Universität Konstanz

03 | Multilayer Perceptron

Giordano De Marzo

https://giordano-demarzo.github.io/

Deep Learning for the Social Sciences

Recap

- Basic Concepts and Notation
- Perceptron
- Limits of the Perceptron
- Shallow Neural Networks

Universität Konstanz

Outline

1. Shallow Neural Networks
2. Multilayer Perceptron Architecture
3. Training Deep Neural Networks
4. Backpropagation Algorithm

Universität
Konstanz

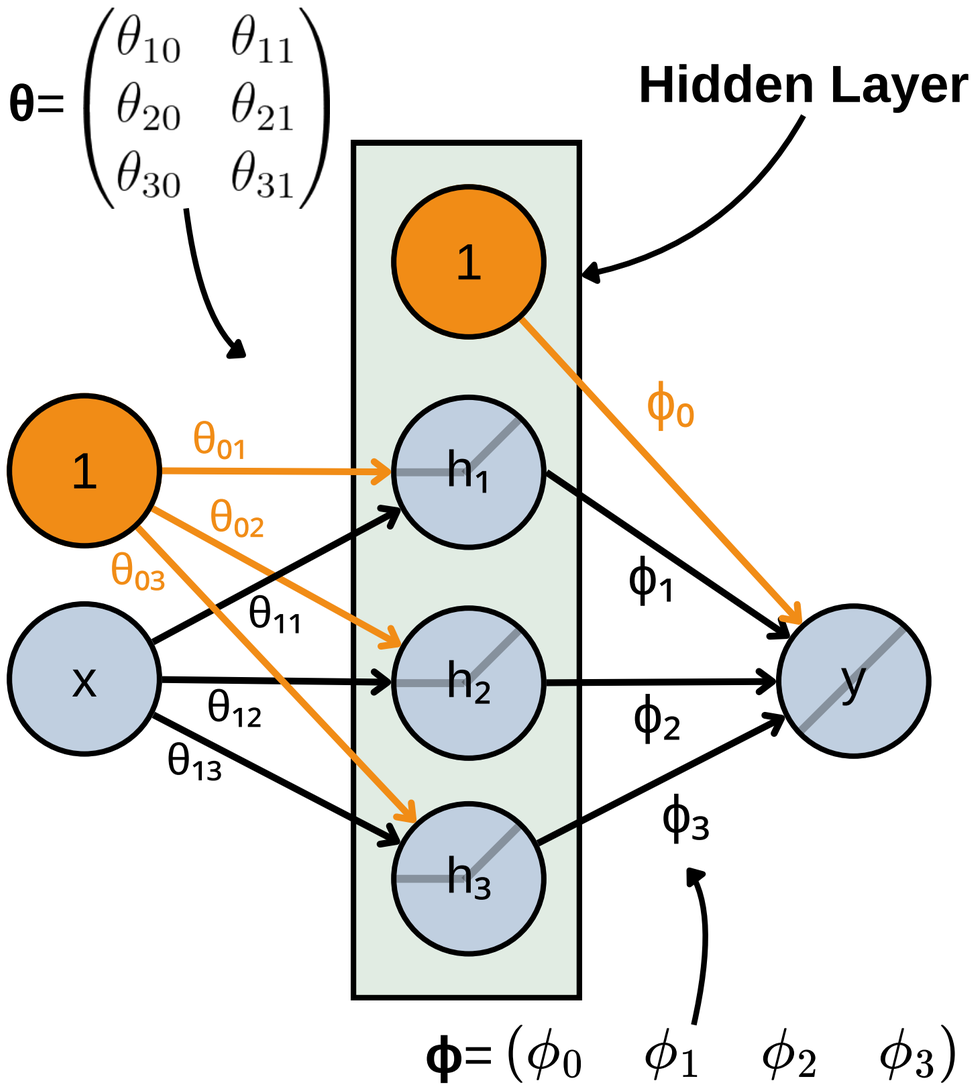
Shallow Neural Networks

Combining Perceptrons

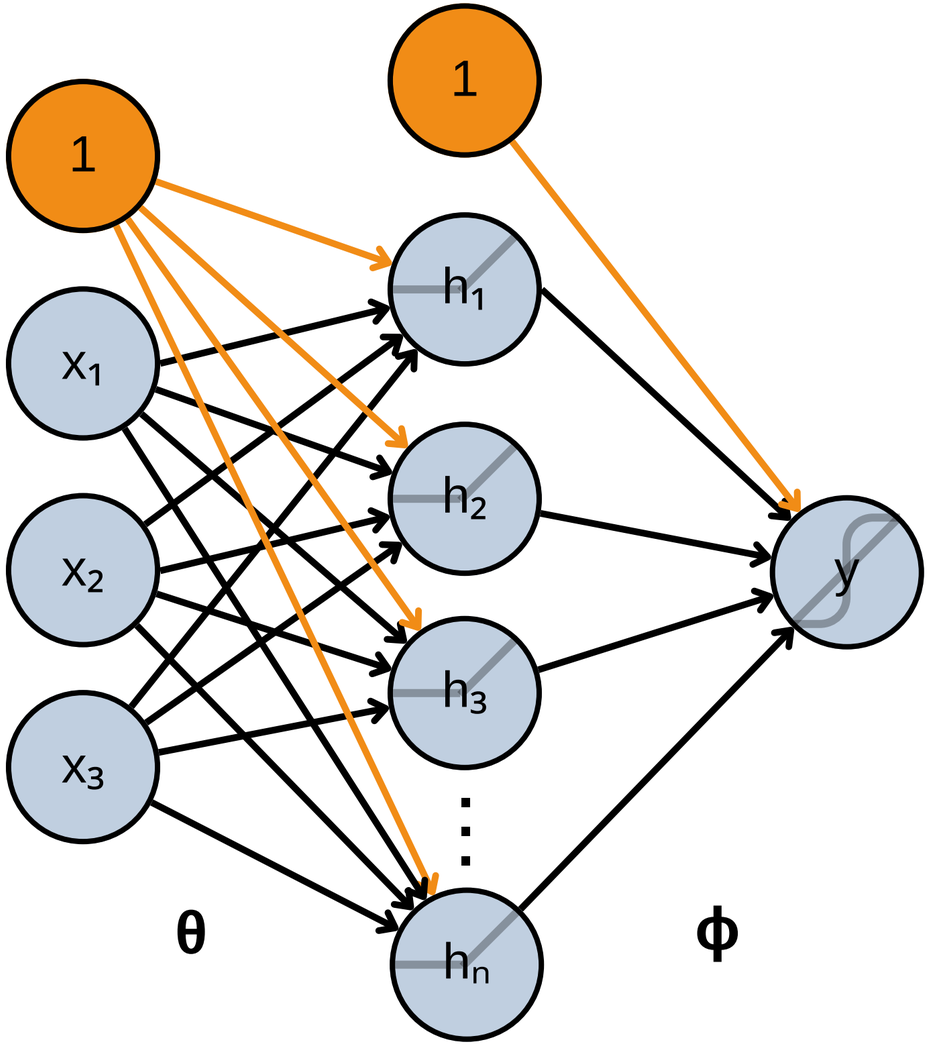
We consider again the simple regression problem with a single input x and output y

- we can apply more than one single perceptron to the input (and dummy)
- each of these perceptrons will produce a different output h_i
- we can then use these outputs as input for another perceptron that produce the output y

In this way we are adding an **hidden layer** to the neural network



Shallow Neural Networks



More generally we can have neural networks with

- as many inputs as we want
- as many hidden neurons as we want

The parameters of this neural network will be contained in two weight matrices

- **θ** connecting the input to the hidden layer
- **ϕ** connecting the hidden layer to the output

This type of neural network with a single hidden layer is called **Shallow Neural Network**

Mathematical Representation

We denote by **x** input vector with the dummy

x=(1, x)

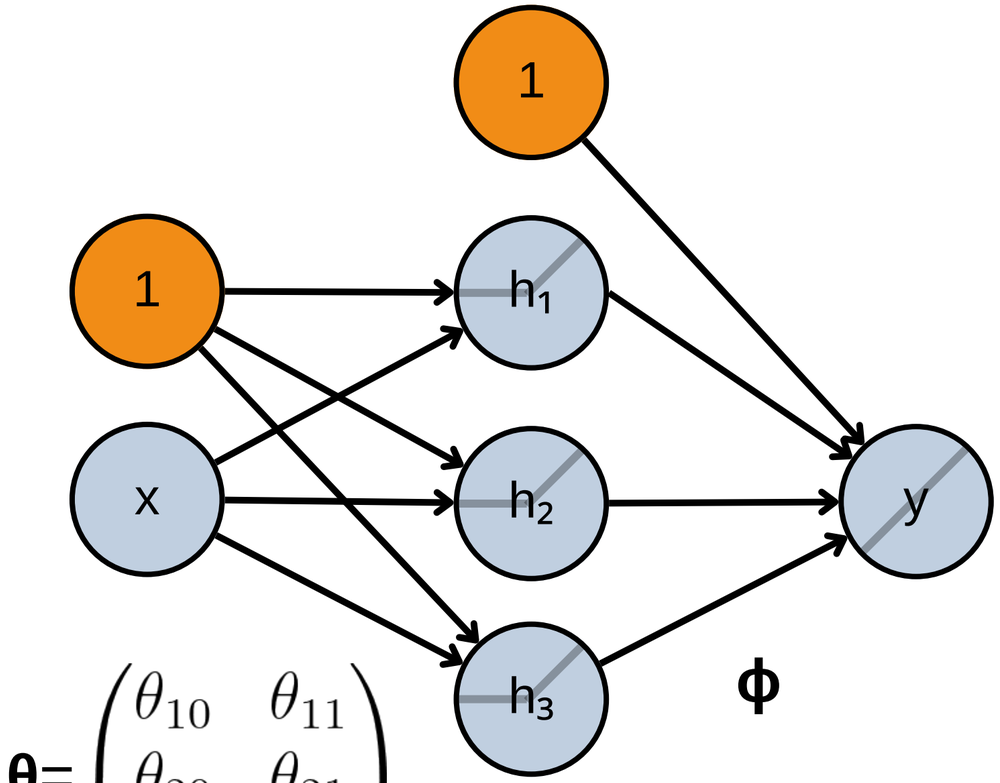
The hidden neurons **h'**=(h_1, h_2, h_3) satisfy

h'=a[**θx**]

Where a is the ReLU activation.

