Lecture 2

neural networks and how to monitor the learning process

Claudia Acquistapace
Istitute for Geophysics and Meteorology
University of Cologne
email: **cacquist@uni-koeln.de**

UNIVERSITY
OF COLOGNE

About the lectures (and me)

Slides of the lectures and additional lectures notes with additional material can be found at:

https://tinyurl.com/teachingUnibo2024

You can always contact me at **cacquist@uni-koeln.de** for questions on the lectures or on whatever (i.e. master thesis, Erasmus, living in Germany etc), I am happy to help you

Info about me

www.claudiaacquistapace.it

and

my Junior research group EXPATS

https://expats-ideas4s.com

We are also on social media, if you want to follow us:

@EXPATS_ideas4s

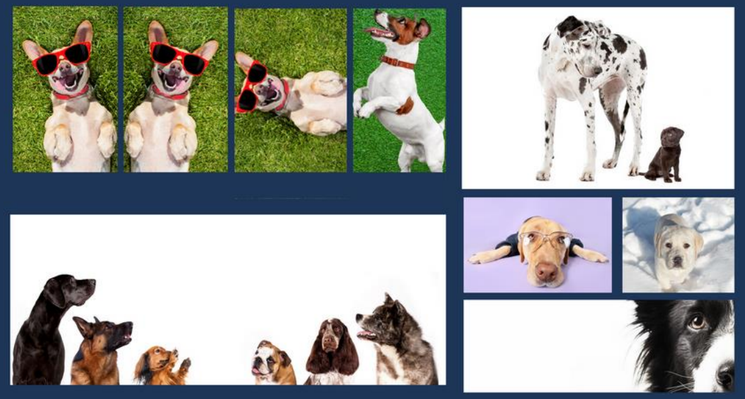
@EXPATS-ideas4s

UNIVERSITY OF COLOGNE

Recap from last week

1

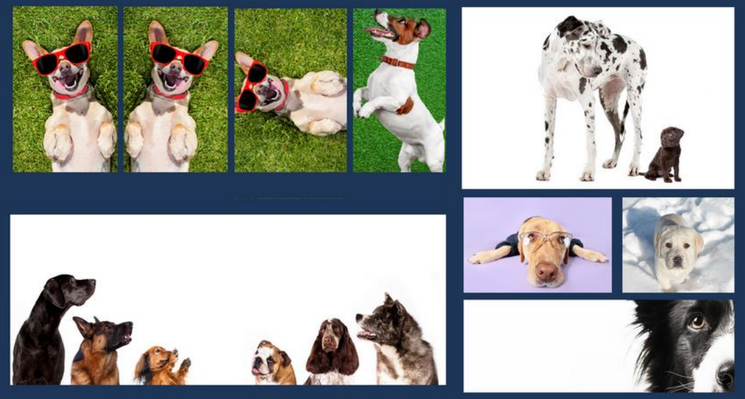
The problem of **image classification** or... assigning a label to an image with a computer



Recap from last week

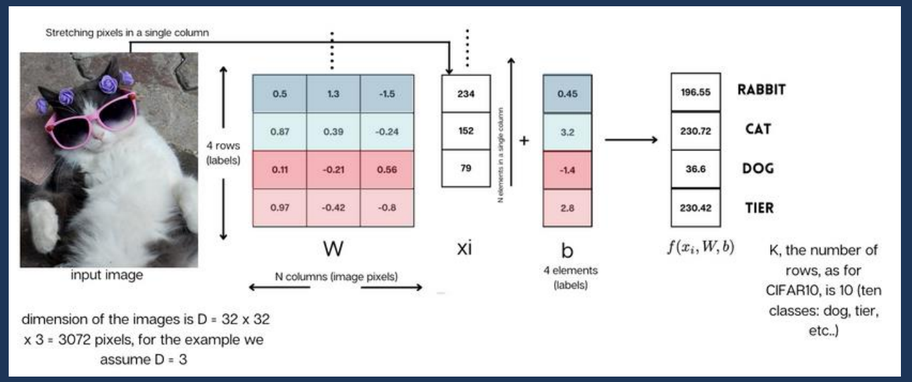
1

The problem of **image classification** or... assigning a label to an image with a computer



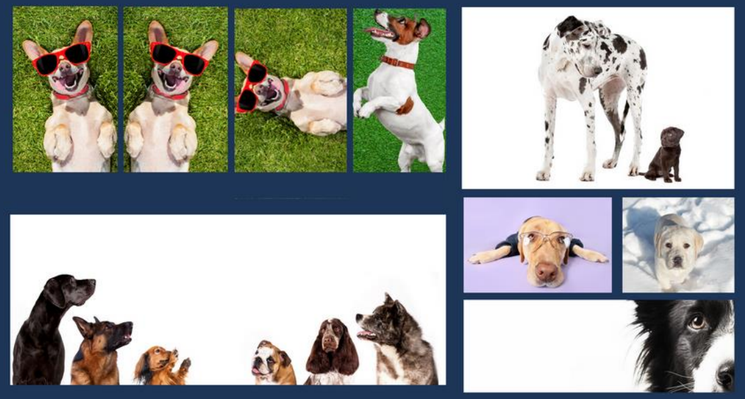
2

The simplest data driven approach: a linear classifier and a loss function

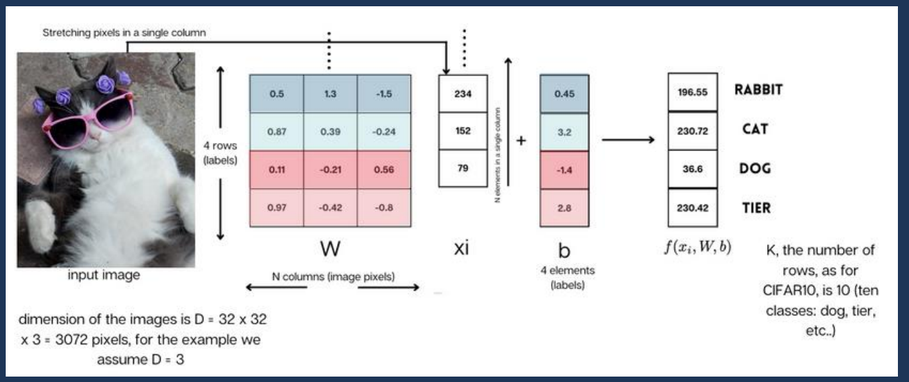


Recap from last week

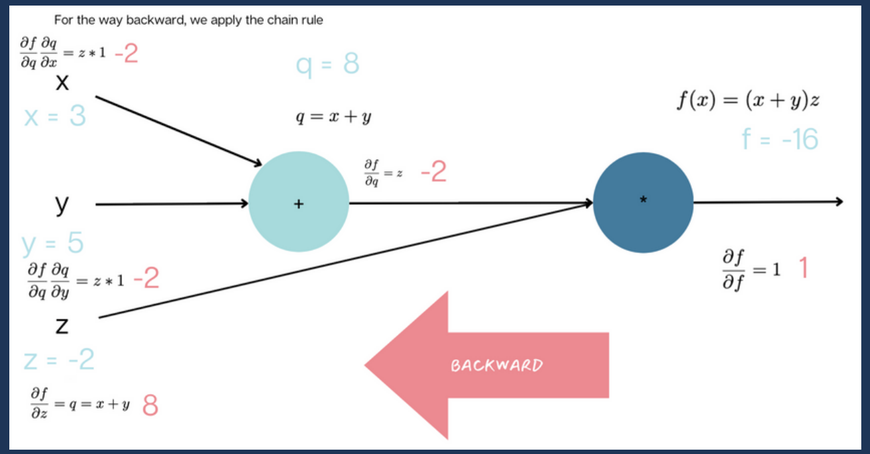
1. The problem of **image classification** or... assigning a label to an image with a computer



2. **The simplest data driven approach**: a linear classifier and a loss function



3. **Learning process via optimization** (and the various processes behind it)



Topics for today

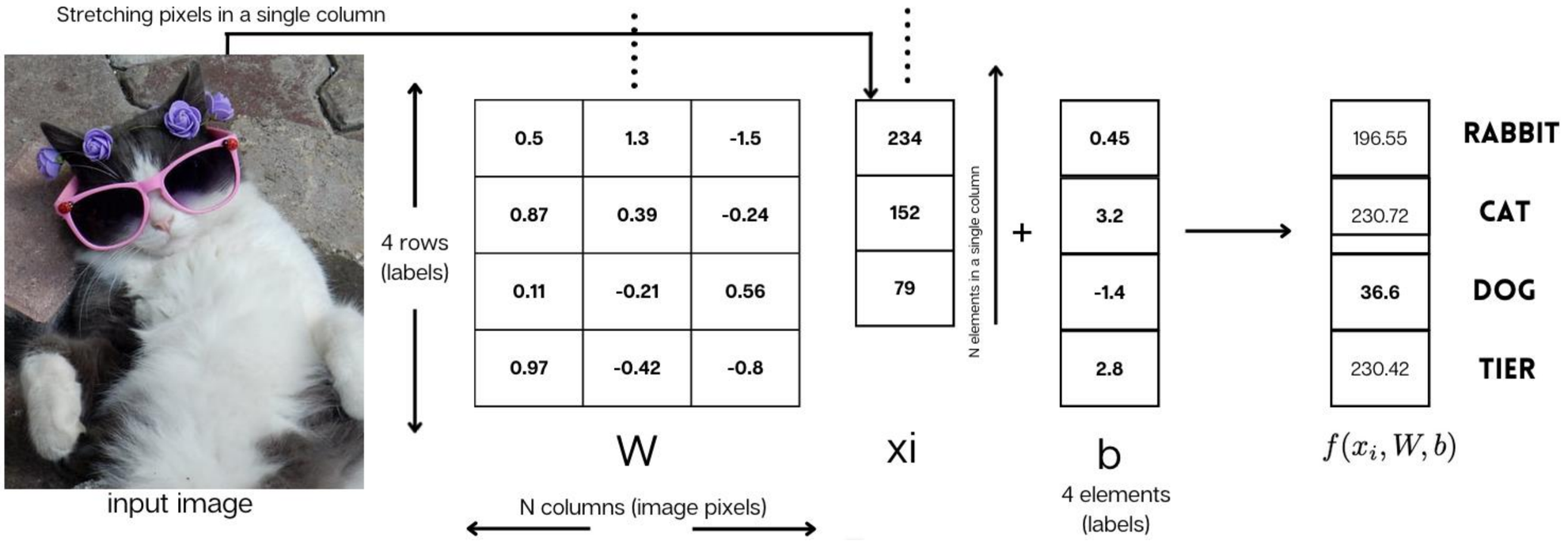
1. Activation functions and multilayer perceptron
2. Artificial neural networks
3. Monitoring the learning process

1

Activation functions and multilayer perceptron

Activation functions and multilayer perceptron

Last time, we discussed a linear classifier computing scores associated to input images as



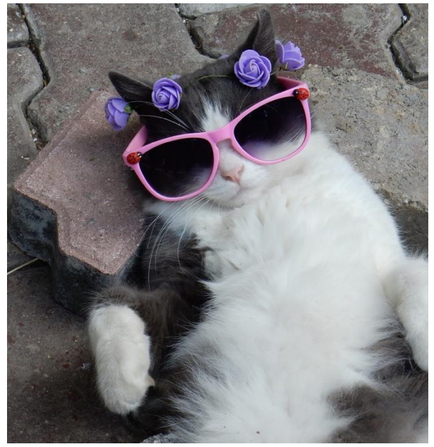
$$f(x_i, W, b) = Wx_i + b$$

$$f(x_i, W) = Wx_i$$

Multilayer perceptron and neural networks

How can we construct a simple neural network starting from our linear model?

$$f(x_i, W, b) = Wx_i + b$$



32 x 32 x 3

3072 in total

3072 x 1

10 x

$f(x_i, W, b)$

10 numbers giving class scores

10 x 1

W

parameters or weights

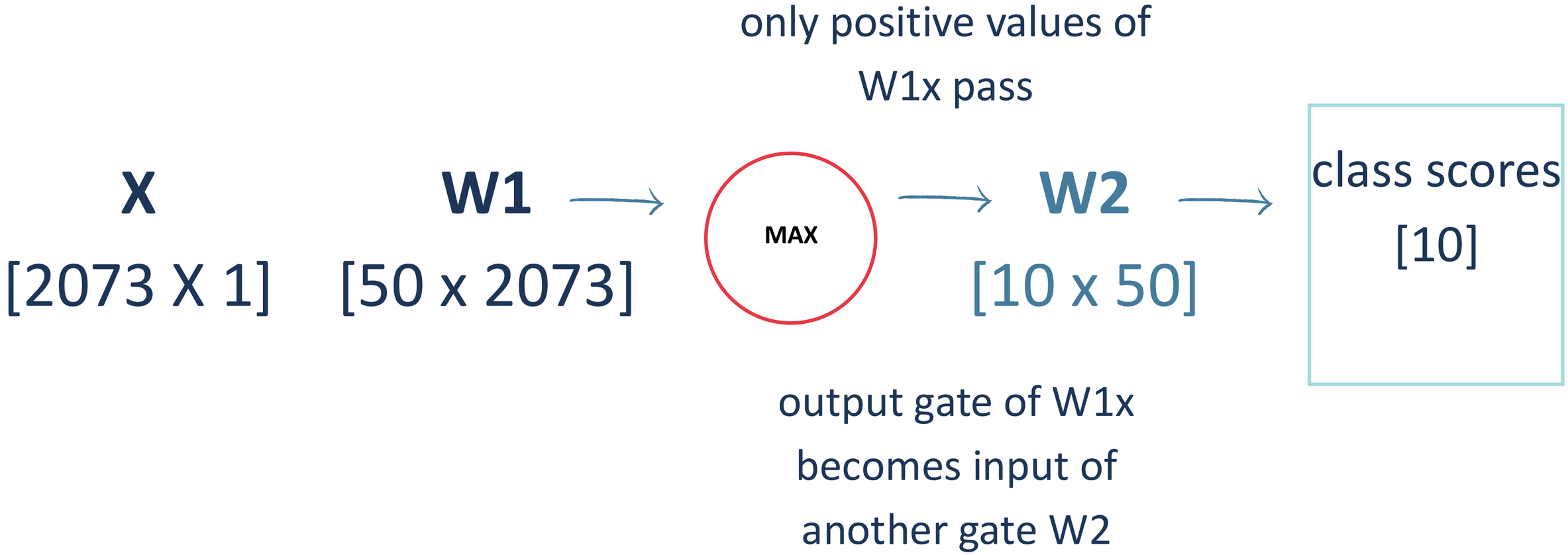
10 x 3072

To build a 2 layer neural network, we introduce a **non-linearity** in this system, for example:

$$s = W_2 max(0, W_1 x)$$

To build a 2 layer neural network, we introduce a **non-linearity** in this system, for example:

$$s = W_2 max(0, W_1 x)$$



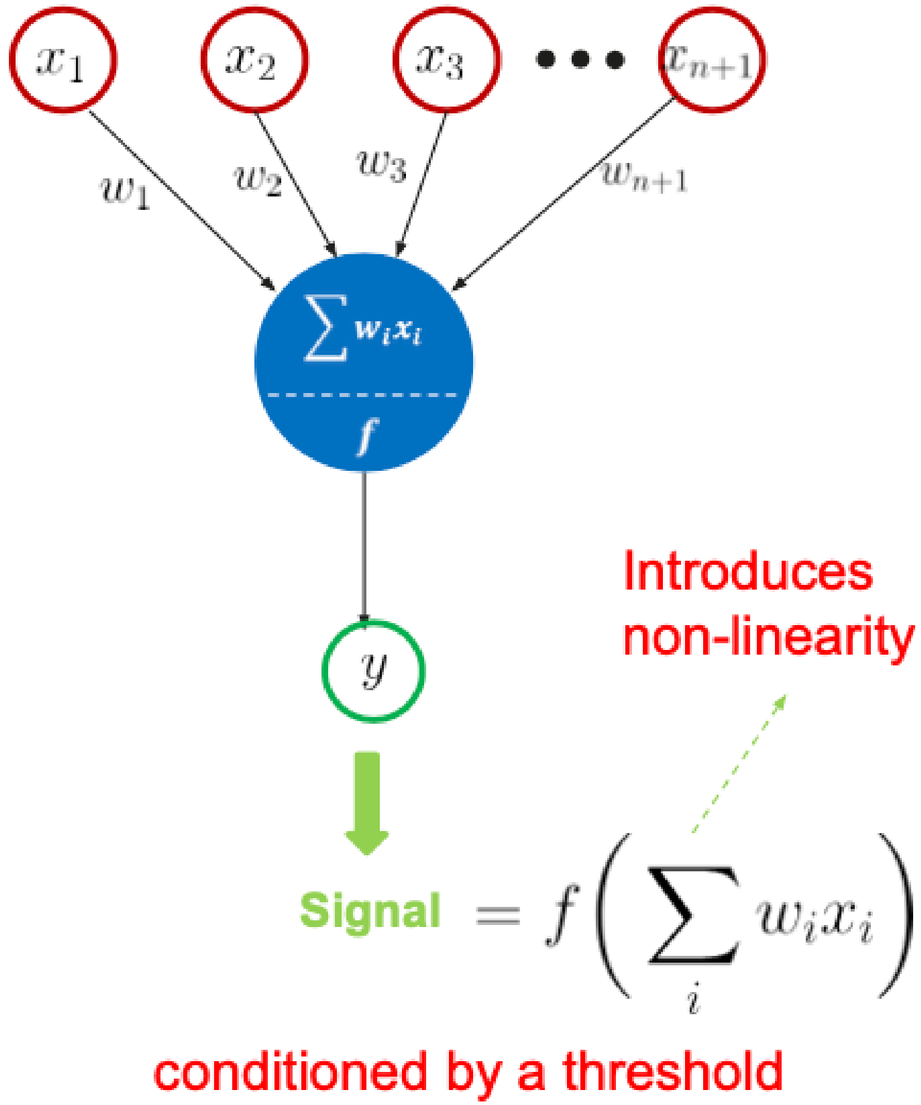
Activation functions

The non-linearity introduced is often called "**activation function**".

These functions take the input number and apply some type of mathematical operations on it.

They introduce a non-linearity which is conditioned by a threshold value

What are typical functional forms for f?



