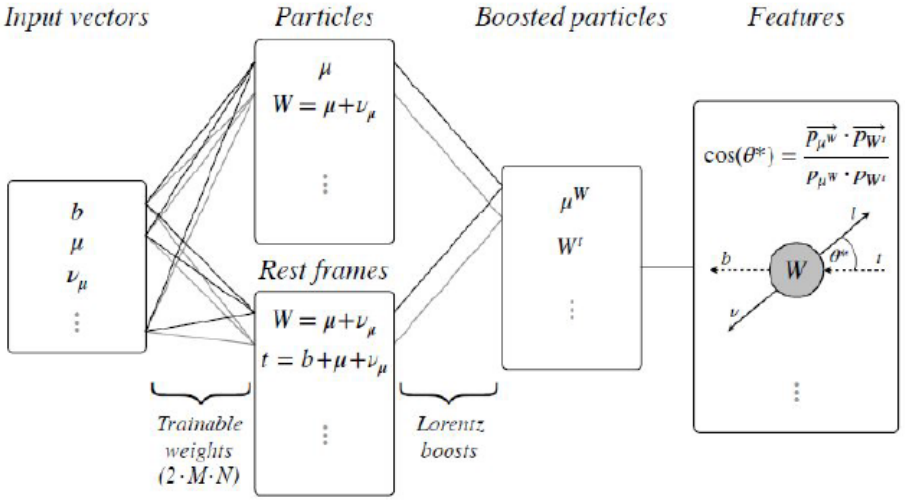


arxiv: 1812.09722

LBN: features extraction

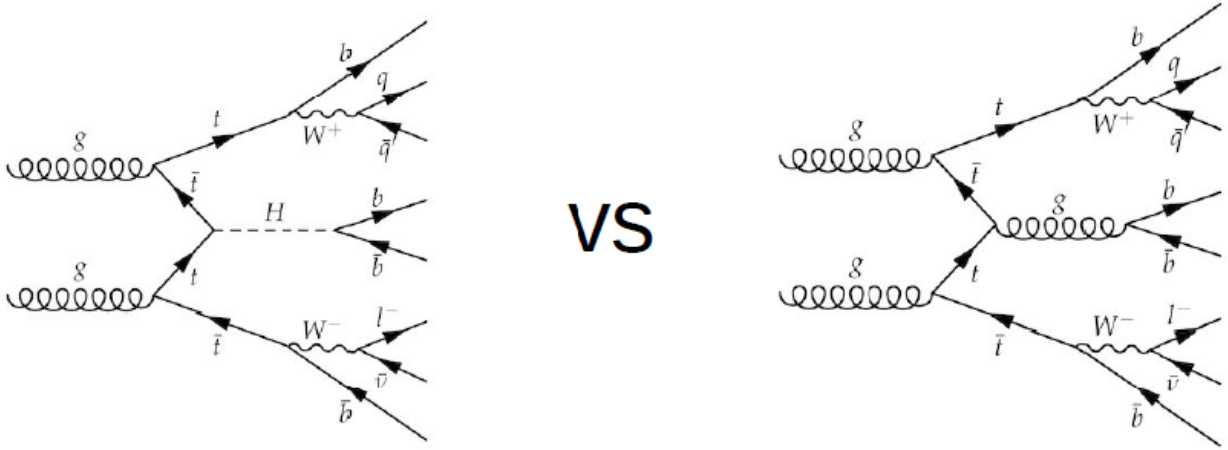
- Extract generic features from boosted particles.
  - Single particle features: E, m, pT, η, ϕ.
  - Pairwise features: such as *cos(θ)* between all pairs.

- E.g. *cos(θ*)* in the semi-leptonic decay of the top quark, defined as the angular difference between the direction of the charged lepton in the *W* rest frame and the direction of the *W* in the top rest frame.

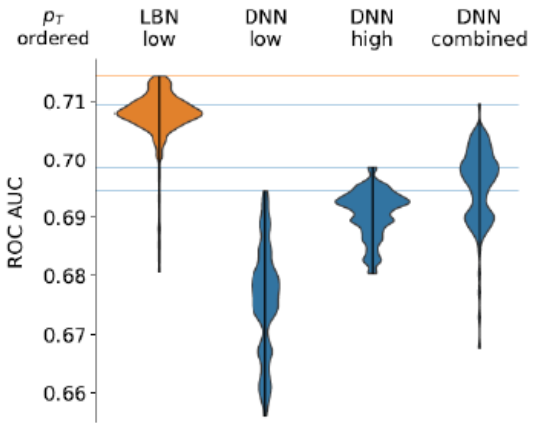


LBN: application

- LBN performance compared to standard DNN with low-, high- and comination of low- and high-level variables.



- LBN shows improved performance in terms of ROC AUC.



Conclusions

- Neural networks are widely used in HEP and will become more and more important.
- A quick overview of the basic structure of the most used NNs in HEP was given.
- New NNs layers, specifically engineered for HEP, were created.
  - In this case, high performance comes also with a good interpretability of the trained parameters.
- NN is a quickly developing field. Exciting time to work on it and to find new applications for HEP.