Hidden units

$$\begin{cases} h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x] \end{cases}$$



$$\boldsymbol{\theta} = \begin{pmatrix} \theta_{10} & \theta_{11} \\ \theta_{20} & \theta_{21} \\ \theta_{30} & \theta_{31} \end{pmatrix}$$

Mathematical Representation

We denote by **x** input vector with the dummy

x=(1, x)

The hidden neurons **h'**=(h_1, h_2, h_3) satisfy

h'=a[**θx**]

Where a is the ReLU activation.

We can then define the hidden unit vector as

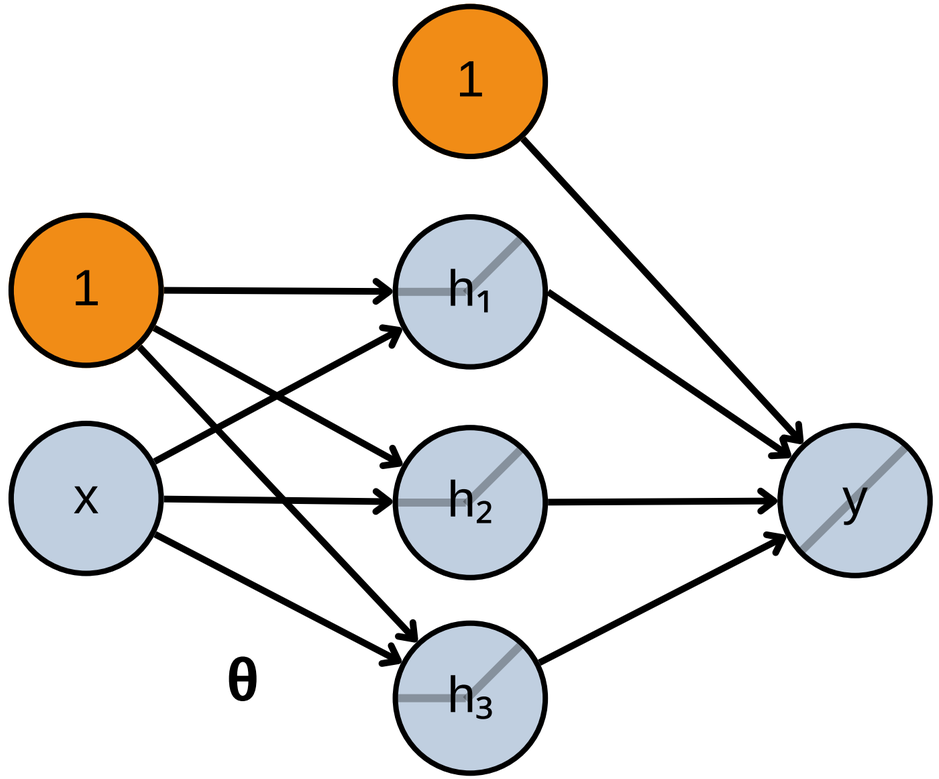
h=(1, h_1, h_2, h_3)=(1, **h'**)

and get the output y as

y=a[**ϕh**]

Where this time a is a linear activation

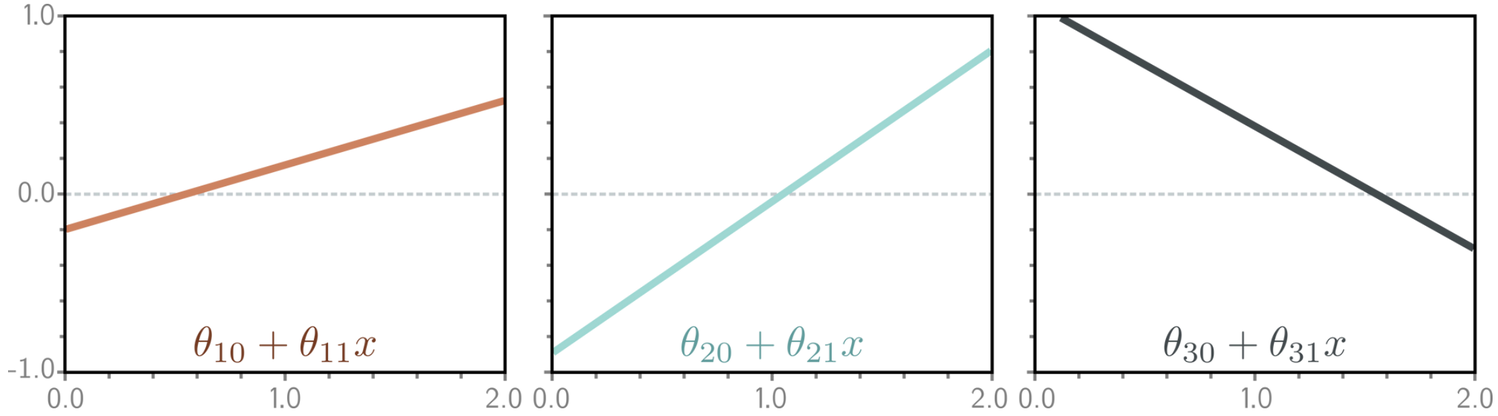
$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$



ϕ= $(\phi_0 \quad \phi_1 \quad \phi_2 \quad \phi_3)$

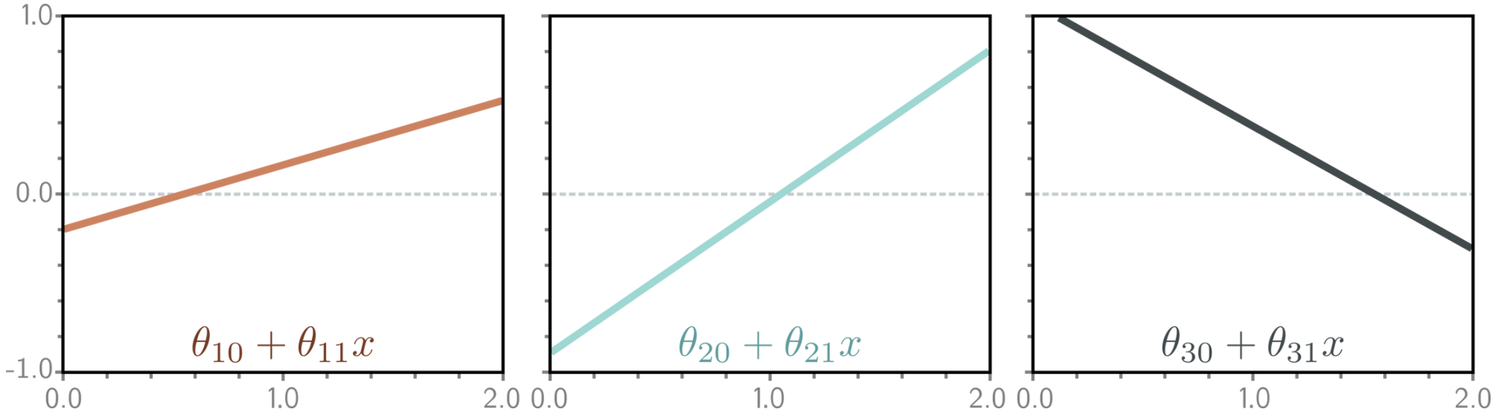
Computing the Hidden Layer

First we need to compute the product between the input and the first weights vector **θ**

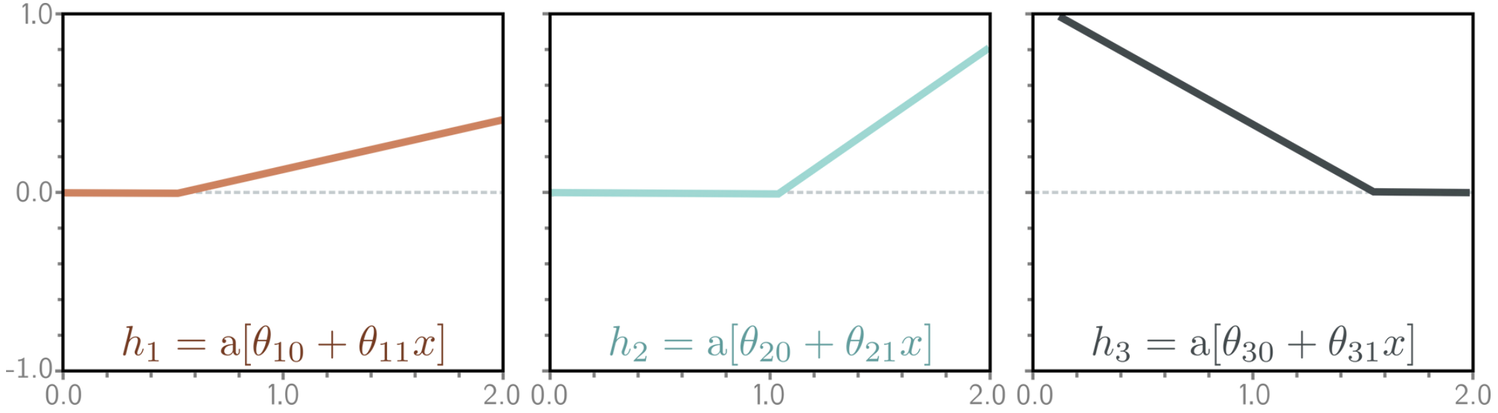


Computing the Hidden Layer

First we need to compute the product between the input and the first weights vector **θ**