Activation functions

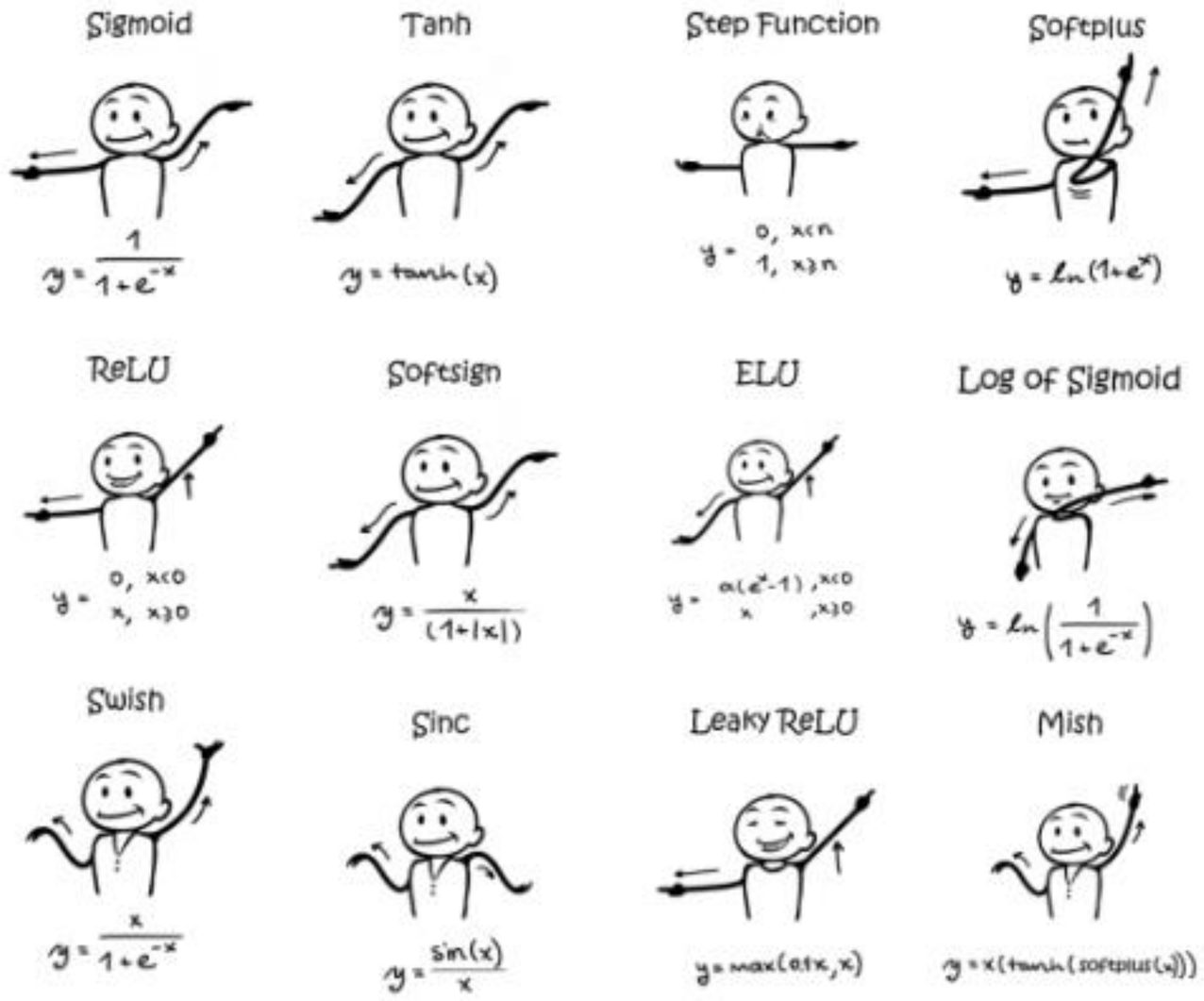
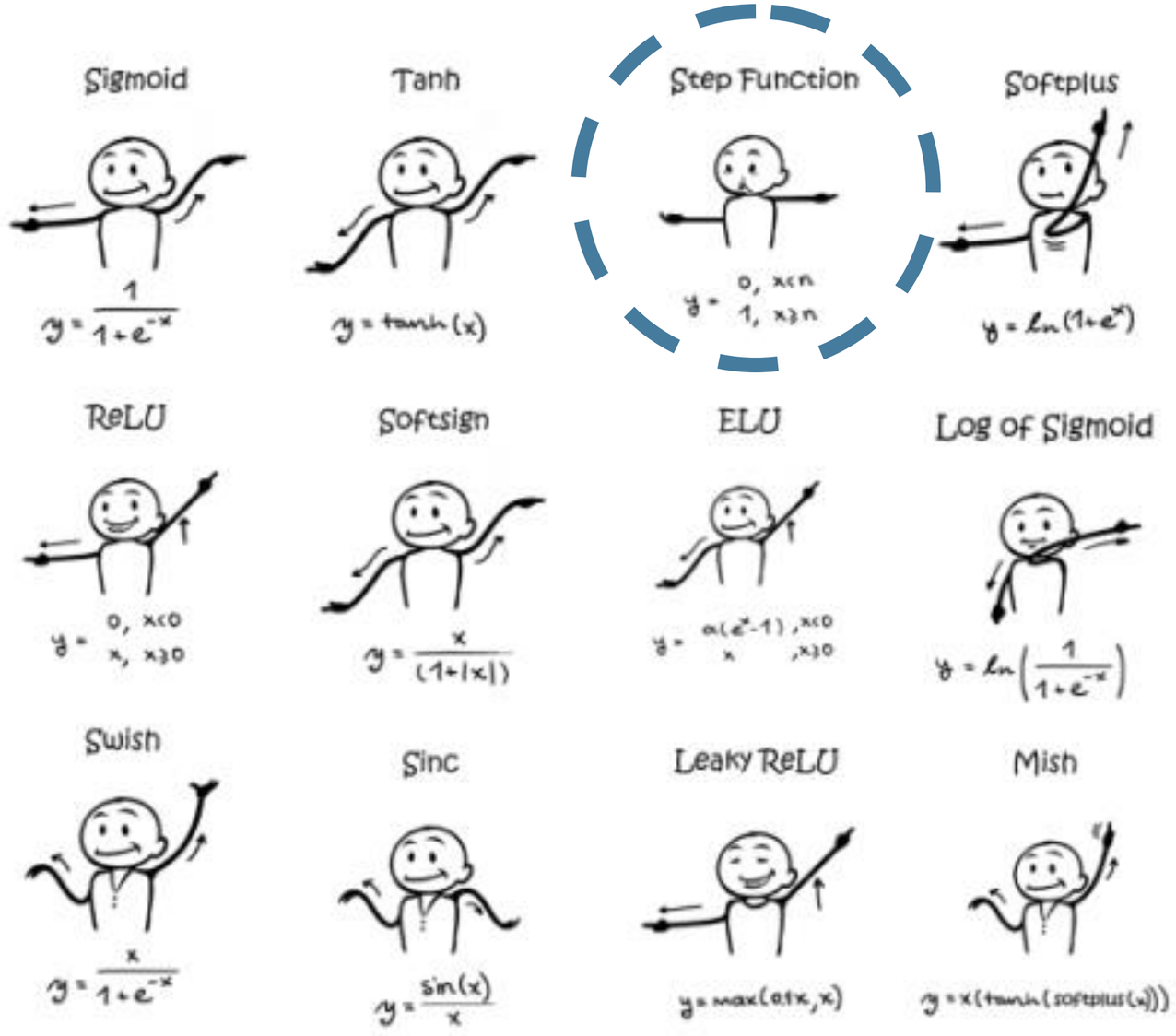
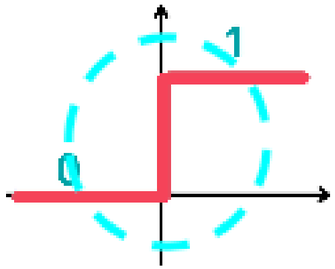


Figure source: https://sefiks.com/tag/activation-function/

Activation functions

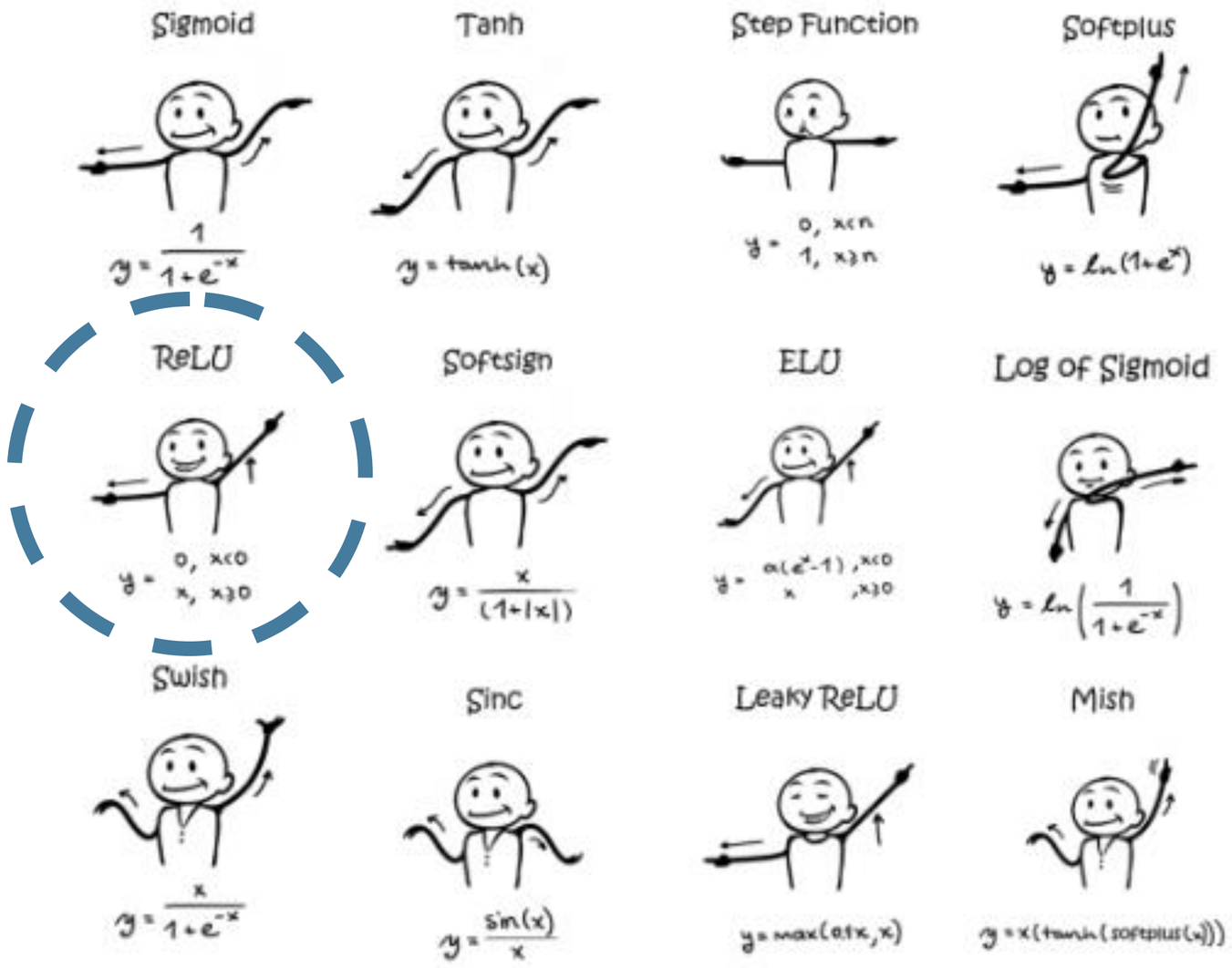


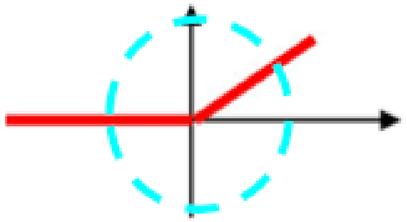


Threshold: x >=0

Figure source: https://sefiks.com/tag/activation-function/

Activation functions

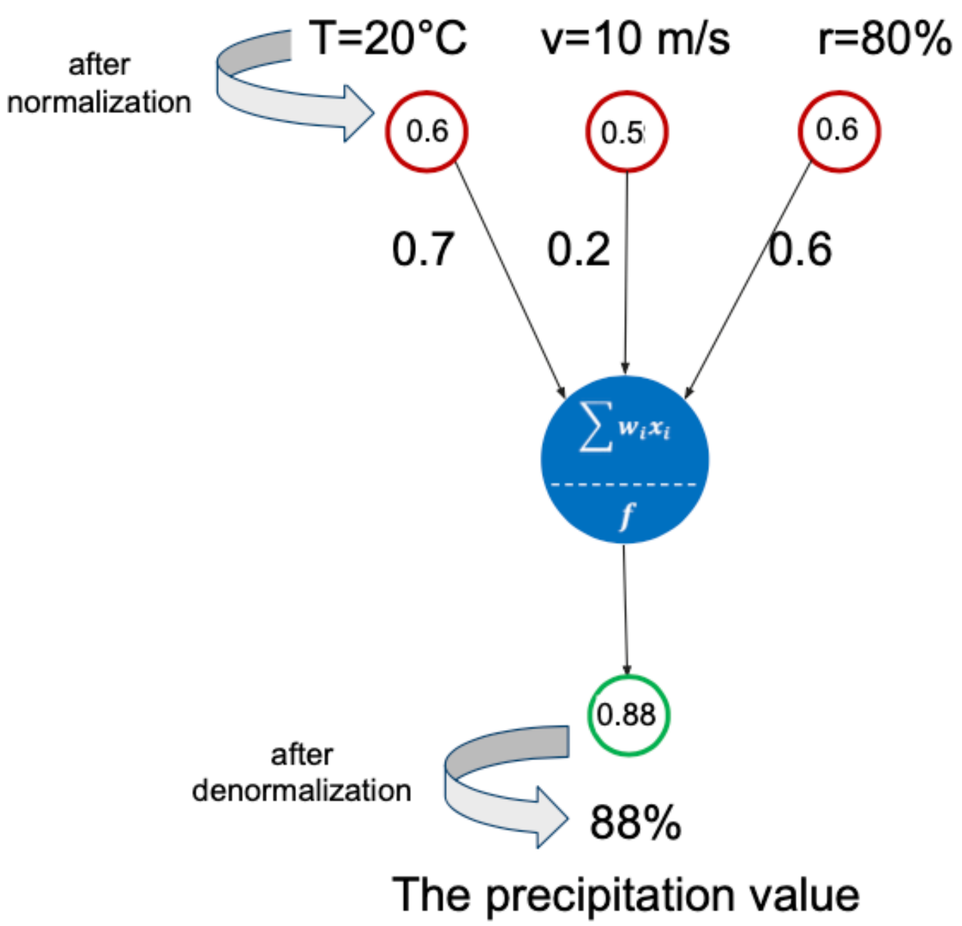




Threshold: x >= 0

Figure source: https://sefiks.com/tag/activation-function/

Example of RELU in action



Activation function formula

$$f\left(\sum_i w_i x_i\right) \Rightarrow ReLU = max\left(0, \sum_i w_i x_i\right)$$

Compute weighted sum

$$\sum_i w_i x_i = (w_1 \times x_1) + (w_2 \times x_2) + (w_3 \times x_3)$$

$$= 0.7 \times 0.6 + 0.2 \times 0.5 + 0.6 \times 0.6$$

$$= 0.88$$

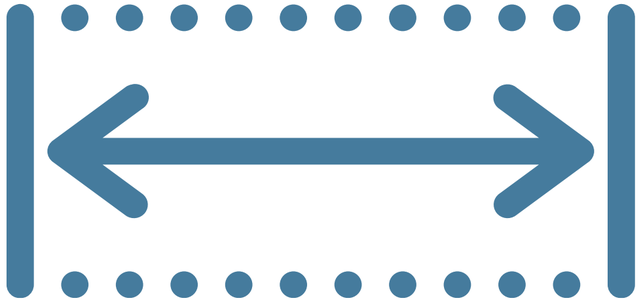
Output of this neutron

$$f\left(\sum_i w_i x_i\right) = max(0, 0.88) = 0.88$$

Figure courtesy of Asma Semcheddine and Prof. Dr. Martin Schulz, Jülich Forschungzentrum

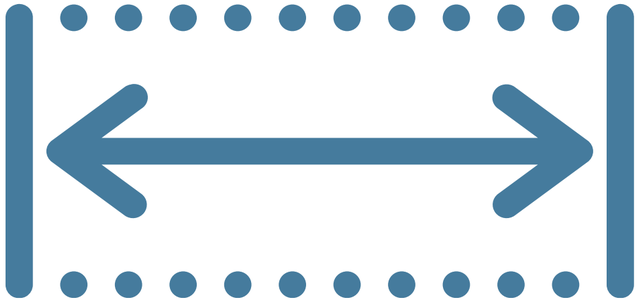
Why do we need them?

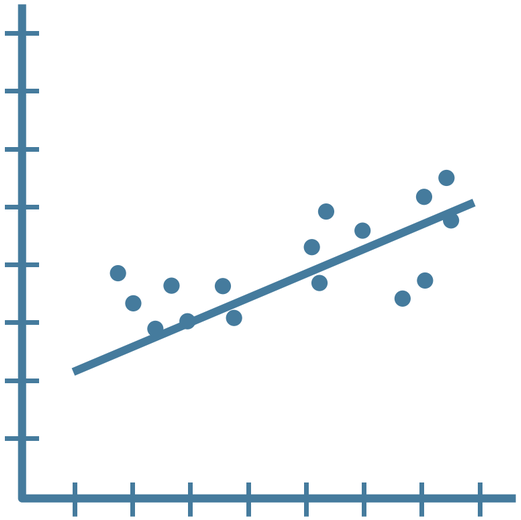
1) They help in **keeping the output of a neuron within a certain range** we decide based on our needs, avoiding computational issues caused by numbers growing to extremely large values in the network.



Why do we need them?

1) They help in **keeping the output of a neuron within a certain range** we decide based on our needs, avoiding computational issues caused by numbers growing to extremely large values in the network.

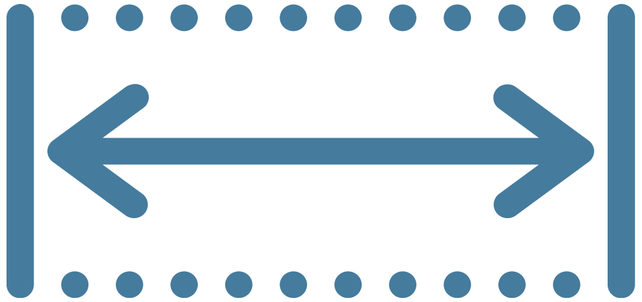


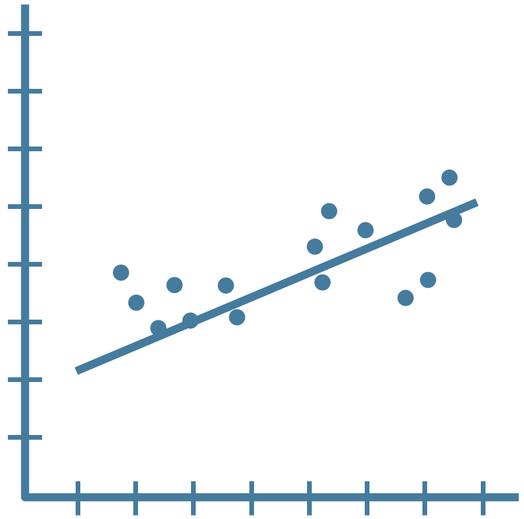


2) They **add non-linearity to the network**. If the model needs to learn non-linear patterns, like for example in classification tasks, then specific non-linear layers need to be added to the network.

Why do we need them?

1) They help in **keeping the output of a neuron within a certain range** we decide based on our needs, avoiding computational issues caused by numbers growing to extremely large values in the network.





2) They **add non-linearity to the network**. If the model needs to learn non-linear patterns, like for example in classification tasks, then specific non-linear layers need to be added to the network.

During backpropagation (gradient descent), gradients get multiplied with the activation functions. If the activation functions re-scale the input into a range of values between 0 and 1, this means that **values of the gradients get strongly reduced**.
In general, gradients tend to vanish because of the depth of the network and this problem goes under the name of "**vanishing gradient problem".**

One slide on history

It all started with a model of the brain... (McCulloch and Pitts)
Linear model with positive/negative outputs given input and weights

1940'

The model was not learning unless you fixed weights

source: Carolina Bento, https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141

One slide on history

It all started with a model of the brain... (McCulloch and Pitts) **Linear model** with positive/negative outputs given input and weights

1940'

The model was not learning unless you fixed weights

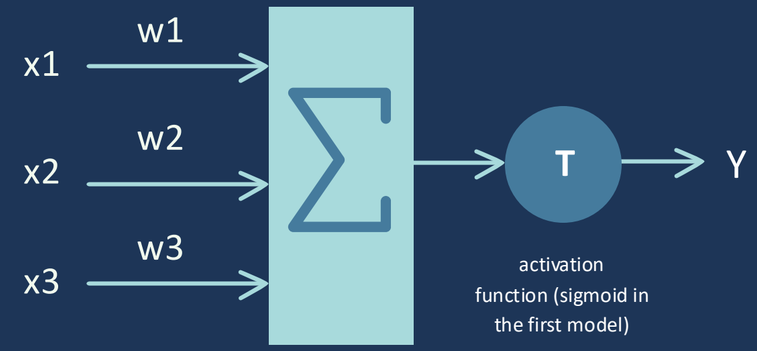
Rosenblatt extended the model to learn weights to generate the output: he created **the perceptron, initially intended for image recognition,** people think it could represent any circuit and logic.

1950'

source: Carolina Bento, https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141

One slide on history

Perceptron combines inputs in a sum and, if the weighted sum exceeds a threshold, the neuron produces the output.



Perceptron is used for binary classification with such descrete output

1940'

It all started with a model of the brain... (McCulloch and Pitts) **Linear model** with positive/negative outputs given input and weights

1950'

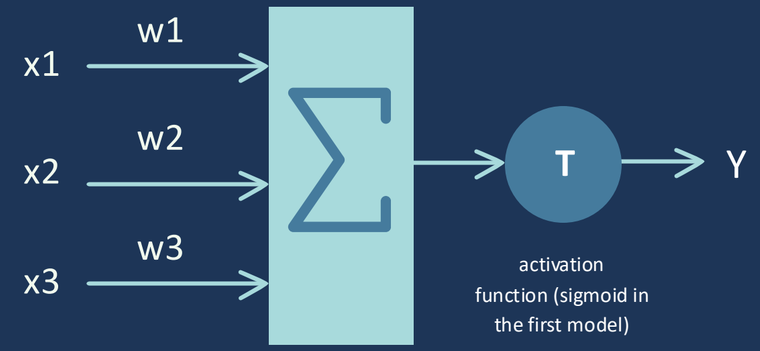
Rosenblatt extended the model to learn weights to generate the output: he created **the perceptron, initially intended for image recognition,** people think it could represent any circuit and logic.

The model was not learning unless you fixed weights

source: Carolina Bento, https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141

One slide on history

Perceptron combines inputs in a sum and, if the weighted sum exceeds a threshold, the neuron produces the output.



Perceptron is used for binary classification with such descrete output

1940'

It all started with a model of the brain... (McCulloch and Pitts) **Linear model** with positive/negative outputs given input and weights

1950'

Rosenblatt extended the model to learn weights to generate the output: he created **the perceptron, initially intended for image recognition,** people think it could represent any circuit and logic.

The model was not learning unless you fixed weights

1969'

Minsky and Papert showed it couldn't represent the XOR gate, and highlighted the fact that Perceptron, with only one neuron, can't be applied to non-linear data.

source: Carolina Bento, https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141

Multi-layer perceptron

The Multilayer Perceptron was developed to tackle this limitation.

It is a neural network where the **mapping between inputs and output is non-linear**.

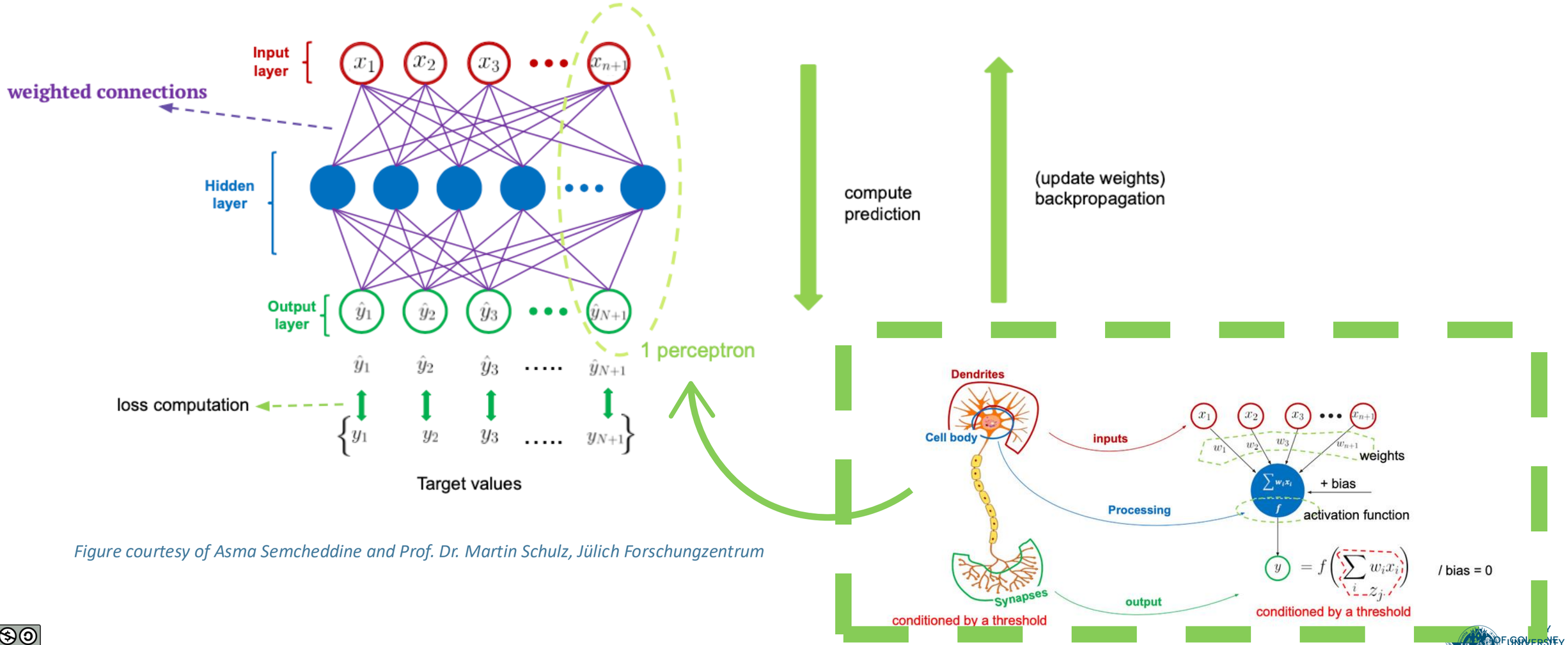
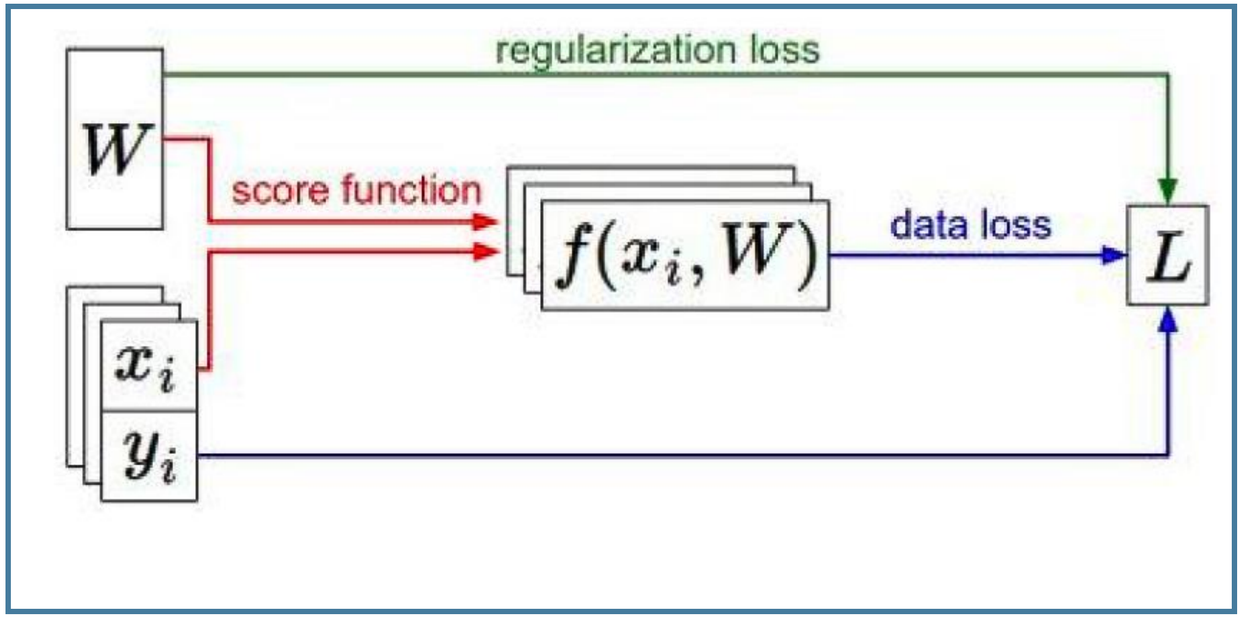
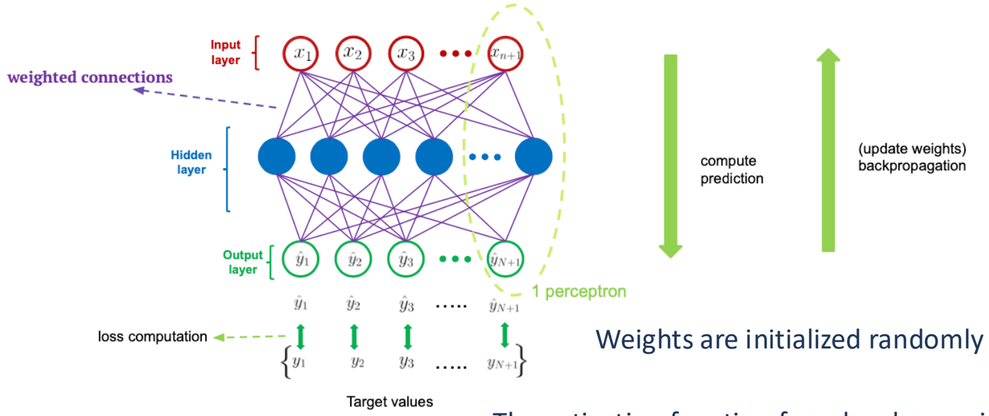


Figure courtesy of Asma Semcheddine and Prof. Dr. Martin Schulz, Jülich Forschungzentrum

Model set up



The Multilayer Perceptron is a neural network where the **mapping between inputs and output is non-linear** and it is our **score function**

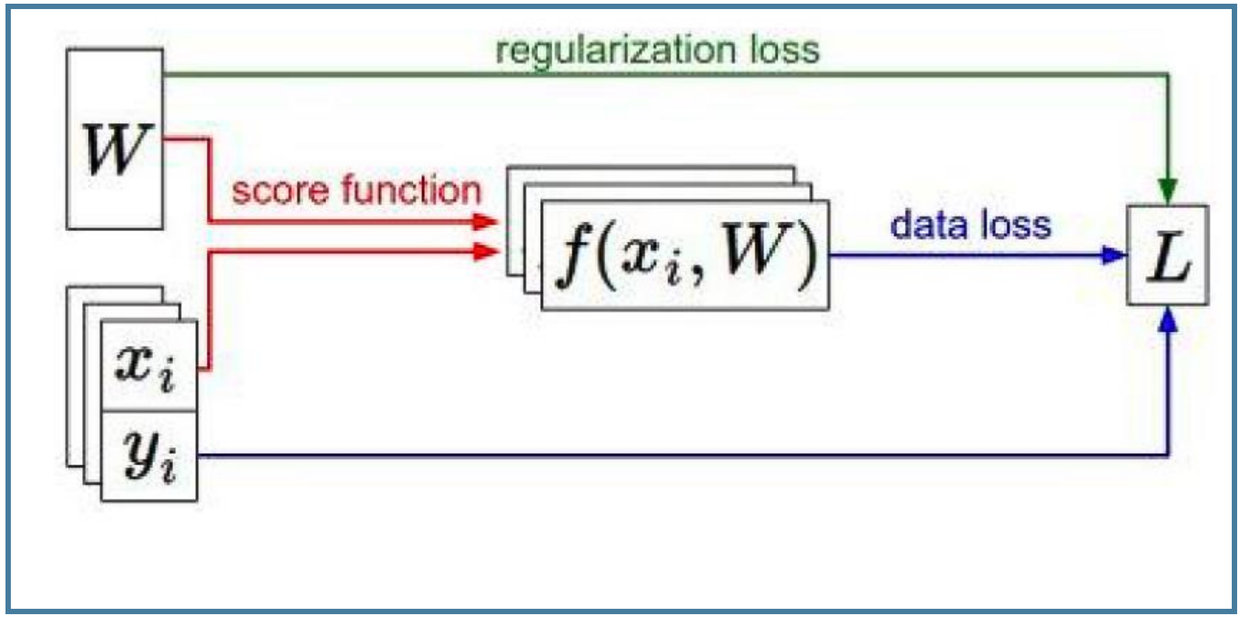


Weights are initialized randomly

The activation function f can be chosen, in our case we take RELU

The multilayer perceptron gives us a score y

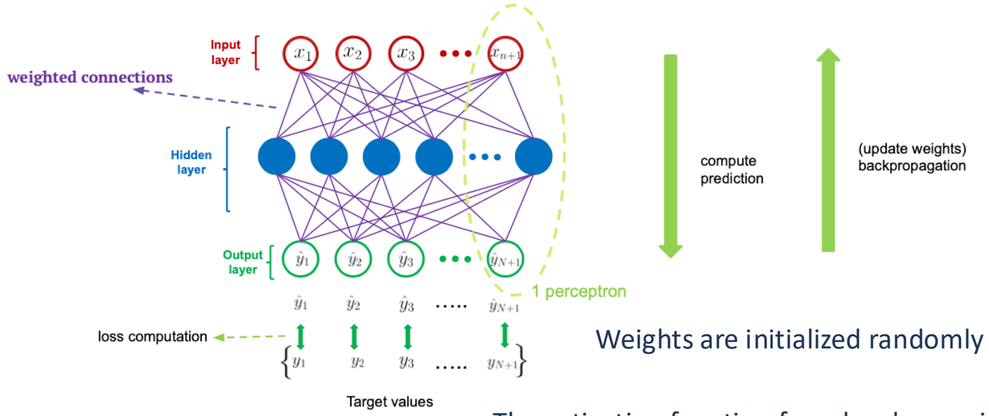
Model set up



We define as **loss function** the Mean Squared error

$$MSE = \frac{1}{N}\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2$$

The Multilayer Perceptron is a neural network where the **mapping between inputs and output is non-linear** and it is our **score function**

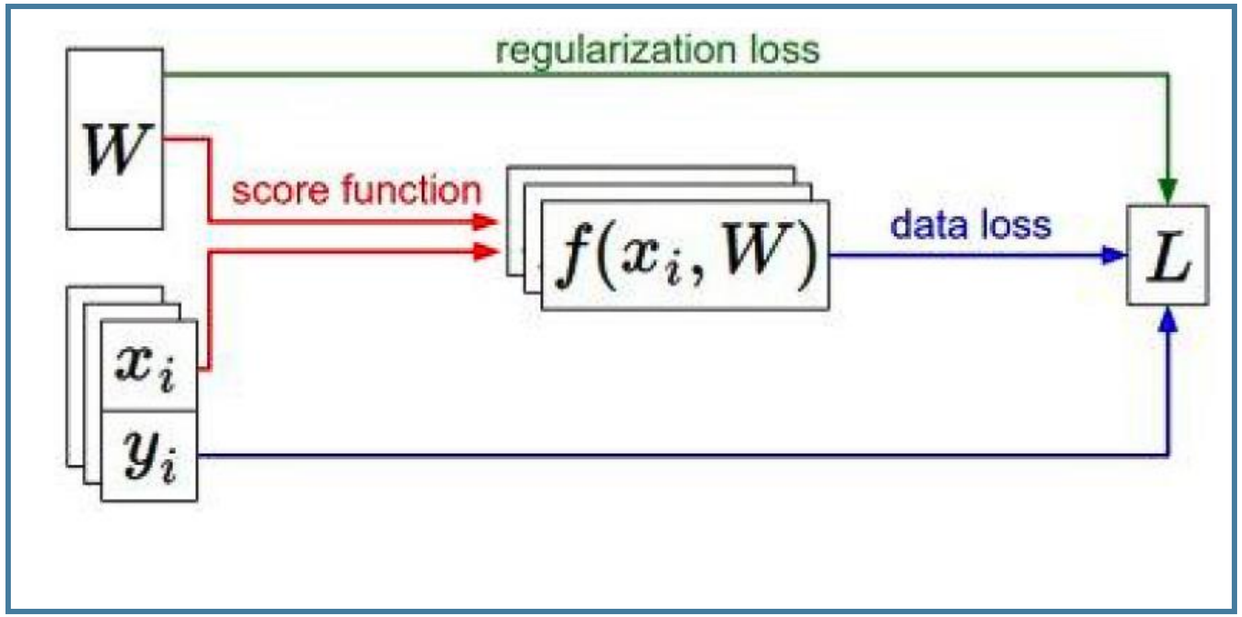


Weights are initialized randomly

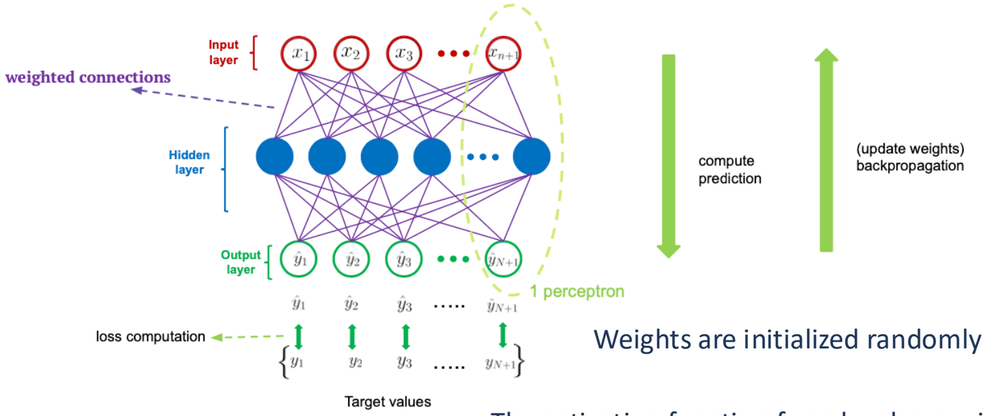
The activation function f can be chosen, in our case we take RELU