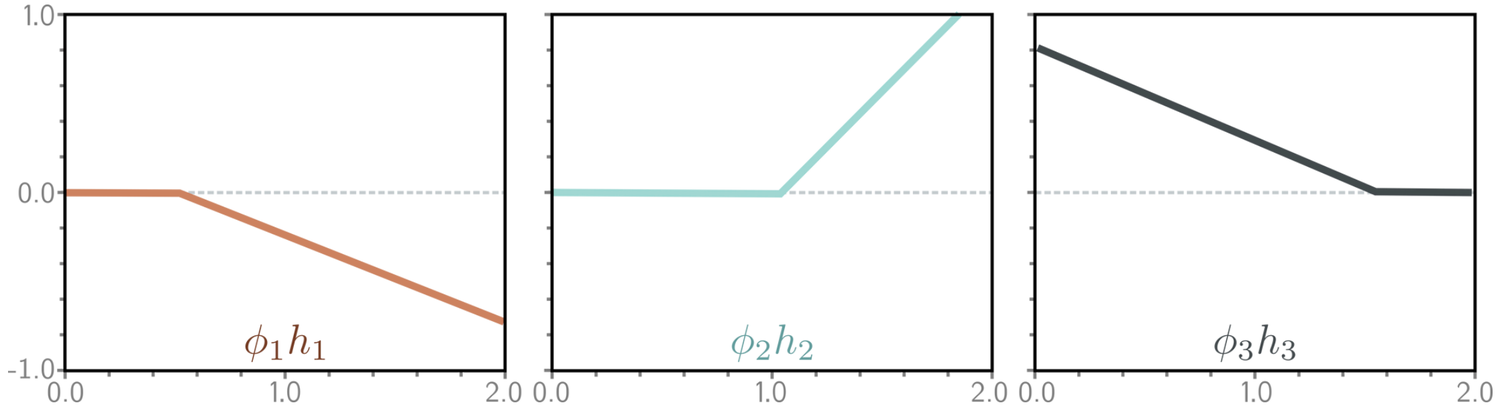


Then we apply the activation function (ReLU) to get the hidden state vector **h**



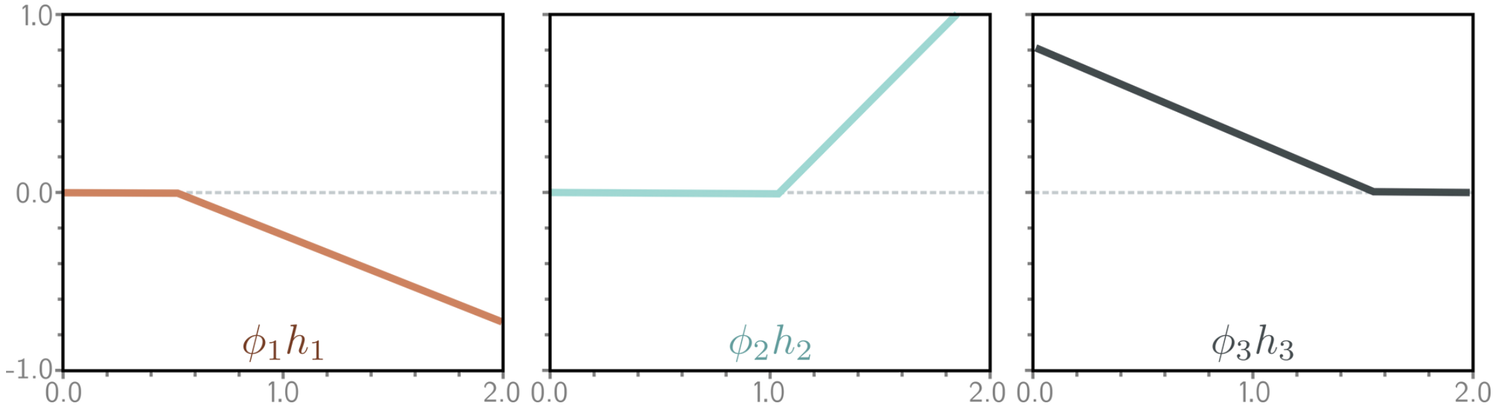
Computing the Output

We then multiply the hidden vector state for the second weight vector **ϕ**

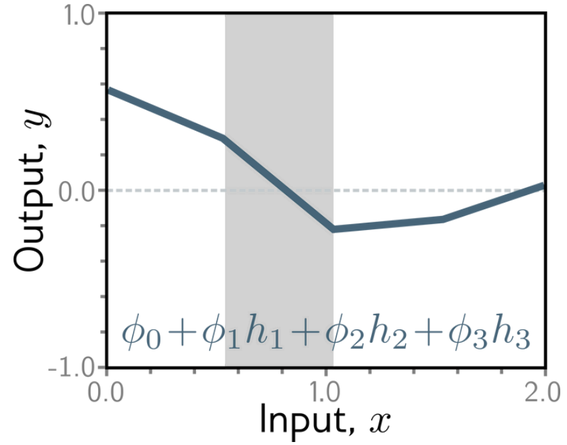


Computing the Output

We then multiply the hidden vector state for the second weight vector **ϕ**



Finally we sum the weighted hidden units to get the output y

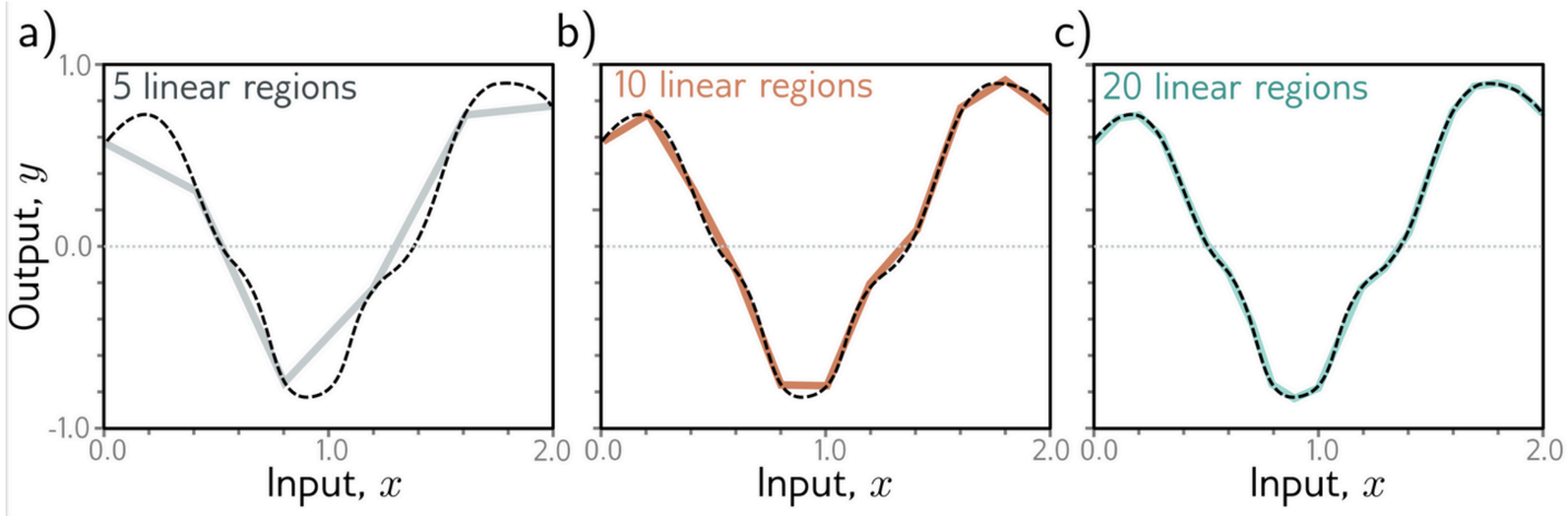


Universal Approximation Theorem

The **Universal Approximation theorem** states that

A shallow neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function

This explains why even relatively simple networks can model complex phenomena, though it does not provide a method to find the optimal network parameters.



https://giordano-demarzo.github.io/teaching/deep-learning-25/shallow_nn/

Universität
Konstanz

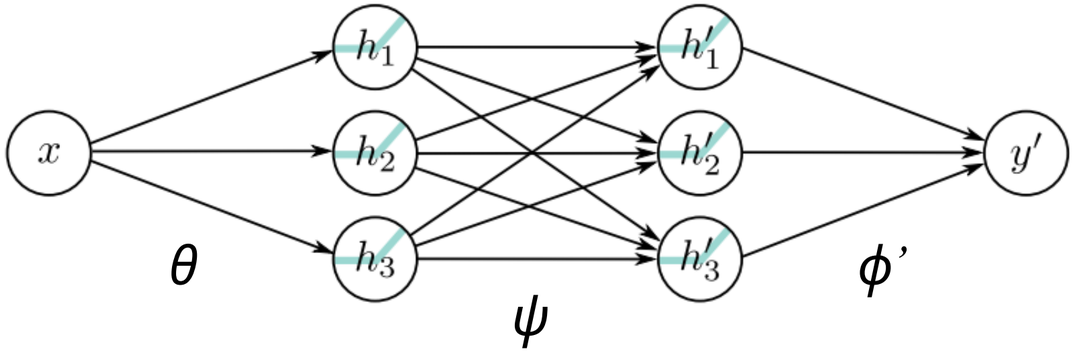
Multilayer Perceptron Architecture

Deep Neural Networks: Multilayer Perceptron

A **deep neural network** is a **feed-forward** neural network with at least **two hidden layers.**

The multilayer perceptron (MLP) is the most simple example of deep neural network:

- it is composed of an input layer (x), an output layer (y) and at least 2 hidden layers (h, h')
- each neuron
  - takes the outputs of the neurons of the previous layer
  - weights (θ, ψ, ϕ) and then sums these outputs, also including the bias
  - computes and then outputs the result of the (non-linear) activation function (a)



Mathematical Representation of MLP

The first hidden layer processes the input

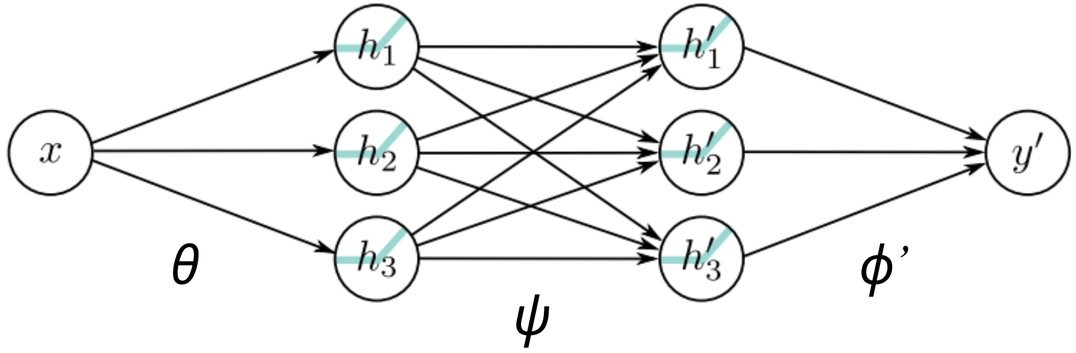
$$
\begin{aligned}
h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\
h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\
h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x].
\end{aligned}
$$

The second hidden layer processes the output of the first hidden layer

$$
\begin{aligned}
h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\
h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3],
\end{aligned}
$$

The output layer computes the output processing the result of the second hidden layer

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3.$$

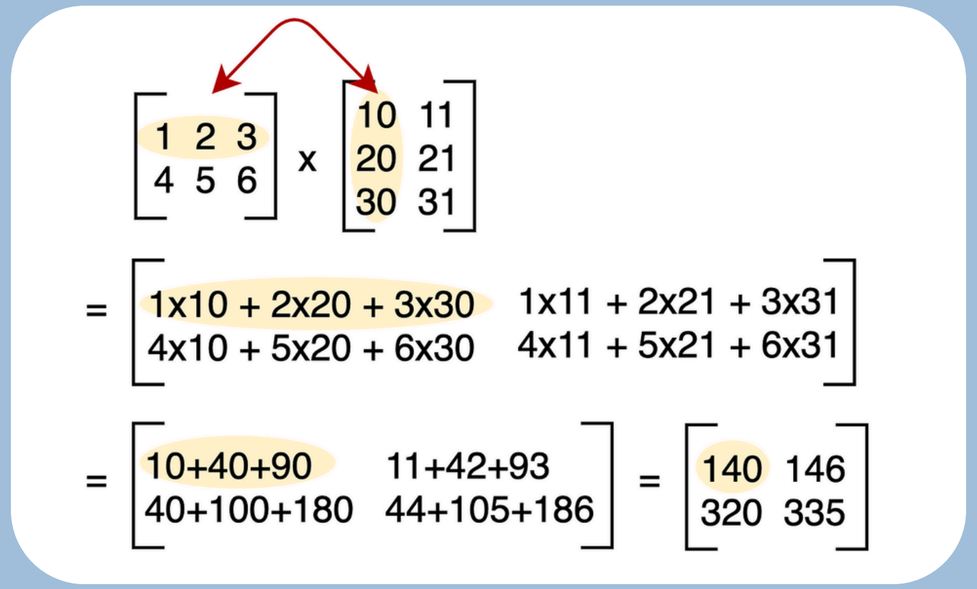


Math Recap

Matrix Multiplication

Matrix Multiplication consists in row by column multiplications:

- the elements (i, j) of the product matrix is obtained starting from the row i of the first matrix and the column j of the second matrix
- in general matrix multiplication is non commutative **A** x **B** ≠ **B** x **A**
- the number of columns of the first matrix must be equal to the number of rows of the second matrix



Mathematical Representation of MLP

The first hidden layer processes the input

$$
\begin{aligned}
h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\
h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\
h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x].
\end{aligned}
$$

$$
\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \theta_{10} \\ \theta_{20} \\ \theta_{30} \end{bmatrix} + \begin{bmatrix} \theta_{11} \\ \theta_{21} \\ \theta_{31} \end{bmatrix} x\right]
$$

The second hidden layer processes the output of the first hidden layer

$$
\begin{aligned}
h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\
h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3],
\end{aligned}
$$

$$
\begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \psi_{10} \\ \psi_{20} \\ \psi_{30} \end{bmatrix} + \begin{bmatrix} \psi_{11} & \psi_{12} & \psi_{13} \\ \psi_{21} & \psi_{22} & \psi_{23} \\ \psi_{31} & \psi_{32} & \psi_{33} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}\right]
$$

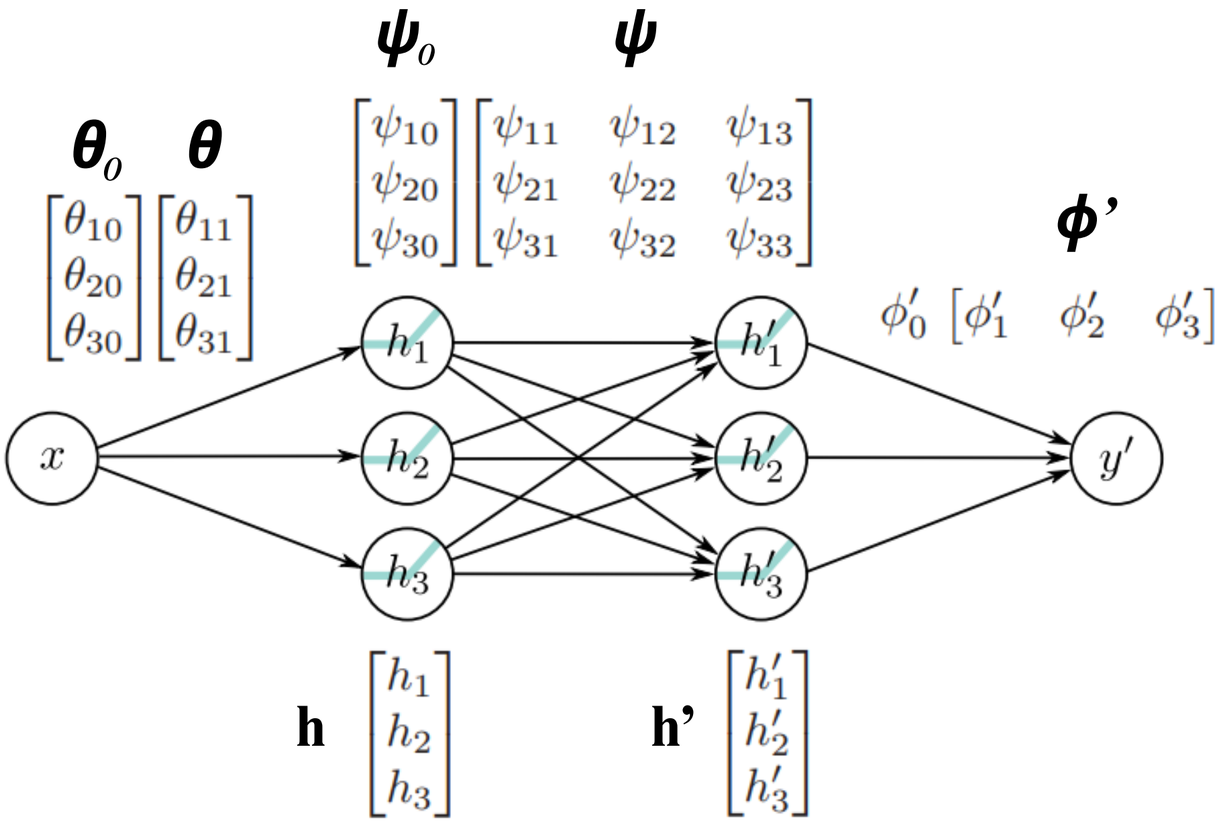
The output layer computes the output processing the result of the second hidden layer

$$
y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3.
$$

$$
y' = \phi'_0 + \begin{bmatrix} \phi'_1 & \phi'_2 & \phi'_3 \end{bmatrix} \begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix},
$$

Matrix Notation

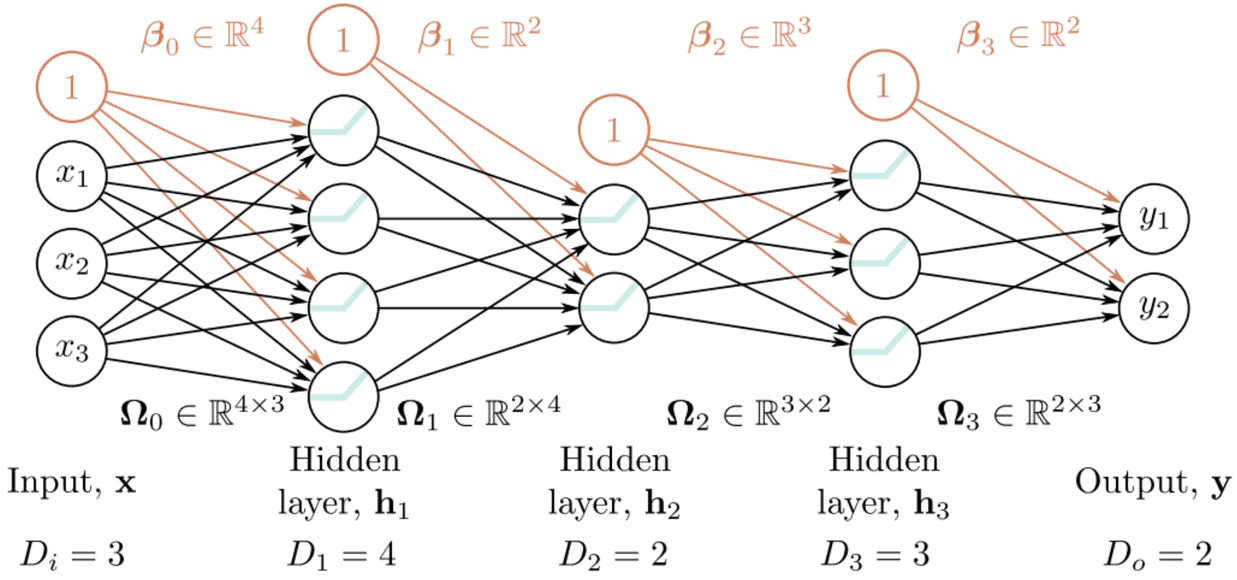
We can write this same neural network in matrix notation. Bold letters denote vectors and mtatrices.



$$
\begin{aligned}
\mathbf{h} &= \mathbf{a}\left[\boldsymbol{\theta}_0 + \boldsymbol{\theta}x\right] \\
\mathbf{h}' &= \mathbf{a}\left[\boldsymbol{\psi}_0 + \boldsymbol{\Psi}\mathbf{h}\right] \\
y' &= \phi_0' + \boldsymbol{\phi}'\mathbf{h}',
\end{aligned}
$$

General Formulation

In general we will have K hidden layers $\mathbf{h_1}...\mathbf{h_k}$ and K+1 weight matrices $\mathbf{\Omega_0}...\mathbf{\Omega_k}$ and bias vectors $\mathbf{\beta_0}...\mathbf{\beta_k}$
Matrix notation is useful to represent large neural networks in a compact notation.

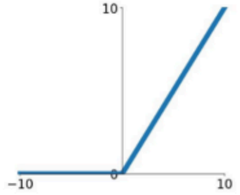


$$
\begin{aligned}
\mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}] \\
\mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1] \\
\mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2] \\
&\vdots \\
\mathbf{h}_K &= \mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{h}_{K-1}] \\
\mathbf{y} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{h}_K.
\end{aligned}
$$

Activation Functions for Hidden Layers

ReLU

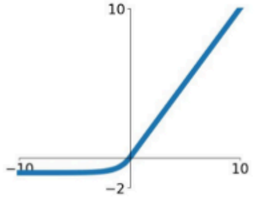
$\max(0, x)$



- Standard option
- Fast computation
- No gradient vanishing

ELU

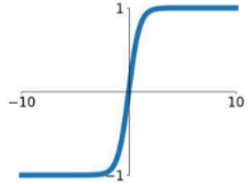
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



- Possible replacement for ReLU
- No gradient sparsity

tanh

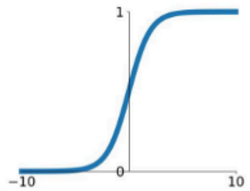
$\tanh(x)$



- Not suitable for dense and convolutional layers:
- Gradient vanishing
- Used in RNN and GAN

Sigmoid

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

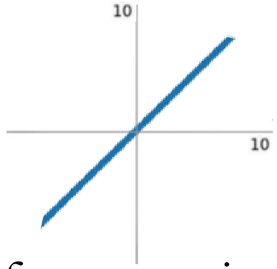


- Similar uses and problems of tanh

Activation Functions for Output Layer

Linear

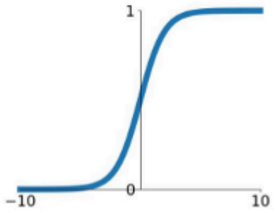
x



- Standard option for regression tasks

Sigmoid

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



- used in binary classification problems (2 classes)
- single output neuron

Softmax

$$\text{SoftMax}(x_i) = \frac{e^{x_i}}{\sum_j^N e^{x_j}}$$

- Used in multi-class classification problems
- N output neurons
- Output of each neuron is in (0,1) and is interpretale as a probability

Let's Make it Easy

This may seems very complex, but the concept is very easy:

- a **DNN** is just a very **complex function with many parameters** (weights and biases)
- this function takes an input, typically an high-dimension vector, and returns an output

Then why using all those neurons and matrices? Can't we just use a standard function with a million parameter and fit this function to the data like we would do with a linear fit?

- possible in theory
- impossible in practice
- DNN are just a very **computationally efficient** way to represent and fit arbitrary complex functions

Our goal is thus to adjust the weights and biases of the neural network to make it represent the function we want (that better fitting real data).

Universität
Konstanz

Training
Deep Neural Networks

Training a Multilayer Perceptron

Training a MLP means finding the best weight matrices and bias vectors that make the MLP output being closer to the desired output. For instance, in a classification task the output must be 0 and 1 depending on whether the input belongs to a class rather than the other.
This requires three main ingredients:

- a **Loss Function** that quantifies how good is the MLP in performing the desired task (how close is its output to the desired output)
- an **Optimization Algorithm** that determines how the parameters of the MLP should be updated to make the Loss decrease
- the **Backpropagation** algorithm, used to compute the derivatives needed in the Optimization Algorithm

Parameters and Hyperparameters

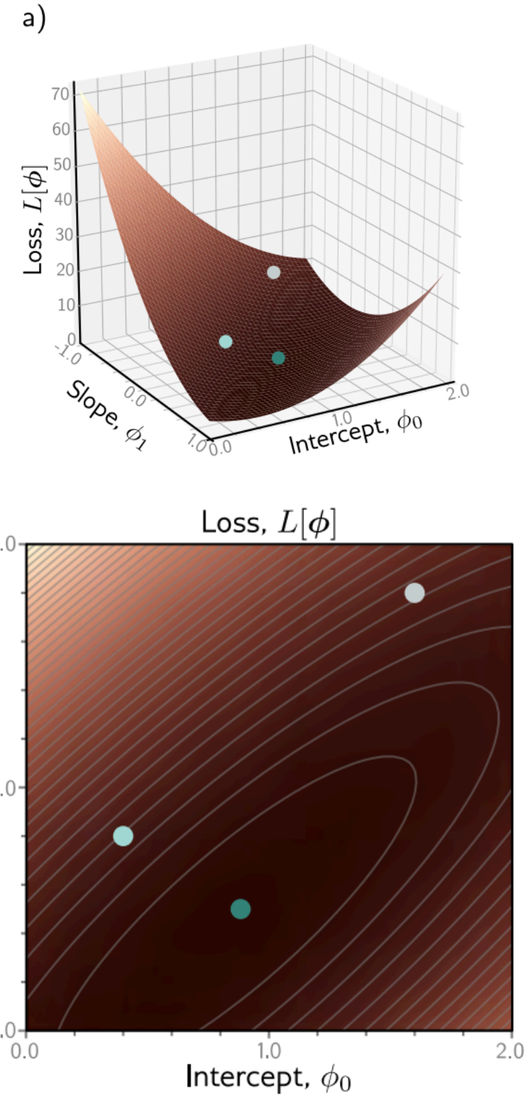
Before learning how to train a MLP we have to distinguish between the Network's Parameters and Hyperparameters

- **Parameters** These are the weights $\Omega_0 ... \Omega_k$ and biases $\beta_0 ... \beta_k$ of the MLP that are learnt during the iterative learning process
- **Hyperparameters** These are parameters that must be manually adjusted or tuned using alternative techniques (e.g. grid search). They include:
  - neural network architecture (number of layers, number of neurons, activations...)
  - batch size
  - learning rate
  - number of epochs
  - optimization algorithm

Loss Function

The **Loss L** is a function that quantifies how distant is the output of the neural network from the ground truth (the y of the training data)

- it is a function of the neural network parameters $L=L[\Omega_0...\Omega_k,\beta_0...\beta_k]$
- it lives in a very highly dimensional space (brute force is impossible)
- the goal is to find the minimum of the Loss function, corresponding to the parameters that produce the best output (closest to the ground truth)



Loss Functions for Regression

In a regression task the neural network receives an input vector $\mathbf{x}_i=(x_{i1}, x_{i2},...)$ and is tasked with predicting a continuous value. We denote by $y_i=y_i(\mathbf{x}_i \mid \boldsymbol{\Omega}_0...\boldsymbol{\Omega}_k,\boldsymbol{\beta}_0...\boldsymbol{\beta}_k)$ the neural network output and by $\hat{y}_i$ the ground truth. The most common options for the loss are

- **Mean Square Error**
  - Most popular option
  - Strongly penalizes outliers

$$L[\boldsymbol{\Omega}_0 \dots \boldsymbol{\Omega}_k, \beta_0 \dots \beta_k] = \frac{1}{N} \sum_i^N \left[y_i(\mathbf{x}_i | \boldsymbol{\Omega}_0 \dots \boldsymbol{\Omega}_k, \beta_0 \dots \beta_k) - \hat{y}_i \right]^2$$

- **Mean Absolute Error**
  - Good alternative
  - Less importance to outliers

$$L[\boldsymbol{\Omega}_0 \dots \boldsymbol{\Omega}_k, \beta_0 \dots \beta_k] = \frac{1}{N} \sum_i^N \left| y_i(\mathbf{x}_i | \boldsymbol{\Omega}_0 \dots \boldsymbol{\Omega}_k, \beta_0 \dots \beta_k) - \hat{y}_i \right|$$

Loss Functions for Classification

In a classification task the neural network receives an input vector $\mathbf{x_i}=(x_{i1}, x_{i2},...)$ and is tasked with predicting to which class the input belongs to. We denote by $\mathbf{y_i}=\mathbf{y_i}(\mathbf{x_i} \mid \mathbf{\Omega_0}...\mathbf{\Omega_k},\mathbf{\beta_0}...\mathbf{\beta_k})$ the neural network output and by $\mathbf{\hat{y}_i}$ the ground truth. Note that in this case the output can be a vector