The multilayer perceptron gives us a score y

Model set up



The Multilayer Perceptron is a neural network where the **mapping between inputs and output is non-linear** and it is our **score function**



Weights are initialized randomly

The activation function f can be chosen, in our case we take RELU

The multilayer perceptron gives us a score y

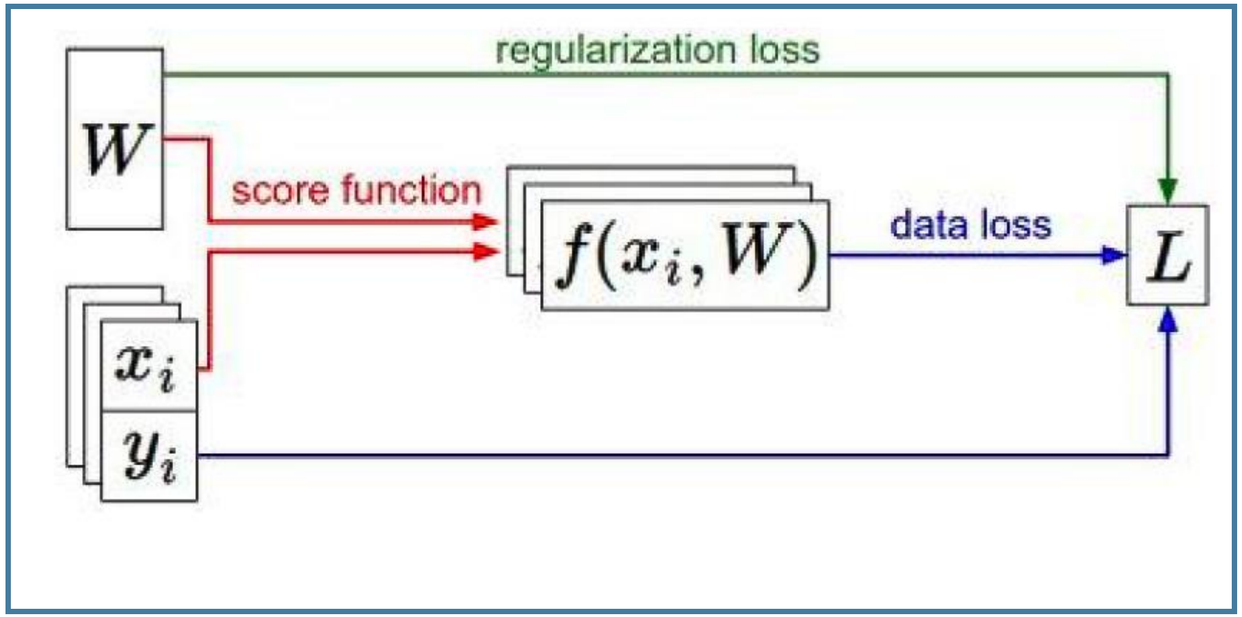
We define as **loss function** the Mean Squared error

$$MSE = \frac{1}{N}\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2$$

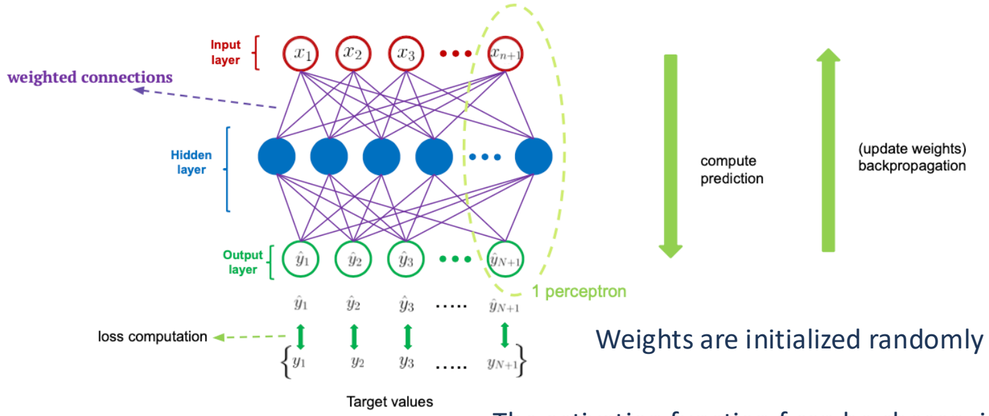
We add the regularisation term (**regularization loss**)

$$\lambda R(W)$$

Model set up



The Multilayer Perceptron is a neural network where the **mapping between inputs and output is non-linear** and it is our **score function**



Weights are initialized randomly

The activation function f can be chosen, in our case we take RELU

The multilayer perceptron gives us a score y

We define as **loss function** the Mean Squared error

$$MSE = \frac{1}{N}\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2$$

We add the regularisation term (**regularization loss**)

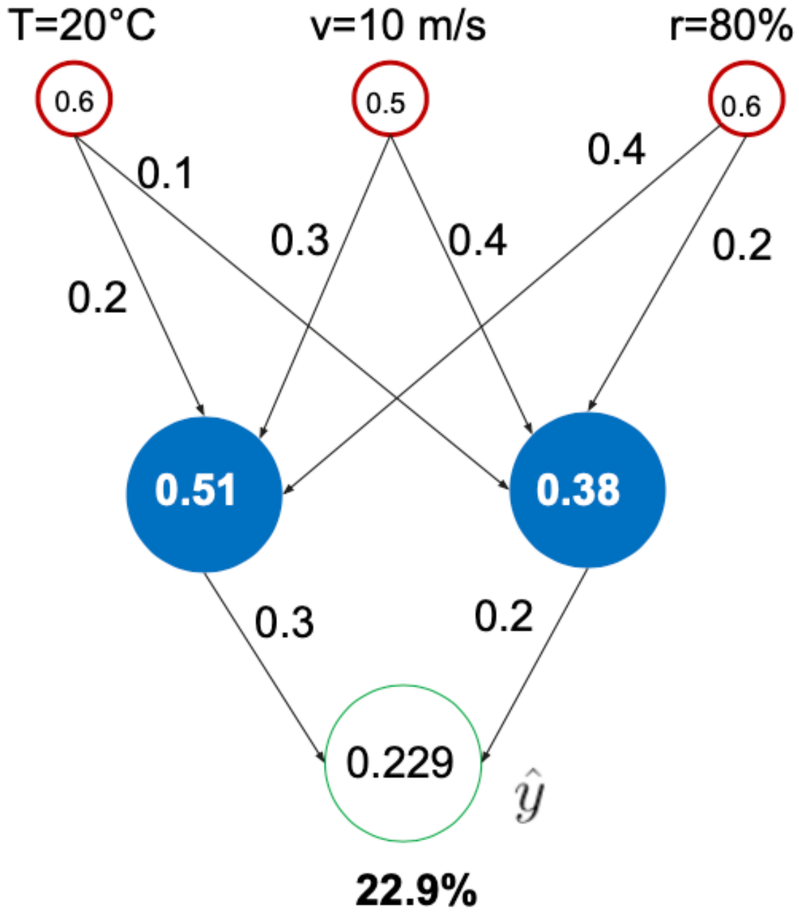
$$\lambda R(W)$$

Weights are learned via **backpropagation** following **gradient descent**.

What does that mean? calculating the gradient of the Mean Squared error across all input/output pairs and unpdating weights with the values obtained with the gradients,

Example of loss computation and back propagation

Immagine to have a neural network with 3 input nodes, 2 hidden nodes, and 1 output node to forecast precipitation based on temperature, wind speed and relative humidity. Our target probability of precipitation is 88% with such values (or y label)

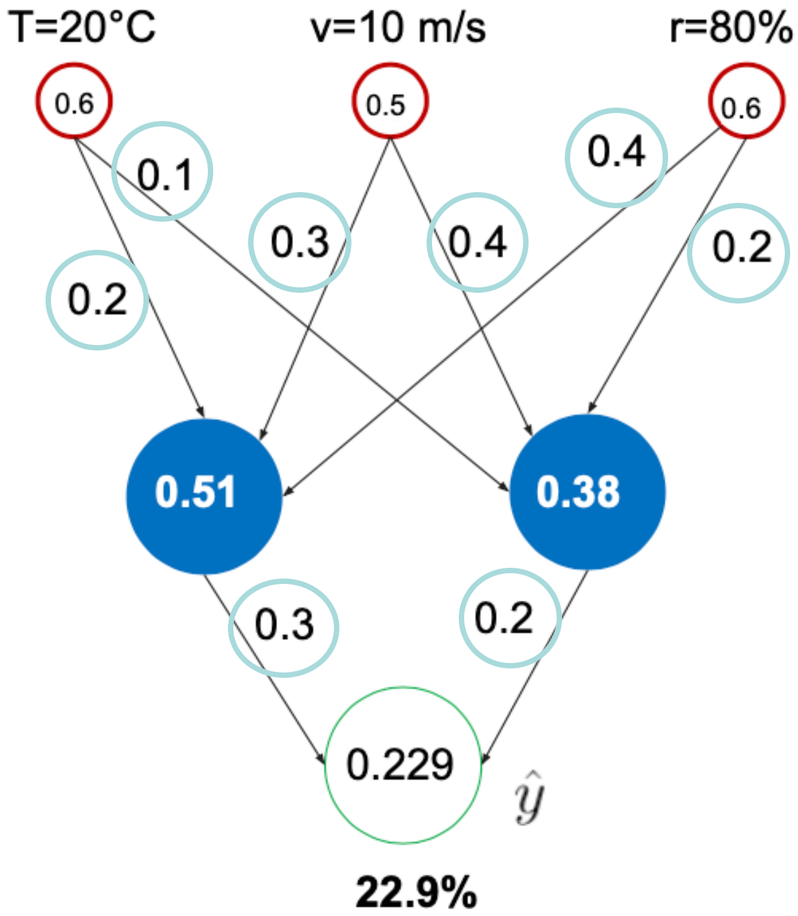


normalized input values

Figure courtesy of Asma Semcheddine and Prof. Dr. Martin Schulz, Jülich Forschungzentrum

Example of loss computation and back propagation

Immagine to have a neural network with 3 input nodes, 2 hidden nodes, and 1 output node to forecast precipitation based on temperature, wind speed and relative humidity. Our target probability of precipitation is 88% with such values (or y label)



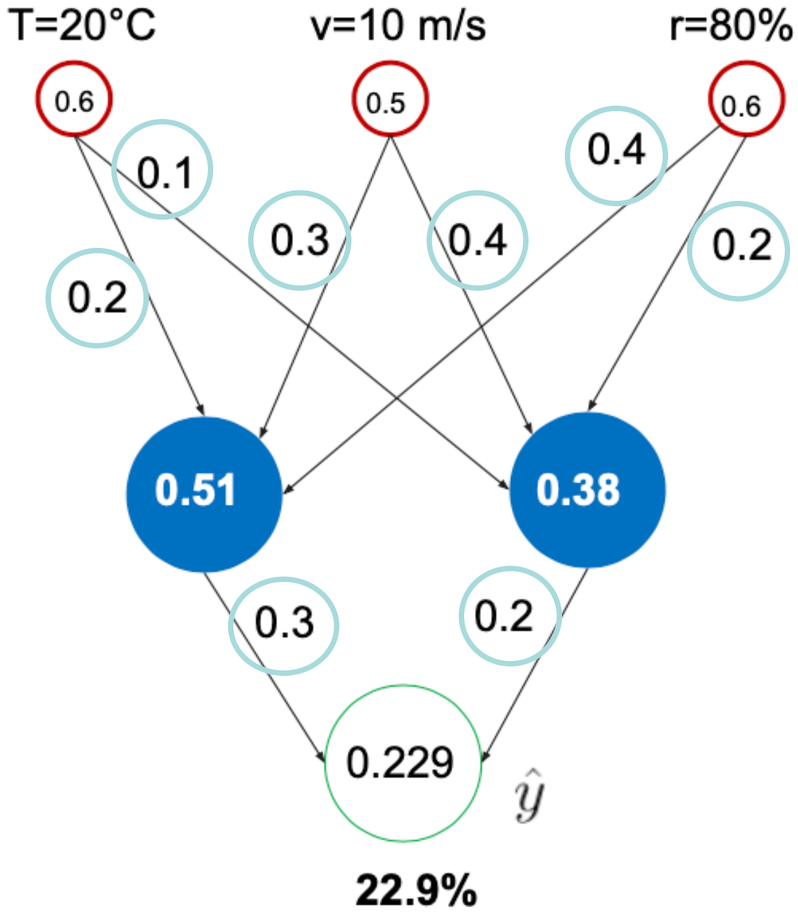
normalized input values

randomly initialized weights Wi

Figure courtesy of Asma Semcheddine and Prof. Dr. Martin Schulz, Jülich Forschungzentrum

Example of loss computation and back propagation

Immagine to have a neural network with 3 input nodes, 2 hidden nodes, and 1 output node to forecast precipitation based on temperature, wind speed and relative humidity. Our target probability of precipitation is 88% with such values (or y label)



normalized input values

randomly initialized weights Wi

Calculating the loss (mean squared error)

$$MSE = \frac{1}{N}\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2$$

we get 0.2119 which is too big! This is because the value we obtain, 22.9% is too far from the value expected of 88%.

Figure courtesy of Asma Semcheddine and Prof. Dr. Martin Schulz, Jülich Forschungzentrum

Example of loss computation and back propagation

Immagine to have a neural network with 3 input nodes, 2 hidden nodes, and 1 output node to forecast precipitation based on temperature, wind speed and relative humidity. Our target probability of precipitation is 88% with such values (or y label)

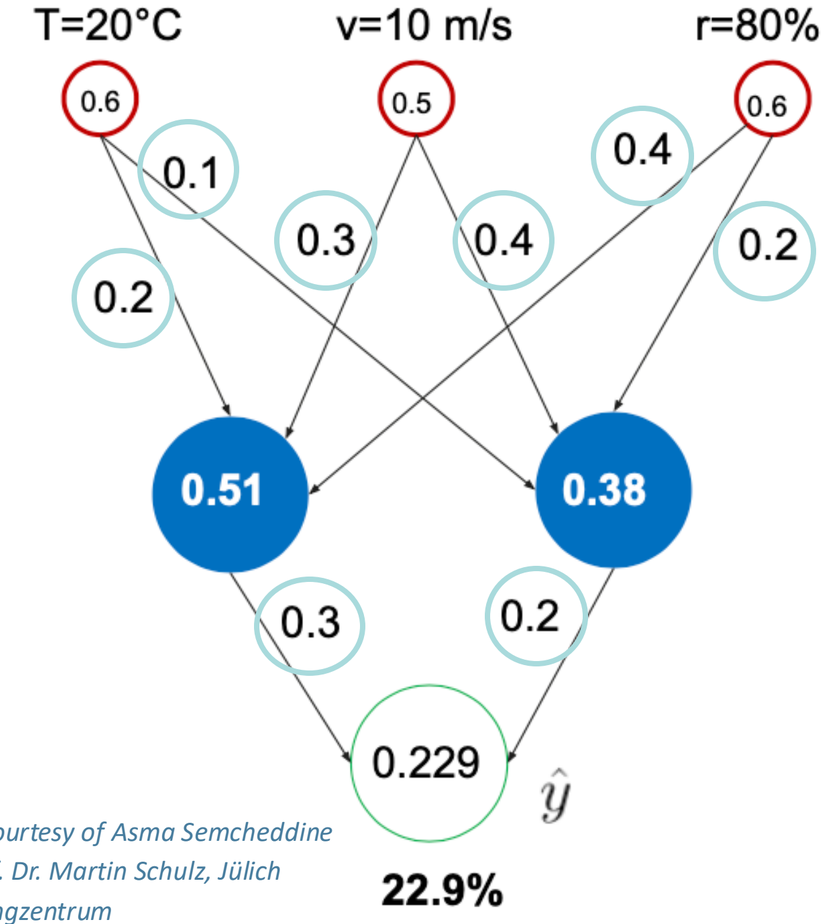


Figure courtesy of Asma Semcheddine and Prof. Dr. Martin Schulz, Jülich Forschungzentrum

normalized input values

randomly initialized weights Wi

Calculating the loss (mean squared error)

$$MSE = \frac{1}{N}\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2$$

we get 0.2119 which is **too big**! This is because the value we obtain, 22.9% is too far from the value expected of 88%.

We do gradient descent, to change the weights to get a lower MSE

$$^{*}w_{ij} = w_{ij} - l_r\frac{\partial MSE}{\partial w_{ij}}$$

new updated weight

old weight

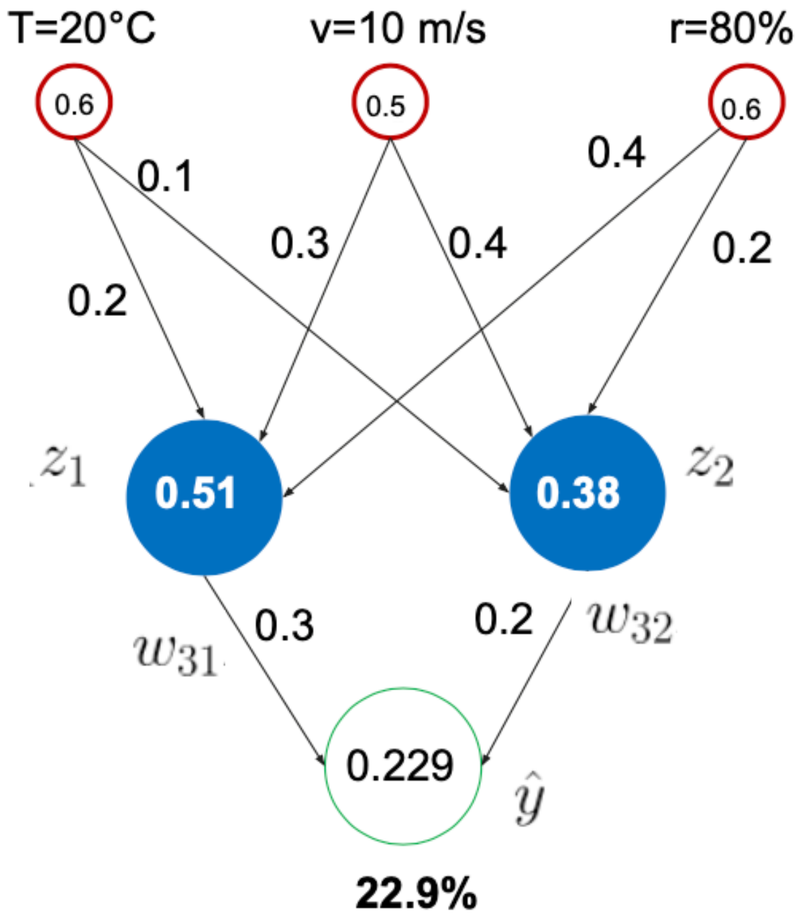
gradient of Loss with respect to the weight

times the learning rate lr

$$l_r = 0.001$$

Example of loss computation and back propagation

We calculate the updated weight for the weight w31



i = 3
j = 1

$$^*w_{ij} = w_{ij} - l_r \frac{\partial MSE}{\partial w_{ij}}$$

$$w_{31} = 0.3$$
$$l_r = 0.001$$

$$\hat{y} = ReLU(z_3) = z_3$$

$$z_3 = w_{31} z_1 + w_{32} z_2$$

Calculating the gradient of the loss with respect to the weight W31

$$MSE = \frac{1}{2}(y_i - \hat{y}_i)^2$$

$$\frac{\partial MSE}{\partial w_{31}} = \frac{\partial MSE}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_{31}}$$

Figure courtesy of Asma Semcheddine and Prof. Dr. Martin Schulz, Jülich Forschungzentrum

Example of loss computation and back propagation

We calculate the updated weight for the weight w31

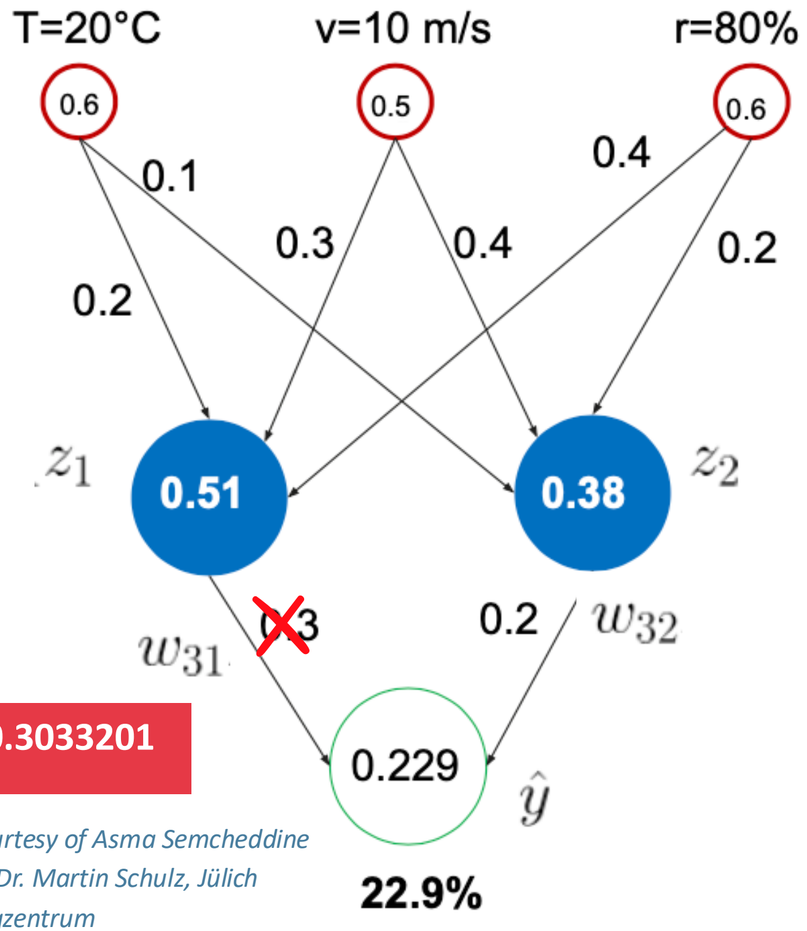


Figure courtesy of Asma Semcheddine and Prof. Dr. Martin Schulz, Jülich Forschungzentrum

i = 3
j = 1

$$^*w_{ij} = w_{ij} - l_r \frac{\partial MSE}{\partial w_{ij}}$$

$$w_{31} = 0.3$$
$$l_r = 0.001$$

$$\hat{y} = ReLU(z_3) = z_3$$

$$z_3 = w_{31} z_1 + w_{32} z_2$$

Calculating the gradient of the loss with respect to the weight W31

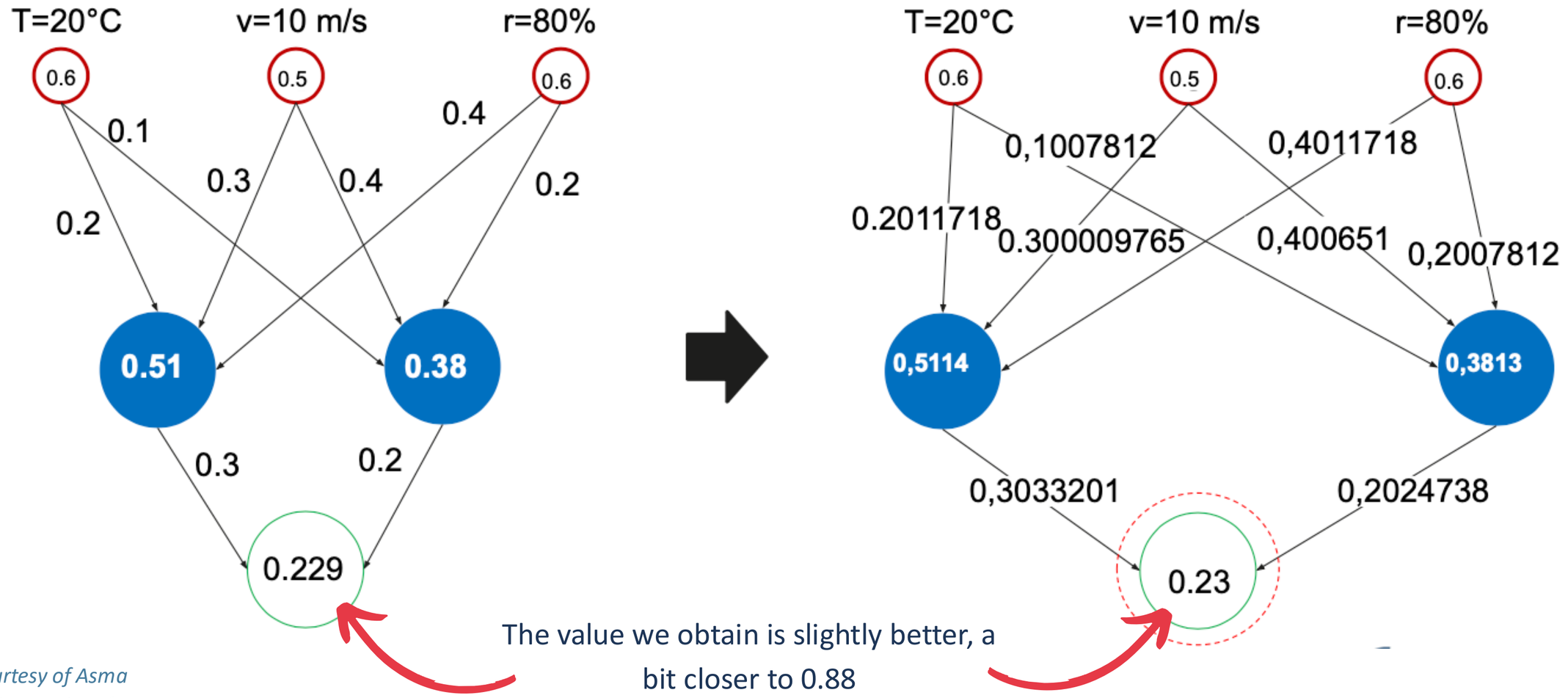
$$MSE = \frac{1}{2}(y_i - \hat{y}_i)^2$$

$$\frac{\partial MSE}{\partial w_{31}} = \frac{\partial MSE}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_{31}}$$

$$\frac{\partial MSE}{\partial w_{31}} = (0,229 - 0,88) \times 1 \times 0,51 = 0,3033201$$

$$^*w_{31} = 0,3 - 0,001 \times 0,3033201 = 0,3033201$$

Updating all weights we can re-calculate the loss with the updated weights



The value we obtain is slightly better, a bit closer to 0.88

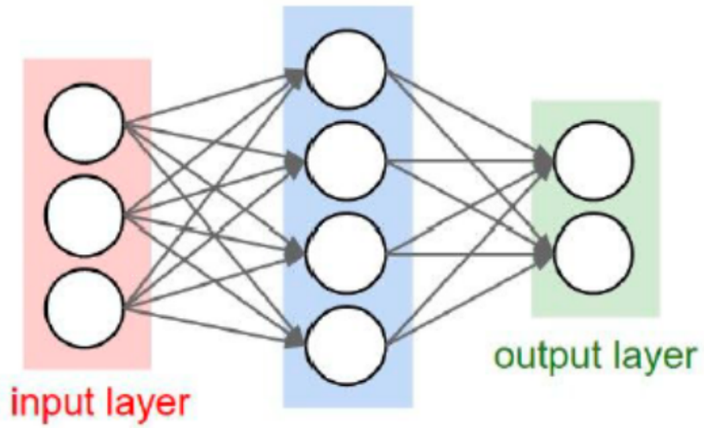
More iterations are needed!!

Figure courtesy of Asma Semcheddine and Prof. Dr. Martin Schulz, Jülich Forschungzentrum

1

Fully connected layers: artificial neural networks

Fully connected layer

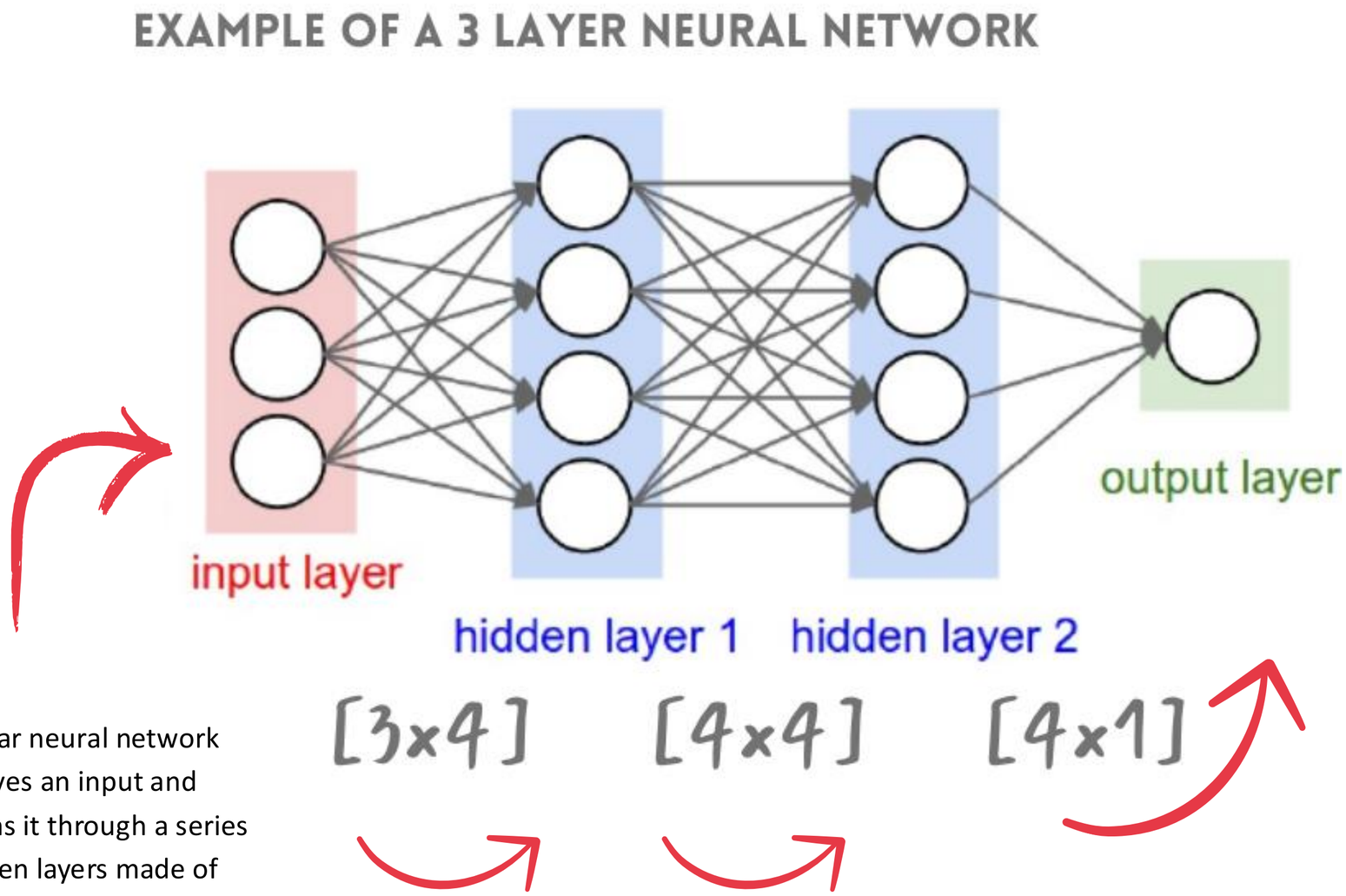


2 layer fully connected neural network. Figure re-elaborated based on the material of the lecture series of the Stanford University's CS231n course.

Each of the fully connected layers works in this way:

no connections among neurons of the same layer.

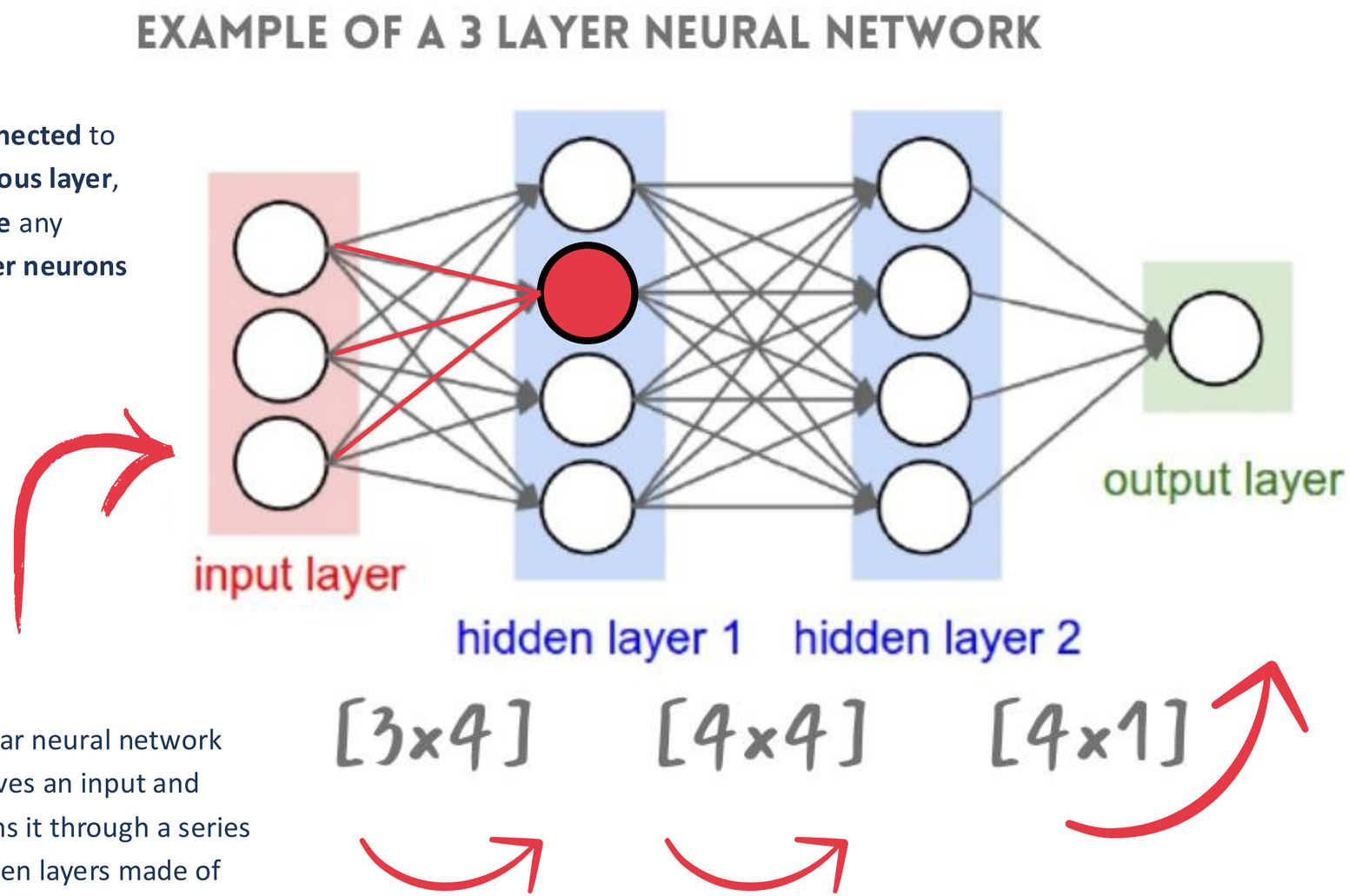
Neural network architectures



A regular neural network receives an input and transforms it through a series of hidden layers made of neurons.

Neural network architectures

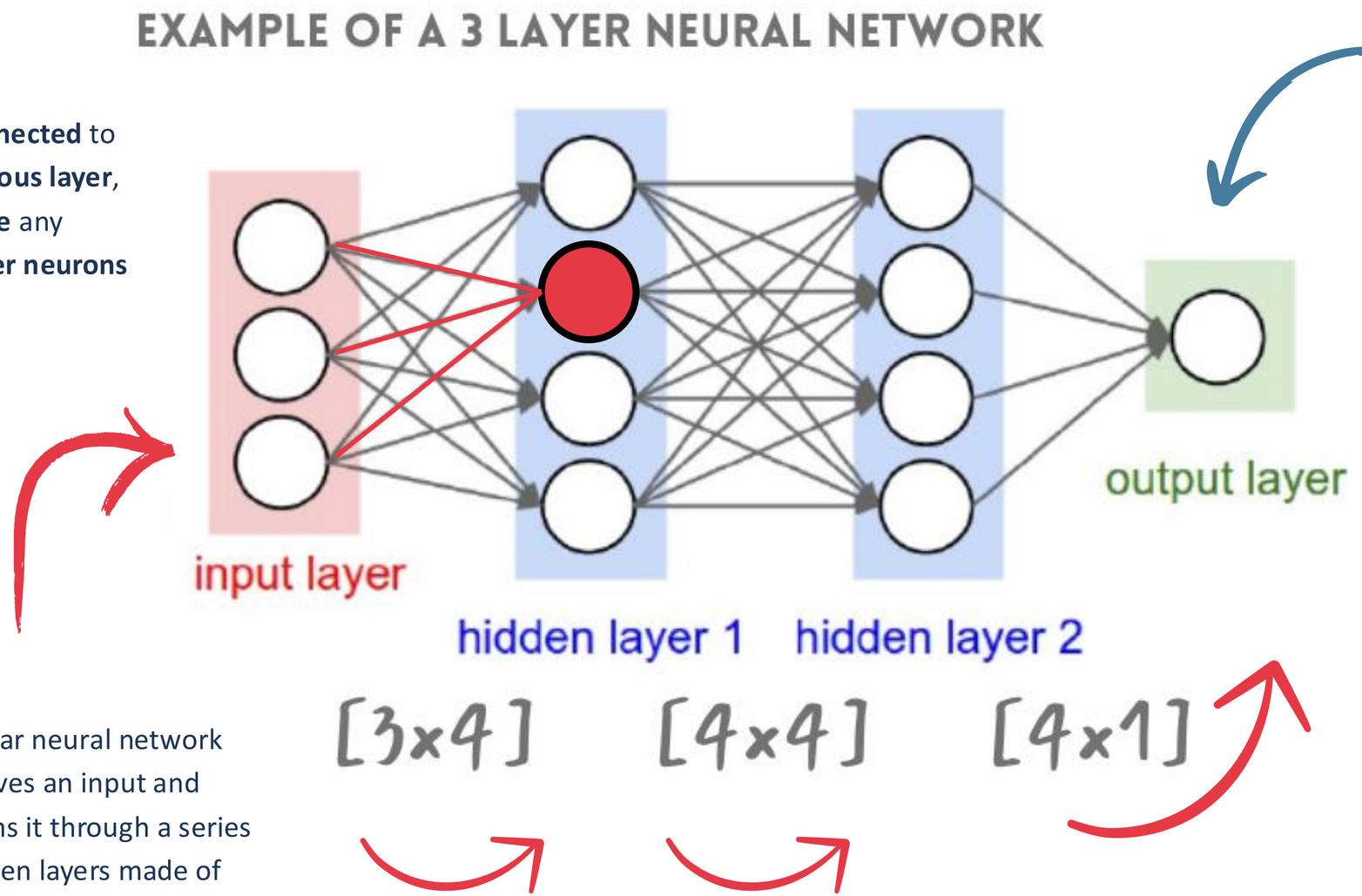
Each neuron is **fully connected** to the neurons of the **previous layer**, but it **does not share** any connection **with the other neurons of his layer**.