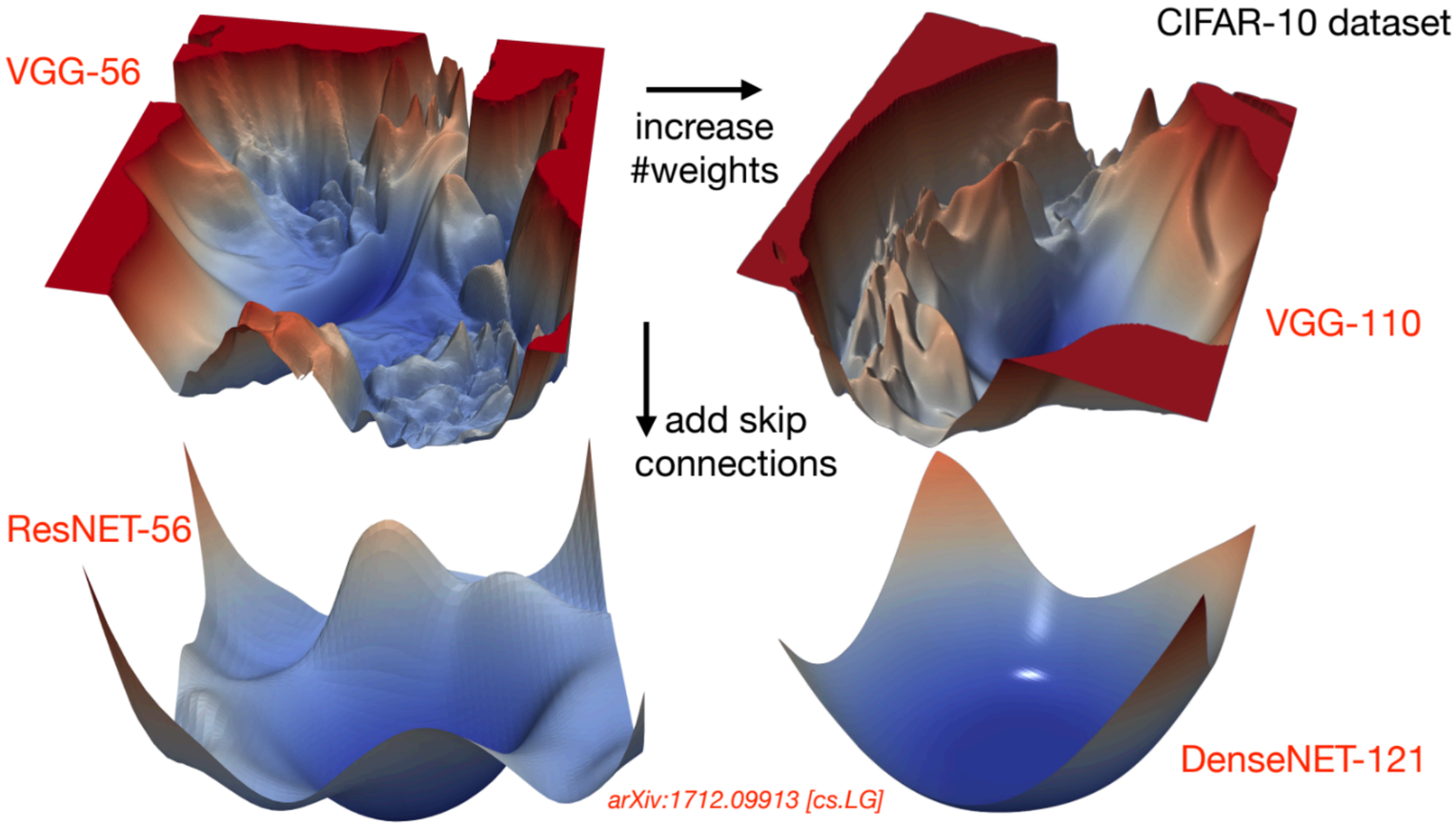
- **Binary Cross Entropy**
  - Most popular option for binary classification tasks
  - Single neuron output + sigmoid (or similar)

$$L[\mathbf{\Omega}_0 \dots \mathbf{\Omega}_k, \beta_0 \dots \beta_k] = -\frac{1}{N}\sum_i^N [\hat{y}_i \log(y_i) + (1-\hat{y}_i) \log(1-y_i)]$$

- **Categorical Cross Entropy**
  - Most popular option for multi-class classification tasks
  - Multiple neurons output + softmax

$$L[\mathbf{\Omega}_0 \dots \mathbf{\Omega}_k, \beta_0 \dots \beta_k] = -\frac{1}{N}\sum_i^N \sum_c^M \hat{y}_{i,c} \log(y_{i,c}) = -\frac{1}{N}\sum_i^N \log(y_{i,c^*})$$

Why going Deep?



Math Recap

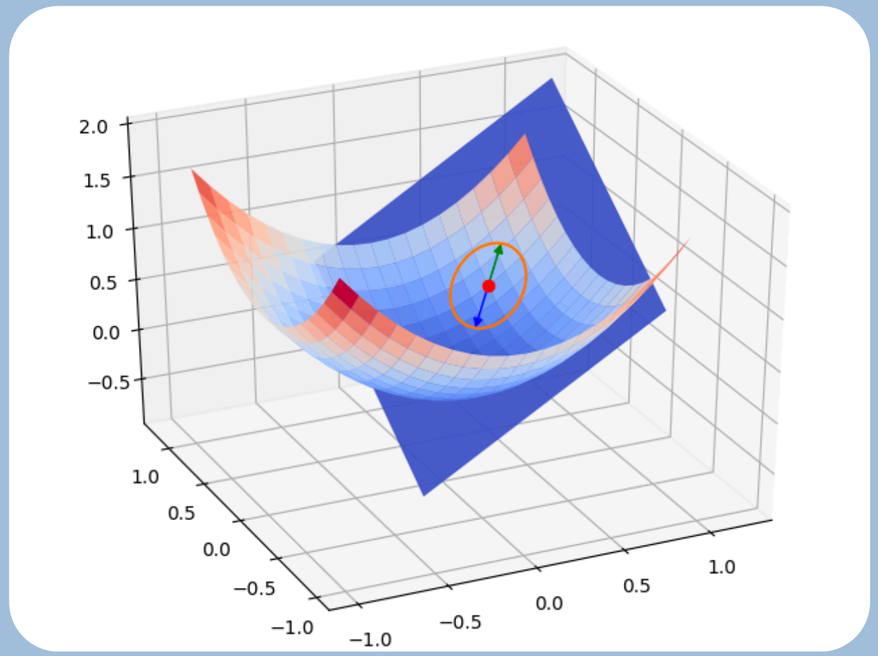
Gradient of a Function

Let us consider a function $f(x_1, x_2, \ldots x_n)$ from R^n (input in n dimensions) to R (output is a real number)

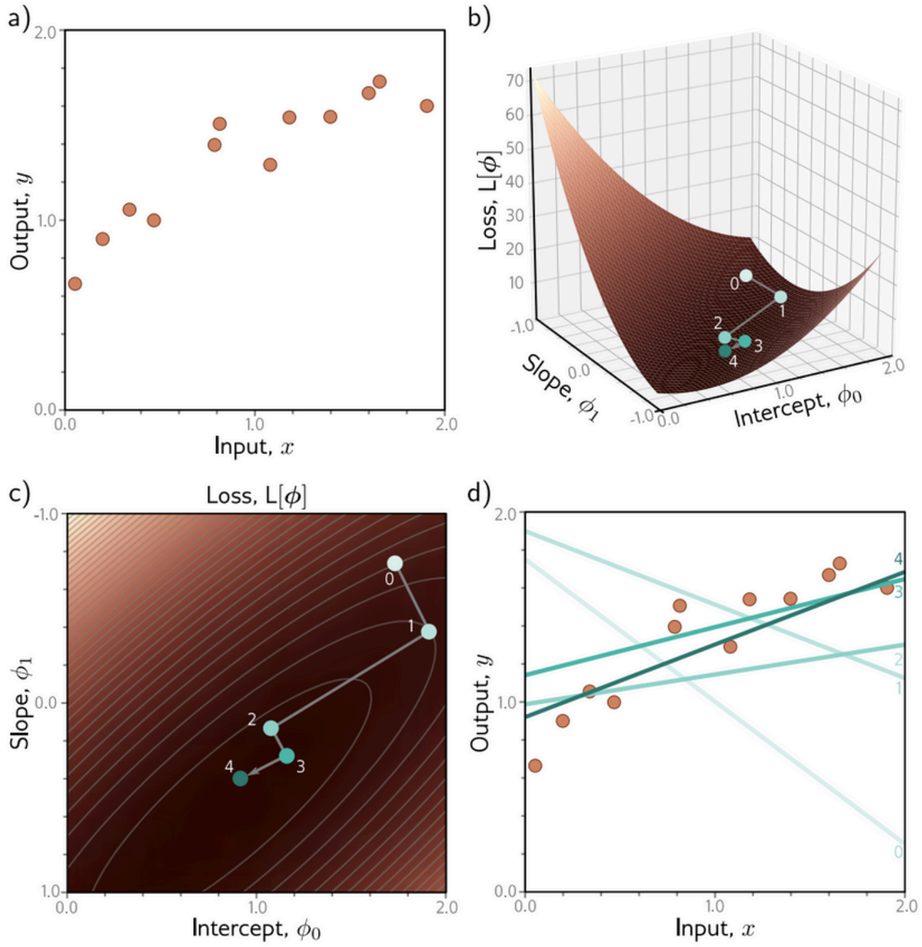
- the gradient is a generalization of the derivative for higher dimensions
- it is a vector that contains all the derivatives of the function with respect to each of its n input variables

$$\nabla f = \left(\frac{df}{dx_1}, \frac{df}{dx_1}, \ldots, \frac{df}{dx_n} \right)$$

- it points in the direction of the greatest rate of increase of the function



Gradient Descent



The **Gradient Descent** is an iterative algorithm that uses the gradient of the Loss function to compute how to update the weights in order to make the Loss decrease. It consists of two steps

- we denote by $\mathbf{W_t}$ the set of all weights and biases at iteration t and we compute the gradient of the Loss in $\mathbf{W_t}$

$$\nabla L(\mathbf{W}_t) = \left(\frac{dL}{d\Omega_{0,0}}, \frac{dL}{d\Omega_{0,1}} \cdots \frac{dL}{d\beta_{0,0}}, \frac{dL}{d\beta_{0,1}} \cdots . \right)_{\mathbf{W}_t}$$

- we use the gradient to update the weights

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \nabla L(\mathbf{W}_t)$$

The process is repeated until the weights stop to change (the minimum is reached).