A regular neural network receives an input and transforms it through a series of hidden layers made of neurons.

Neural network architectures

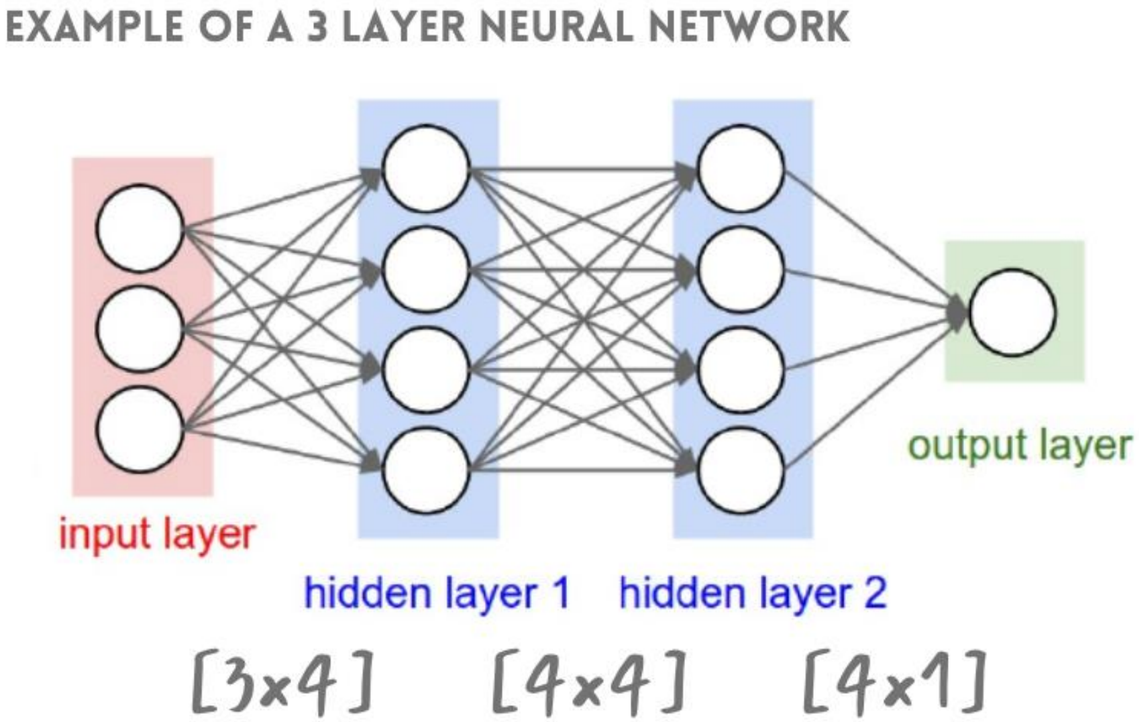
Each neuron is **fully connected** to the neurons of the **previous layer**, but it **does not share** any connection **with the other neurons of his layer**.



The last connected layer is called the output layer; When the goal is **to perform a classification, it contains simply the class scores.**

A regular neural network receives an input and transforms it through a series of hidden layers made of neurons.

Neural network architectures



Number of layers: 3

Number of neurons: 4 + 4 + 1

Number of parameters:

weights: [3x4] + [4x4] + [4x1] = 12 + 16 + 4 = 32

biases: 4 + 4 + 1 = 9

in total: 41 learnable parameters

size of the neural network: number of neurons in the layers of the network (as said before, excluding the input layer) or the number of parameters.

2

Monitoring the learning process
(what to do to try to avoid problems)

Pre-processing of the data

1) Normalization of the input data

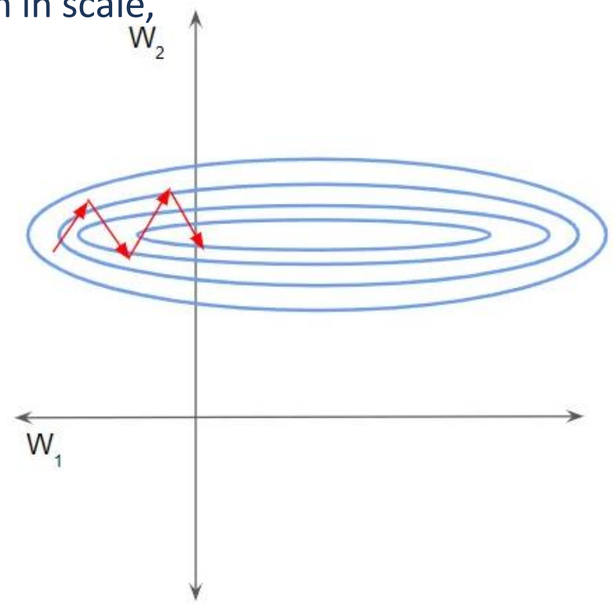
normalization brings all the features of the input on the same scale. But why we need it?

Pre-processing of the data

1) Normalization of the input data

normalization brings all the features of the input on the same scale. But why we need it?

Imagine that we have in particular two features having very **different scales**. Also the weights associated to the features, since the network output is the linear combination of the feature vectors, will differ very much in scale,



large updates on the directions with the largest weights, and much smaller updates on the direction of the smallest feature, gradient oscillates a lot and needs more steps to converge

Pre-processing of the data

1) Normalization of the input data

normalization brings all the features of the input on the same scale. But why we need it?

Imagine that we have in particular two features having very **different scales**. Also the weights associated to the features, since the network output is the linear combination of the feature vectors, will differ very much in scale,

large updates on the directions with the largest weights, and much smaller updates on the direction of the smallest feature, gradient oscillates a lot and needs more steps to converge

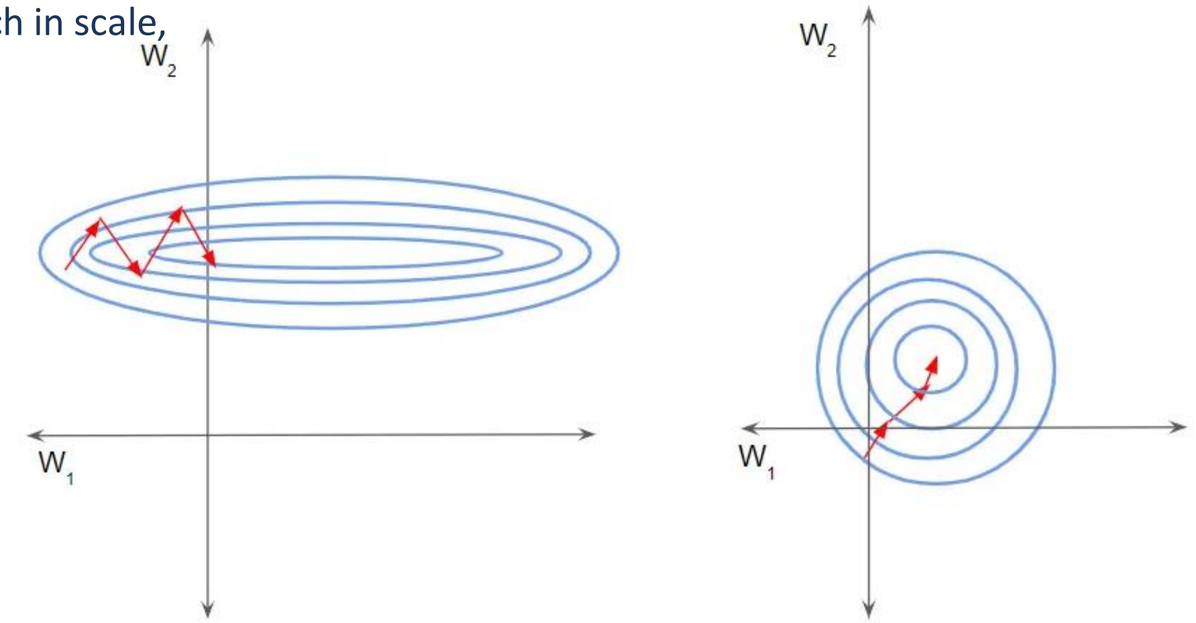


Figure 2.1: a) Example of gradient descent for two un-normalized features and b) gradient descent when the features w1 and w2 are normalized. The graphics are taken from the wonderful article on the batch normalization from Ketan Doshi

features are normalized, the contributions to the gradient during gradient descent will be of the same order, generating a more homogeneous descent

1) Normalization of the input data

1) Normalization of the input data

3 main types of pre-processing

Zero-centering. subtracting the mean from each of the individual features of the data. It concides with the operation of centering the data around the origin. With images, it is common to subtract a single mean value from all pixels of the image (as done in AlexNet), or to do this operation for each of the RGB channels (as done in VGGnet).

Normalization. This is the process that brings data dimensions all on the same scale, approximately. It is obtained by first zero-centering the data, and then by dividing each dimension by its standard deviation. **In case of images, this type of pre-processing is not really needed because pixels scales are relatively equal** (in the range 0-255).

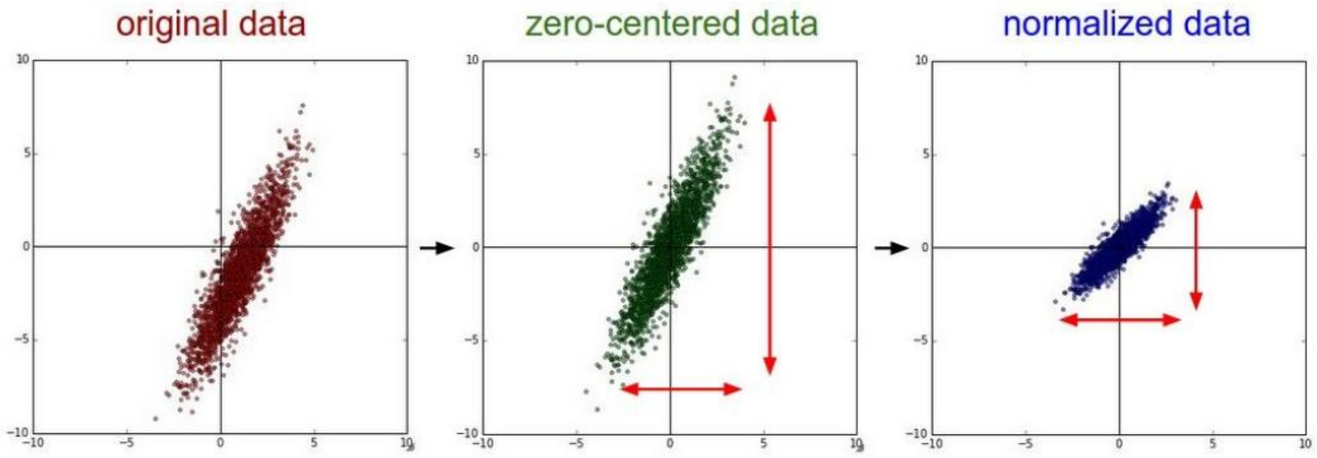


Figure 2.2: The pre-processing methods (figure from the slides of the lecture series of the Stanford course on Computer vision with CNN.

1) Normalization of the input data

3 main types of pre-processing

Zero-centering. subtracting the mean from each of the individual features of the data. It concides with the operation of centering the data around the origin. With images, it is common to subtract a single mean value from all pixels of the image (as done in AlexNet), or to do this operation for each of the RGB channels (as done in VGGnet).

Normalization. This is the process that brings data dimensions all on the same scale, approximately. It is obtained by first zero-centering the data, and then by dividing each dimension by its standard deviation. **In case of images, this type of pre-processing is not really needed because pixels scales are relatively equal (in the range 0-255).**

PCA and whitening. Data are centered as described above, but in this case we then calculate the covariance matrix that will give information about the correlation structure of the data. Then we apply the singular value decomposition to obtain the eigenvectors and array of singular values. The transformation of data covariance into the identity matrix corresponds to squeeze the data in an isotropic bubble (**PCA dimensionality reduction)** -- can greatly amplify noise

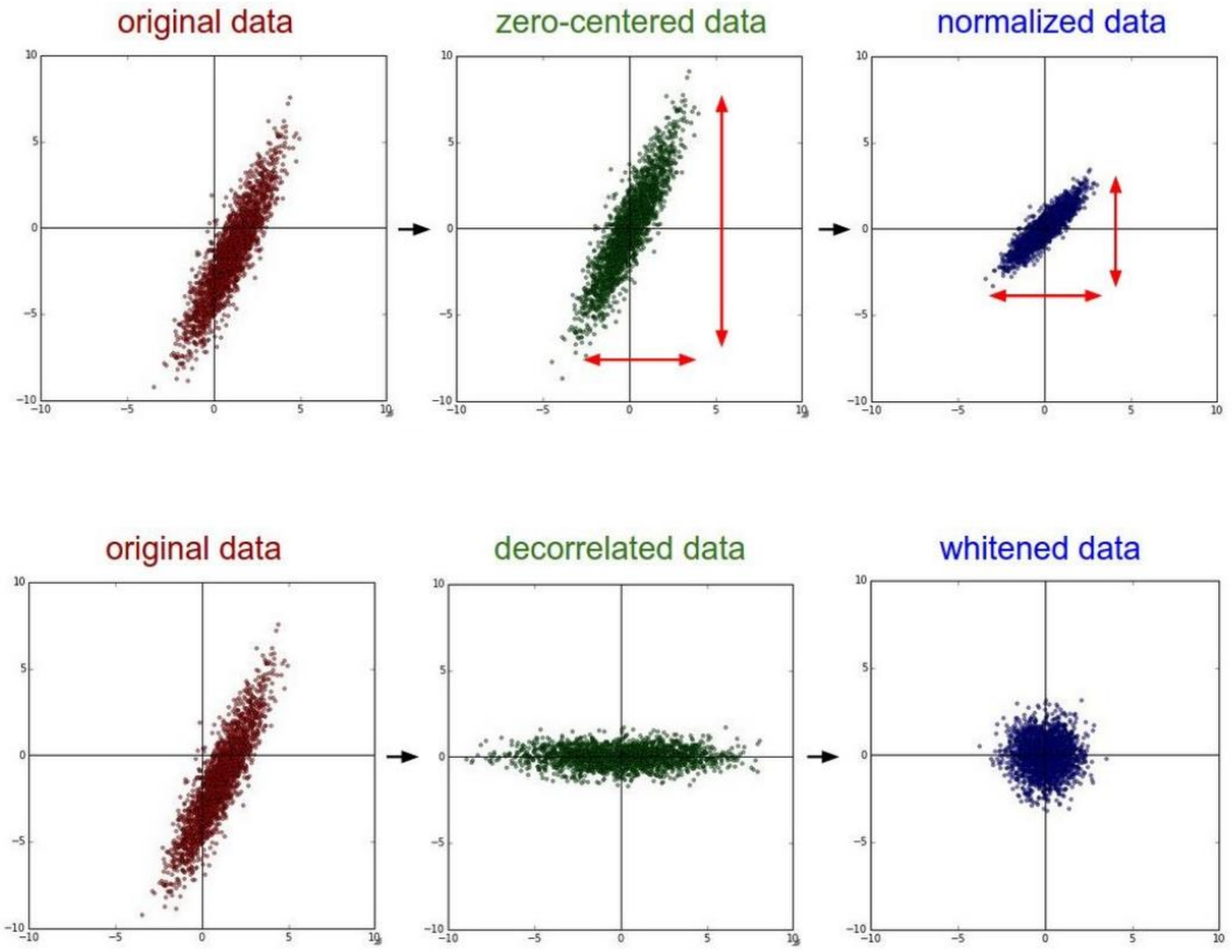


Figure 2.2: The pre-processing methods (figure from the slides of the lecture series of the Stanford course on Computer vision with CNN.

2) Weight initialization

Option 1) set all weights initially to 0

the network will not learn because there is no simmetry breaking: all neurons will do the same thing, and they will all give the same gradient, so they will be updated in the same way.

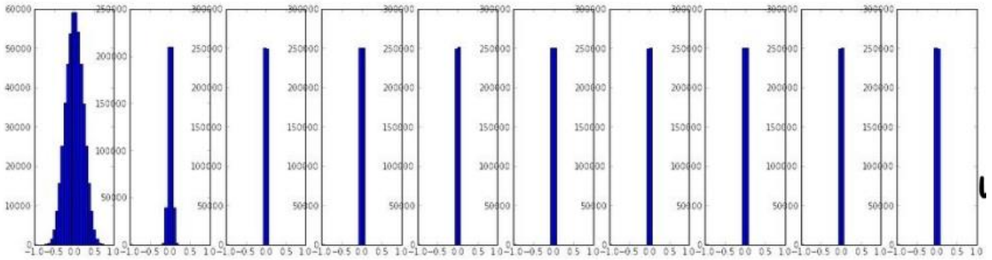
2) Weight initialization

Option 1) set all weights initially to 0

the network will not learn because there is no simmetry breaking: all neurons will do the same thing, and they will all give the same gradient, so they will be updated in the same way.

Option 2) set of small random numbers.

the simmetry would be broken, but the network might not work for deep architectures



Distribution of the activation functions when initialization is performed using small random numbers. The mean stays constant, but the variance gets soon attenuated to zero. From Stanford lectures

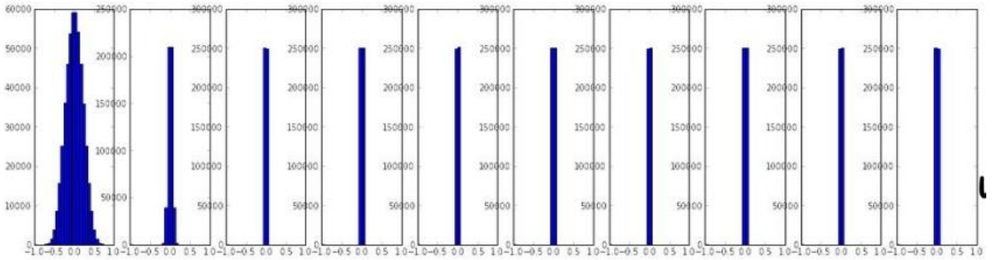
2) Weight initialization

Option 1) set all weights initially to 0

the network will not learn because there is no simmetry breaking: all neurons will do the same thing, and they will all give the same gradient, so they will be updated in the same way.

Option 2) set of small random numbers.

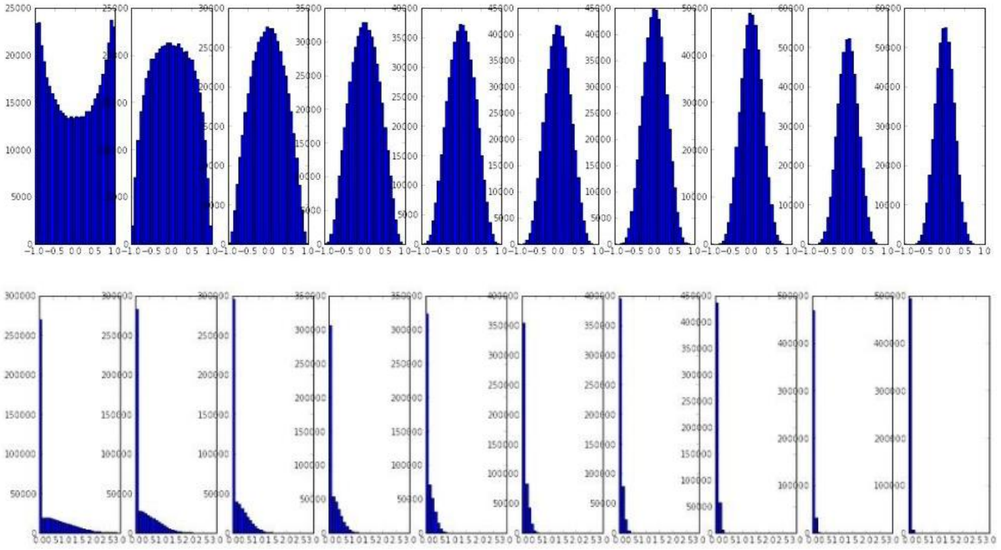
the simmetry would be broken, but the network might not work for deep architectures



Distribution of the activation functions when initialization is performed using small random numbers. The mean stays constant, but the variance gets soon attenuated to zero. From Stanford lectures

Option 3) “Xavier initialization” [Glorot et al., 2010]

Reasonable initialization based on mathematical derivation, assumes linear activations but with RELU it breaks)



Distribution of the activation functions for Xavier initialization with linear activation (top) and with ReLU non linear activations (bottom)

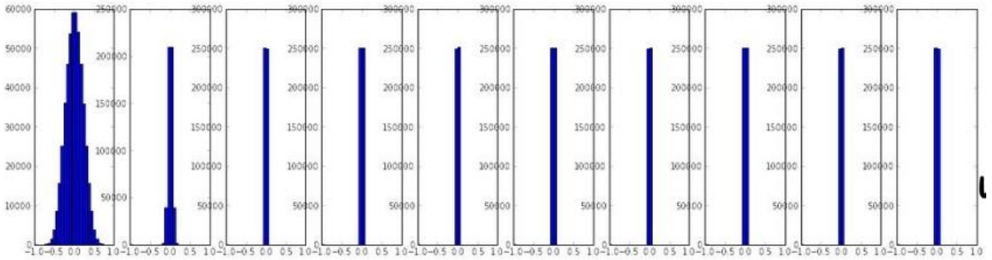
2) Weight initialization

Option 1) set all weights initially to 0

the network will not learn because there is no simmetry breaking: all neurons will do the same thing, and they will all give the same gradient, so they will be updated in the same way.

Option 2) set of small random numbers.

the simmetry would be broken, but the network might not work for deep architectures



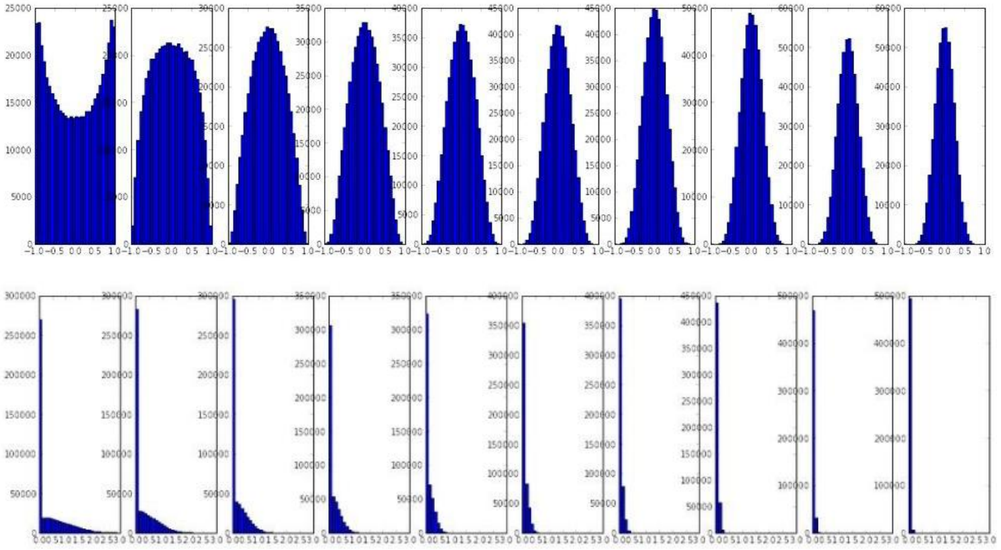
Distribution of the activation functions when initialization is performed using small random numbers. The mean stays constant, but the variance gets soon attenuated to zero. From Stanford lectures

Option 3) “Xavier initialization” [Glorot et al., 2010]

Reasonable initialization based on mathematical derivation, assumes linear activations but with RELU it breaks)

Option 4) He et al., 2015

accounted for non linearities by modifying a scaling factor and making it work with Relu activations

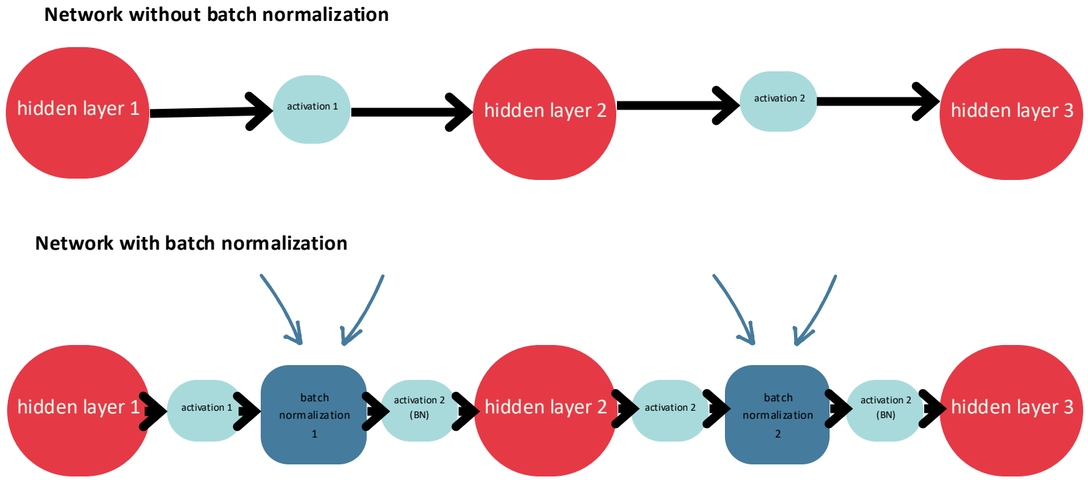


Distribution of the activation functions for Xavier initialization with linear activation (top) and with ReLU non linear activations (bottom)

3) Batch normalization

What is batch normalization?

Batch normalization comes in the network as an additional layer that is usually added to other blocks of the architecture, like the convolutional or the fully connected layer.



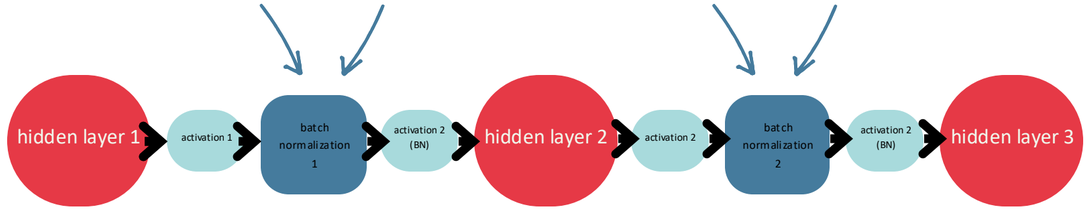
3) Batch normalization

What is batch normalization?

Batch normalization comes in the network as an additional layer that is usually added to other blocks of the architecture, like the convolutional or the fully connected layer.

Network without batch normalization

Network with batch normalization

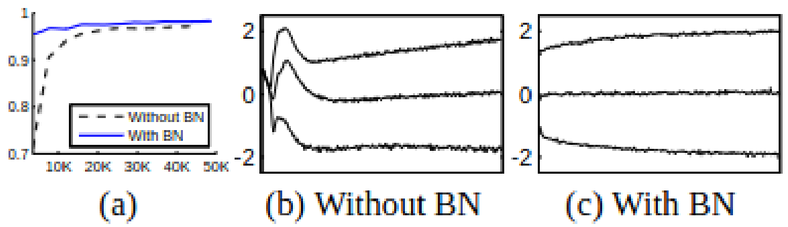


Why does batch normalization help in training neural networks?

1) It reduces the internal covariate shift.

Covariate shift occurs when the model is trained with data having a very different distribution with respect to the data which are used for inference, slowing down convergence.
Batch Normalization helps the network train faster and achieve higher accuracy.

From the batch optimization paper (Ioffe and Szegedy, 2015): (a) (b, c) The evolution of input distributions to a typical sigmoid, over the course of training, shown as {15, 50, 85}the percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift



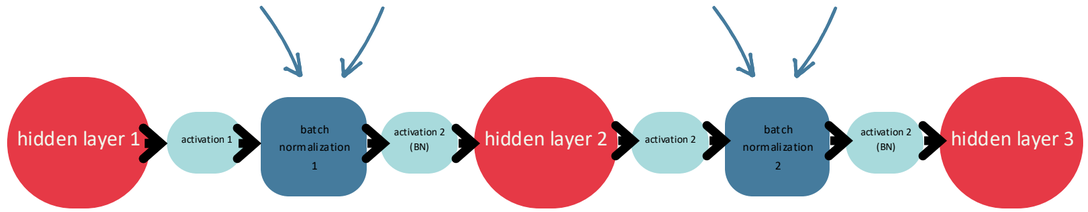
3) Batch normalization

What is batch normalization?

Batch normalization comes in the network as an additional layer that is usually added to other blocks of the architecture, like the convolutional or the fully connected layer.

Network without batch normalization

Network with batch normalization

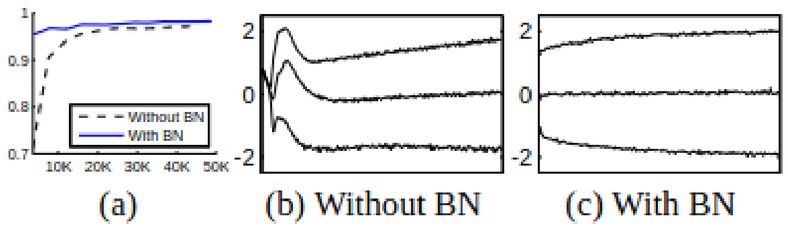


Why does batch normalization help in training neural networks?

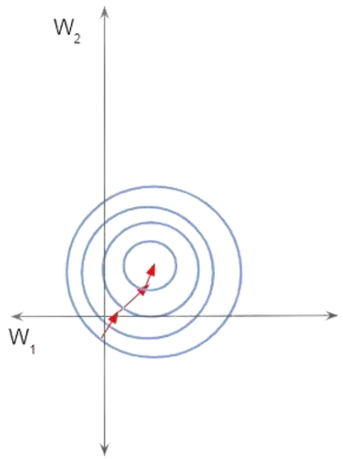
1) It reduces the internal covariate shift.

Covariate shift occurs when the model is trained with data having a very different distribution with respect to the data which are used for inference, slowing down convergence.
Batch Normalization helps the network train faster and achieve higher accuracy.

From the batch optimization paper (Ioffe and Szegedy, 2015): (a) (b, c) The evolution of input distributions to a typical sigmoid, over the course of training, shown as {15, 50, 85}the percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift



2) It smooths the loss function and the gradient.



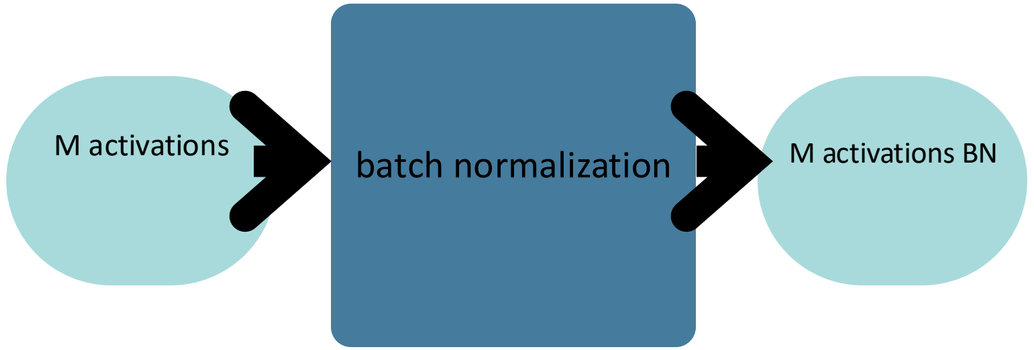
Smoothens the loss landscape by changing the distribution of the weights of the networks. In this way, steps of gradient descent can be larger in a given direction and learning rate can be higher. If you want to know more about this aspect, check the paper from Li et al., 2018.

How does batch normalization practically work (does the normalization task)?

Parameters of the batch normalization layer:

- two **learnable** parameters (**beta** and **gamma**)
- two **non-learnable parameters** (mean moving average and variance moving average)

During training feedforward phase, we provide as input a mini-batch of data, i.e. M samples of the N features



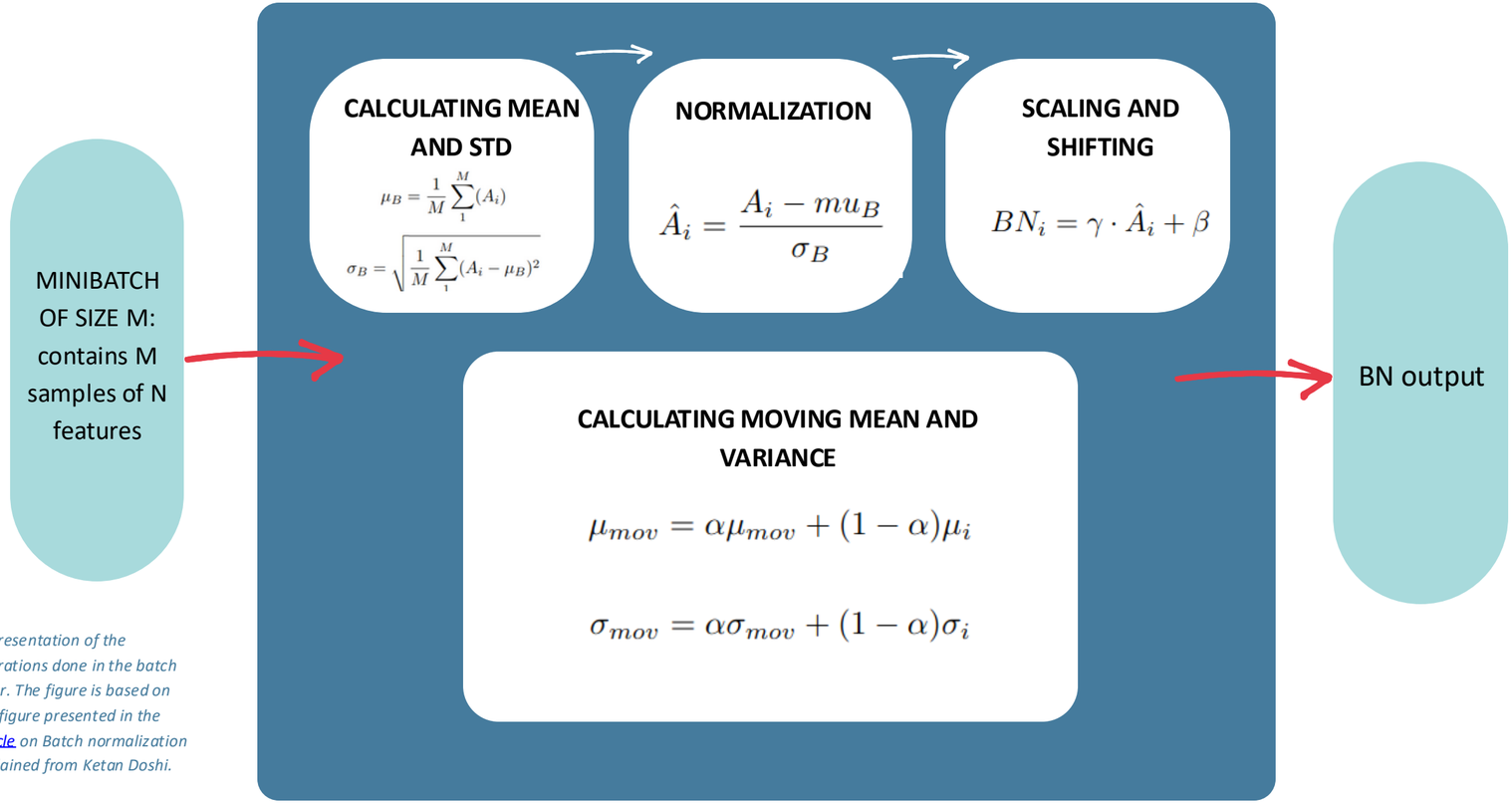
How does batch normalization practically work (does the normalization task)?

Parameters of the batch normalization layer:

- two **learnable** parameters (**beta** and **gamma**)
- two **non-learnable parameters** (mean moving average and variance moving average)

During training feedforward phase, we provide as input a mini-batch of data, i.e. M samples of the N features

Operations the batch does in training



Representation of the operations done in the batch layer. The figure is based on the figure presented in the article on Batch normalization explained from Ketan Doshi.

the parameters **beta and gamma are learned** in the training process, like all weights and the **batch is optimizing them during training** to fit the values to those giving the best predictions.