Stochastic Gradient Descent

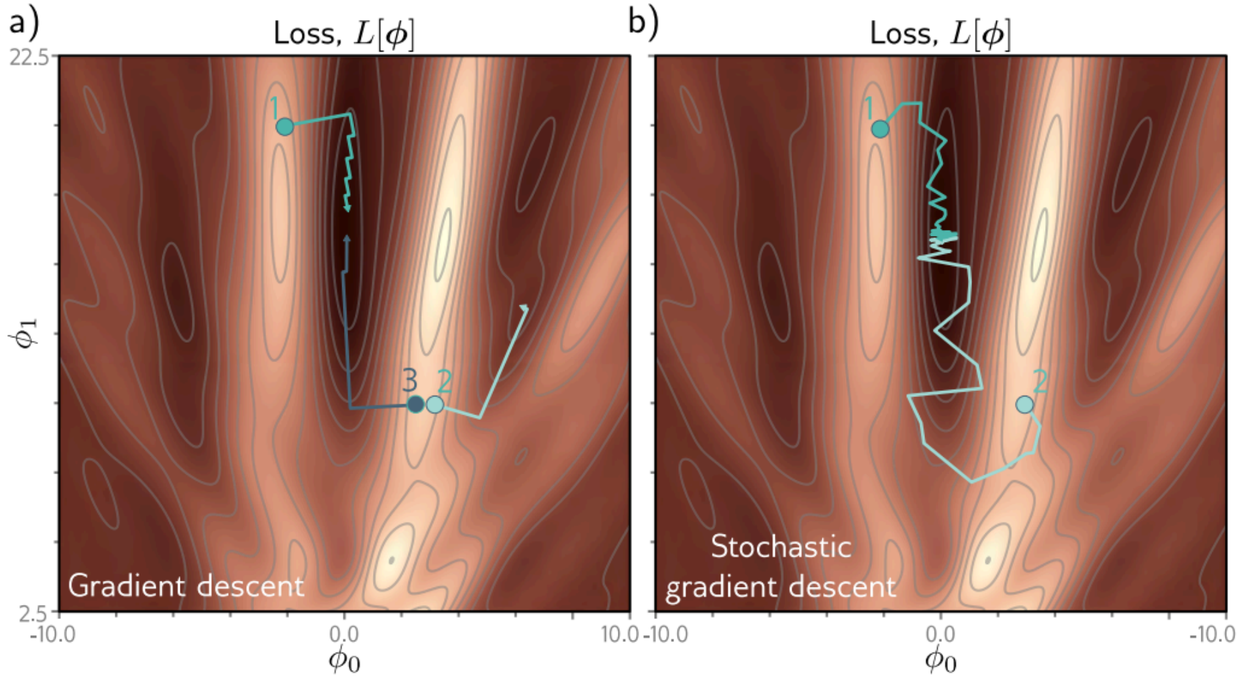
The Gradient Descent has some limits:

- it becomes computationally inefficient for large datasets
- it may get stuck in local minima

For these reasons we introduce the

Stochastic Gradient Descent

- the dataset is split into mini-batches (whose dimension is called batch size)
- the weights are updated one mini-batch at a time



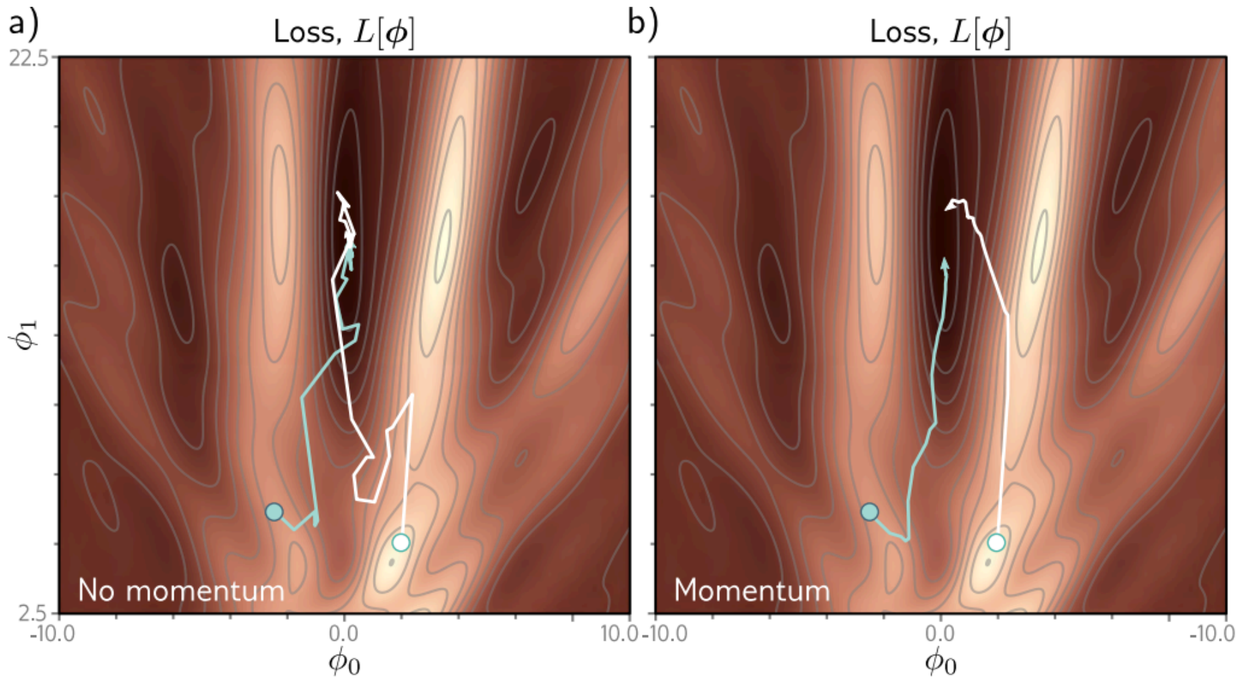
Momentum

In order to reduce fluctuations due to the stochastic nature of the algorithm we can introduce momentum. This means taking into account the "velocity" at the previous time step and not only the gradient of the Loss

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \mathbf{v}_{t+1}$$

$$\mathbf{v}_{t+1} = \alpha \mathbf{v}_t + (1-\alpha)\nabla L(\mathbf{W}_t)$$

Typically α~0.9-0.99. This makes the convergence faster and more regular.



Other Optimization Algorithms

The Stochastic Gradient Descent is a good algorithm, but in Deep Neural Networks it is not enough. There are many other algorithms to improve convergence:

- ADA
- ADAdelta
- RMSProp
- ADAM

These algorithms include momentum and an adaptive learning rate, that varies over time and is different for each individual weight. In most practical applications ADAM and RMSProp are generally the best options.

Universität
Konstanz

Backpropagation Algorithm

Deriving the Loss

In order to implement any optimization algorithm we need to compute the derivative of the Loss with respect to each parameter. The Loss is an extremely complex function living in an highly dimensional space. In order to compute its gradient efficiently we use the **Backpropagation Algorithm**:

- complex name, simple concept: compute derivatives
- it relies on the chain rule of derivatives
- it consists of two steps:
  - a forward pass
  - a backward pass

Using the Backpropagation makes computing the gradients as time consuming as computing the Loss, but a lot of memory is required (~10x model size).

Math Recap

Chain Rule for Derivatives

The chain rule is a simple mathematical rule for computing the function of composite functions. Let us consider two functions f(x) and g(x), we want to compose these functions to get F=f(g(x)) and then compute the derivative of this composed function with respect to x. The general rule is

$$\frac{dF}{dx} = \frac{dF}{dg}\frac{dg}{dx}$$

Let us consider for example the functions f(x)=sin(x), g(x)=log(x), leading to F(x)=sin(log(x))