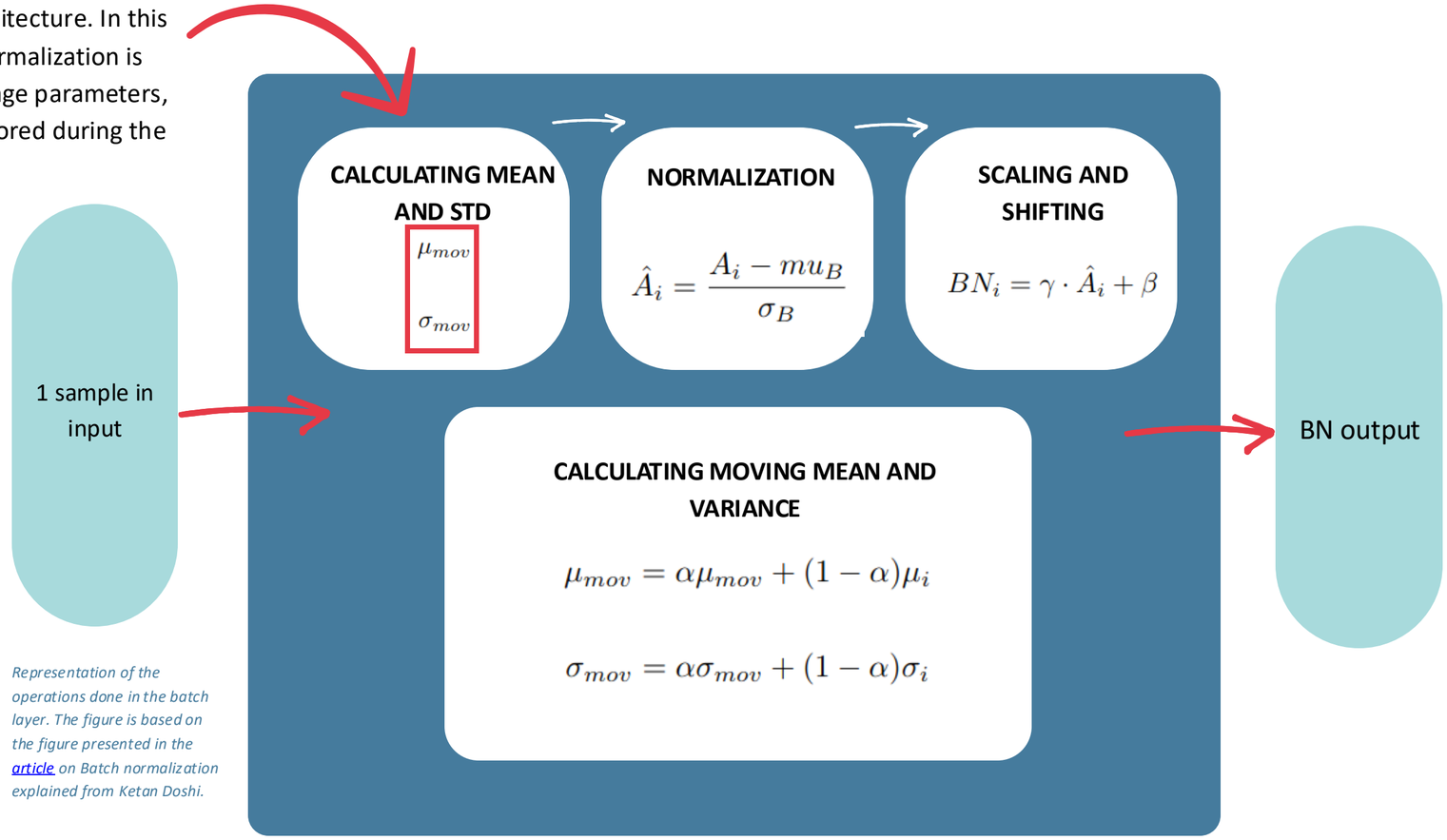
scaling and shifting is what allows the batch to shift the output to a different mean and standard deviation

In addition to these, used to generate the output, the layer also stores a running count of the exponential moving average of the mean and variance, obtaining an EMA at the end of the training, that we will then use in the inference phase

How does batch normalization practically work (does the normalization task)?

During inference, after the training, the activations flow in the same architecture. In this case, in the batch layer, the normalization is done using the two moving average parameters, that have been calculated and stored during the training

Operations the batch does in inference



Representation of the operations done in the batch layer. The figure is based on the figure presented in the article on Batch normalization explained from Ketan Doshi.

While ideally we could have calculated and saved the mean and variance for all the data in training, this operation would have been very expensive. Moving average is a good proxy and it is more efficient because the calculation is incremental.

Methods to avoid overfitting

Data augmentation

In classification tasks, this can be easily done by manipulating the input images using some transformations like rotation, rescaling or shifting

L1/L2 regularization

In **L1 regularization**, we add to the cost function the term $\lambda|w|$, which allows all the weights to decay to zero. It penalizes the sum of the absolute values of the weights, and it is robust to outliers.

L2 regularization instead we add the term

$$\frac{1}{2}\lambda w^2$$

which penalizes the sum of the square values of the weights (the peaky weight vectors) and preferring the diffuse ones, but it is less solid to outliers.

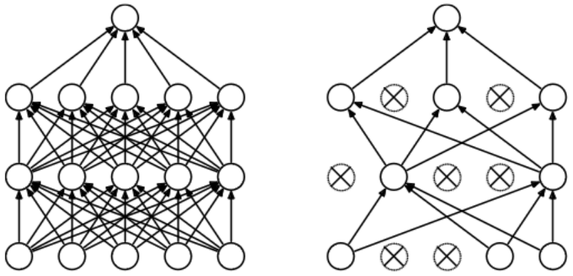
Max norm constraints

avoid overfitting by limiting the values of the weights in the model so that they modulus is less than a fixed threshold. Typically, after parameter update, the vector of weights is forced to satisfy

$$||\vec{w}||_{L_2} < C$$

Data dropout

reduce the independent learning units of the network, diminishing the complexity of the model, by ignoring some sets of neuron units of the model. Implemented in the training phase, by keeping a neuron active with a given probability p, and set to zero otherwise, becoming p another hyperparameter of the model. During testing there is no dropout applied

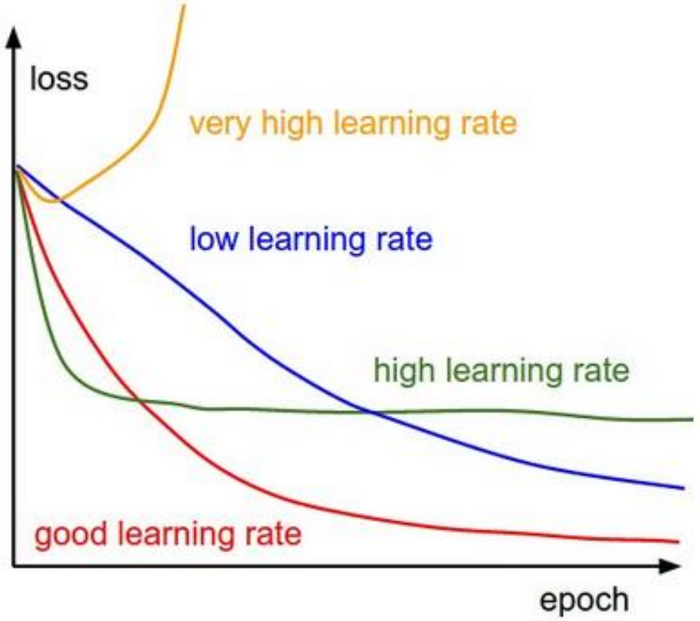


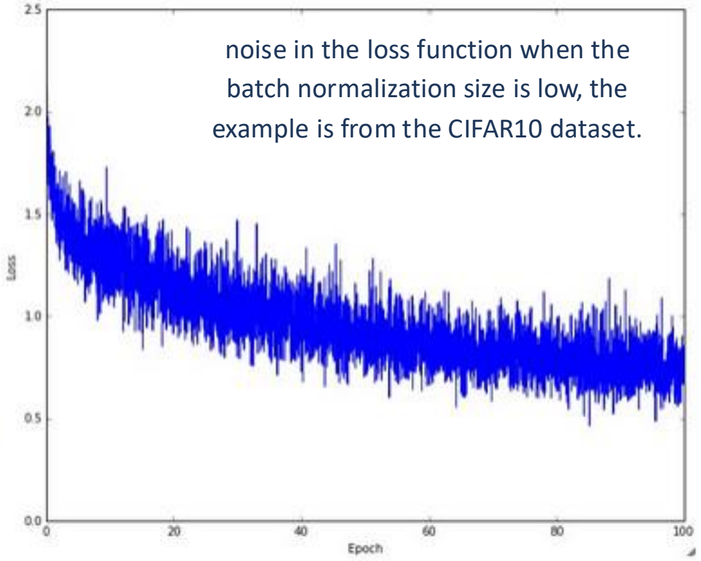
(a) Standard Neural Net (b) After applying dropout.

Monitoring learning process: what to plot

To monitor the learning process of the network, one should **look at how some parameters evolves as the epochs** of the iterations **progress**. In particular, It is useful to plot as a function of epochs:

Loss function drawings indicating the shapes that reveal high, low and good learning rate.

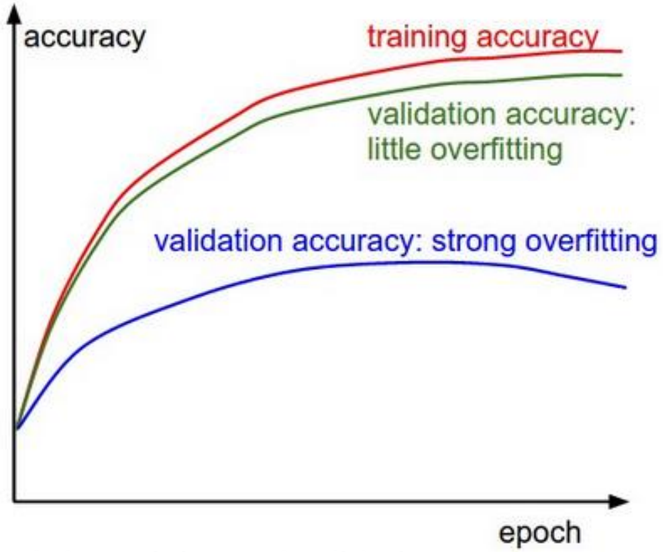




noise in the loss function when the batch normalization size is low, the example is from the CIFAR10 dataset.

1) the loss function: from the shape of the loss function as a function of the epochs we can get some information on the correctness of the learning rate value we assigned.

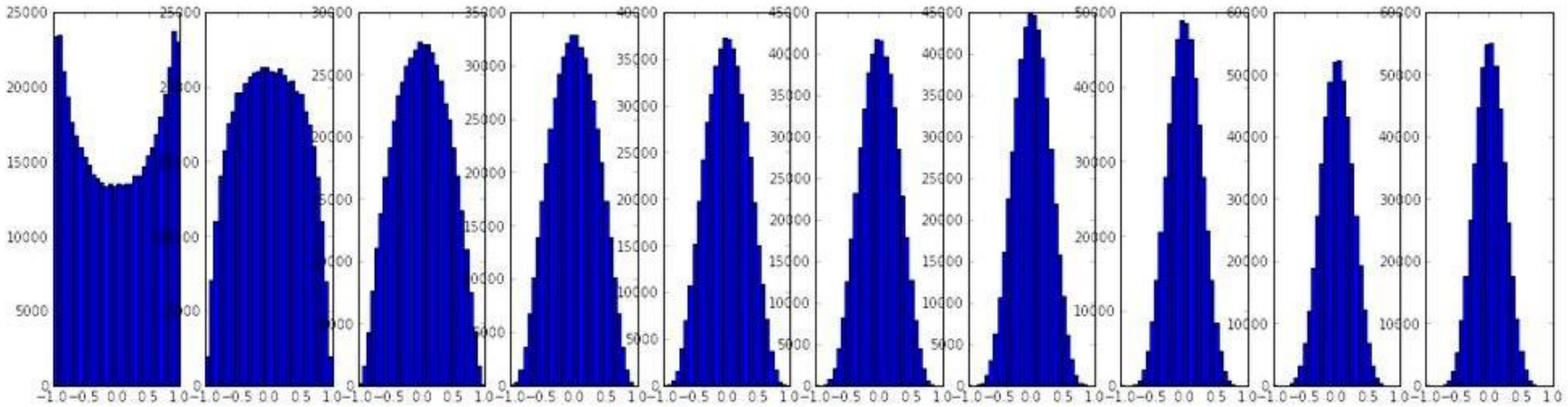
training and validation accuracy for the case of strong and little overfitting.



2) the training and validation accuracy: these two quantities can give indications on the amount of overfitting of the model.

3) ratio of the weights: it is another quantity giving indications on the learning rate. It is calculated by taking the ratio of the update values to the magnitude values of the weights and the reference value indicating a good learning rate is 1.3.

4) activation and gradient distributions per each layer: a useful tool is the visualization of the distributions of the activations or the gradients at each layer.

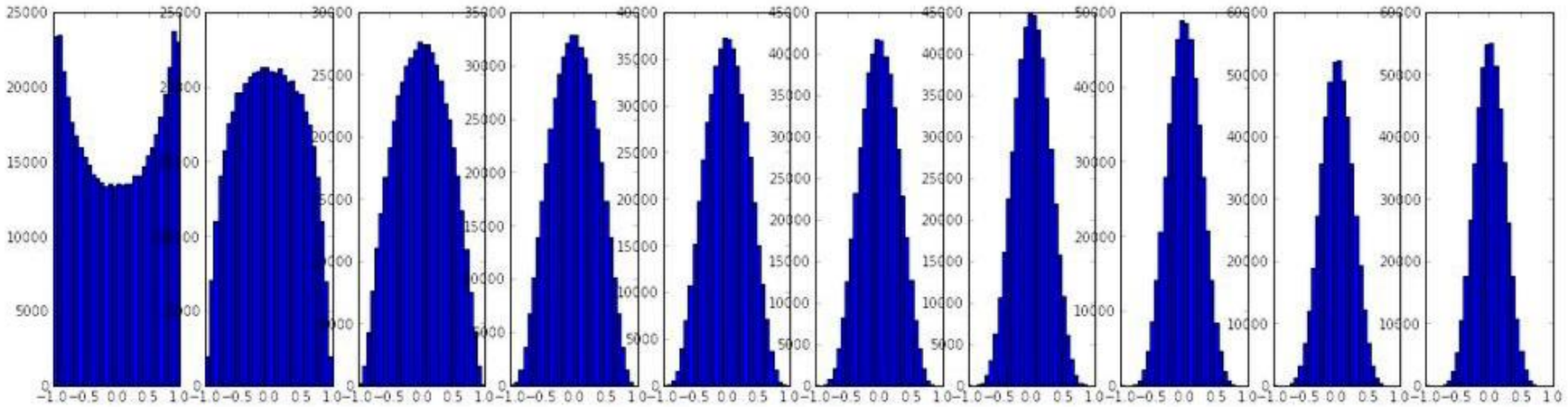


Distribution of the activation functions when initialization is performed using small random numbers. The mean stays constant, but the variance gets soon attenuated to zero. From Stanford lectures

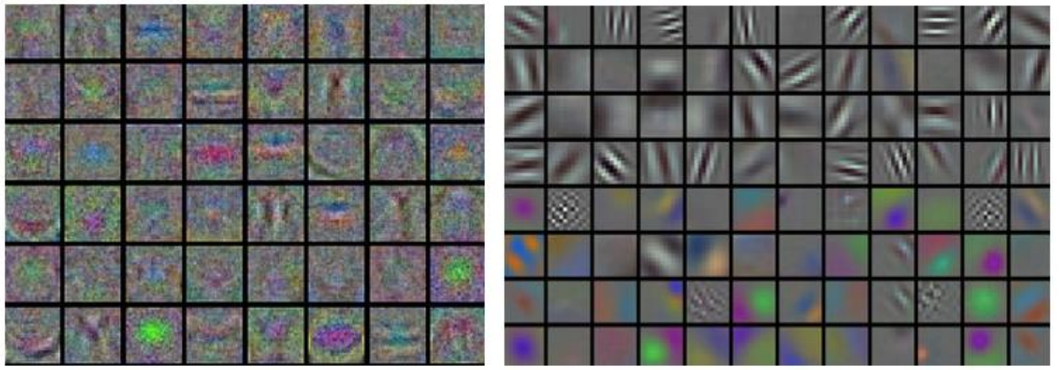
3) ratio of the weights: it is another quantity giving indications on the learning rate. It is calculated by taking the ratio of the update values to the magnitude values of the weights and the reference value indicating a good learning rate is 1.3.

4) activation and gradient distributions per each layer: a useful tool is the visualization of the distributions of the activations or the gradients at each layer.

Distribution of the activation functions when initialization is performed using small random numbers. The mean stays constant, but the variance gets soon attenuated to zero. From Stanford lectures



5) visualization of the first layer: when you work with images, it can be useful to visualize the features (weights) of the first layer. Noisy features could reveal unconvergence in the network, wrong learning rate, or low regularization penalty



Left: noisy weights for the first layer of the neural network, right: smooth features that indicate the training is going fine, from the CS231n Stanford course in computer vision.

Methods to update parameters

Vanilla update

The update is done along the direction of the negative gradient.

$$x = x - \text{learning-rate} * dx$$

Newton's method

it is a second order method that iterates an update dependent on the Hessian matrix, i.e.. a matrix of the second order partial derivatives of the function. The gradient vector is the same seen in the gradient descent. With the local curvature given by the Hessian, updates are more efficient.

$$x \leftarrow x - [Hf(x)]^{-1} \nabla f(x)$$

Momentum update

based on interpreting the loss as a potential energy function, it sets the initial parameters like to put a particle in an initial position with zero velocity. Then, if we imagine to apply a force to the particle, this force is the exactly the negative gradient of the loss function. In this case the gradient impacts the velocity, and then the velocity impacting on the position. There's a new hyperparameter, that can be associated in the physical meaning to the role of friction that dampens the velocity and reduces the kinetic energy of the system.

$$v = \mu v - \text{learning-rate} * dx$$

$$x = x + v$$

Nesterov momentum

similar to the one of the momentum update, but here we threat the future approximate position as a "look ahead".

$$x_{ahead} = x + \mu * v$$

$$v = \mu v - \text{learning-rate} * dx_{ahead}$$

$$x = x + v$$

Methods to update parameters

Vanilla update

The update is done along the direction of the negative gradient.

$$x = x - \text{learning-rate} * dx$$

Newton's method

it is a second order method that iterates an update dependent on the Hessian matrix, i.e.. a matrix of the second order partial derivatives of the function. The gradient vector is the same seen in the gradient descent. With the local curvature given by the Hessian, updates are more efficient.

$$x \leftarrow x - [Hf(x)]^{-1} \nabla f(x)$$

Momentum update

based on interpreting the loss as a potential energy function, it sets the initial parameters like to put a particle in an initial position with zero velocity. Then, if we imagine to apply a force to the particle, this force is the exactly the negative gradient of the loss function. In this case the gradient impacts the velocity, and then the velocity impacting on the position. There's a new hyperparameter, that can be associated in the physical meaning to the role of friction that dampens the velocity and reduces the kinetic energy of the system.

$$v = \mu v - \text{learning-rate} * dx$$
$$x = x + v$$

Nesterov momentum

similar to the one of the momentum update, but here we threat the future approximate position as a "look ahead".

$$x_{ahead} = x + \mu * v$$
$$v = \mu v - \text{learning-rate} * dx_{ahead}$$
$$x = x + v$$

good practice in machine learning is to **anneal the learning rate over time**. You can imagine the learning rate as a sort of level of kinetic energy available in the system. When it is too high, particles bounces randomly around and cannot reach the minima. However, we need to be careful on how we make the learning rate decay, because too slow decay can make the system converge too quickly, without finding the best position. Types of implementation for the learning decay: the step decay, the exponential decay, 1/t decay.

and many more: Adagrad, RMSProp, Adam, L-BFGS

that’s it for today!