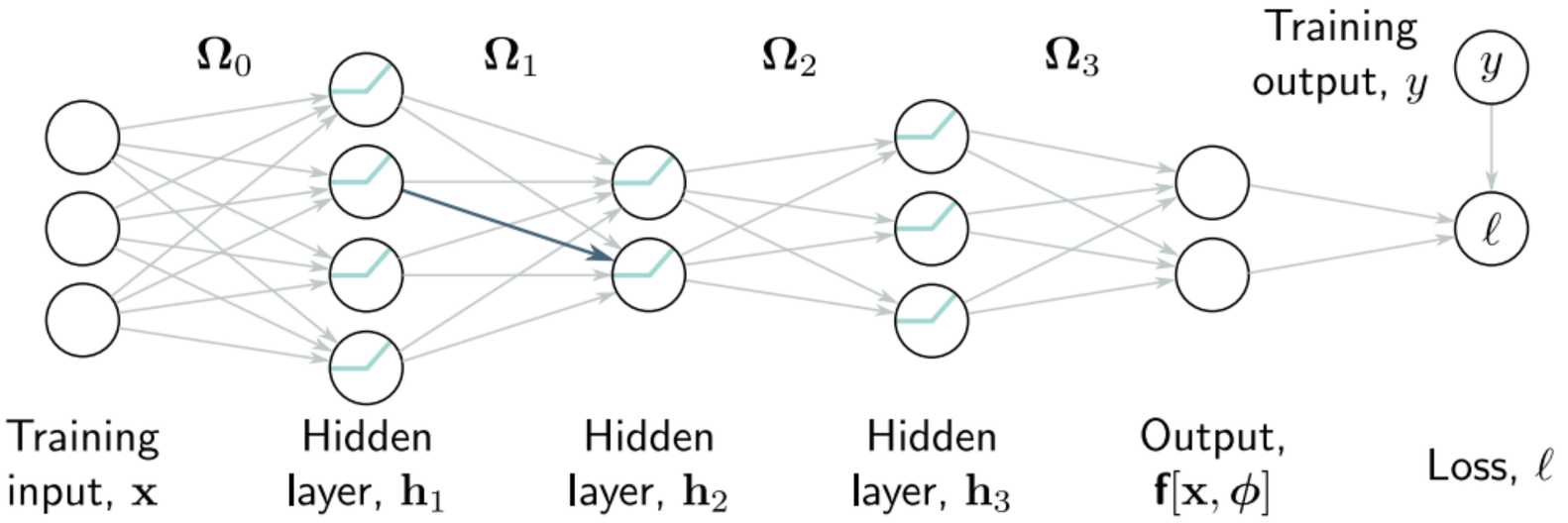
$$\frac{dF}{dx} = \frac{dF}{dg}\frac{dg}{dx} = \frac{d\sin(\log(x))}{d\log(x)}\frac{d\log(x)}{dx} = \cos(\log(x)) \cdot \frac{1}{x}$$

Forward Pass

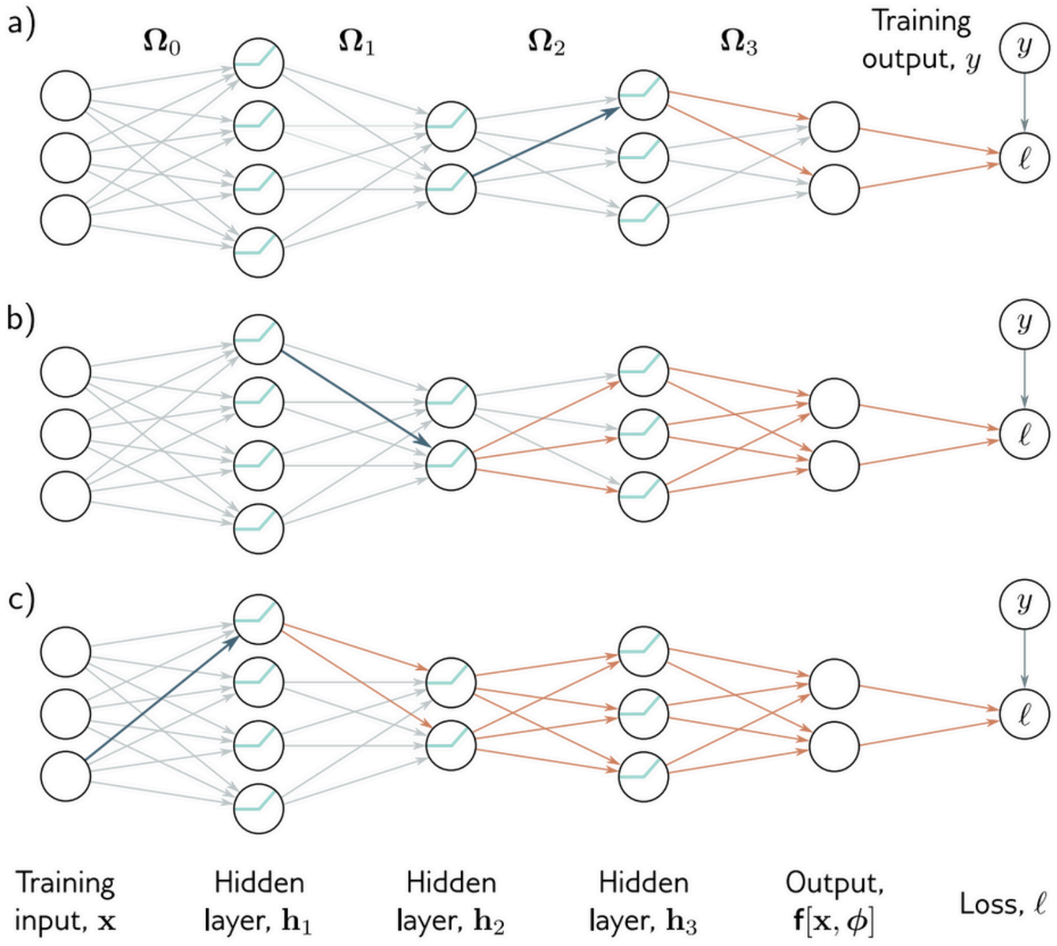
In the forward pass the DNN is feed with the training examples and it output is computed.

Moreover, for each neuron, the following quantities are stored

- the activation of each neuron h
- the argument (pre-activation) of each activation f



Backward Pass



In the backward pass, starting from the last layer and going back till reaching the first, we iteratively derive the loss with respect to each parameter. This is done by using the chain rule

- we start computing the derivative of the loss with respect to the weights in the last layer (easy)
- we compute the derivative with respect to the weights in the previous layer composing derivatives and using the chain rule to combine them

The idea is simple, for technical details look in the book, it is very well explained!

Example: Forward Pass

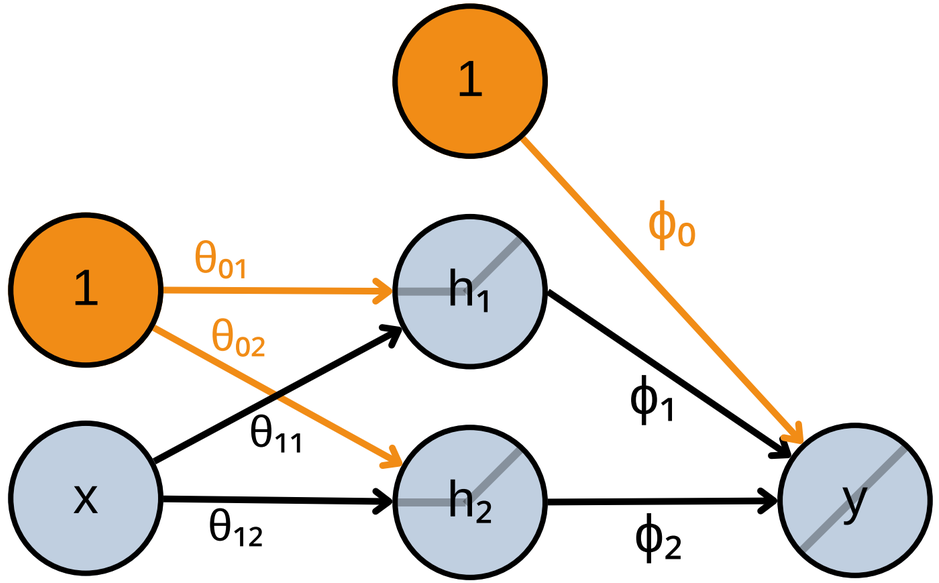
First we perform the forward pass and we compute the following quantities

- preactivations f
- hidden units h
- outputs y
- Loss L

Preactivations are the results of the **hidden units** computations before we apply the activation function

$$f_1 = \theta_{11}x + \theta_{01}, \quad h_1 = a(f_1)$$

$$f_2 = \theta_{21}x + \theta_{02}, \quad h_2 = a(f_2)$$



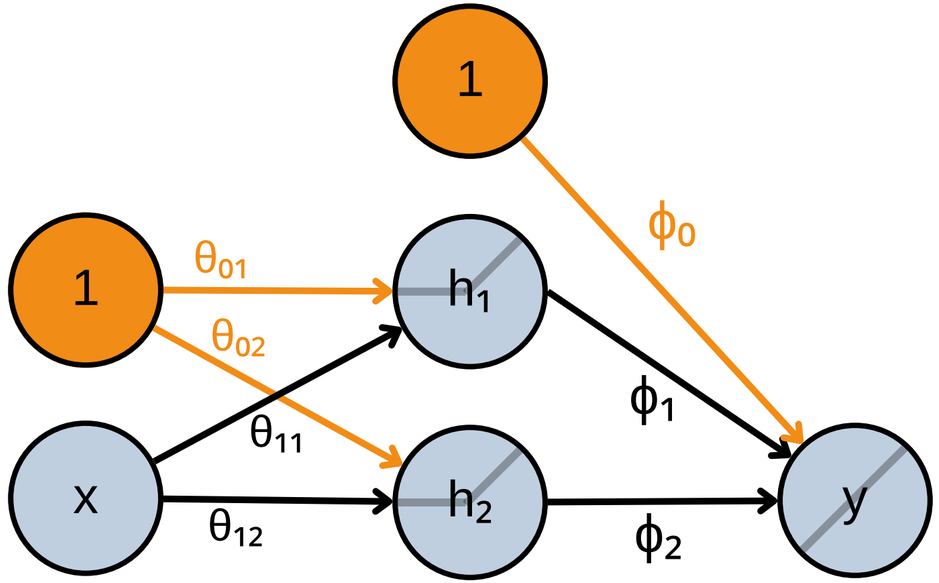
Example: Forward Pass

First we perform the forward pass and we compute the following quantities

- preactivations f
- hidden units h
- outputs y
- Loss L

The **output** of the neural network can be defined in terms of the hidden units

$$y = \phi_1 h_1 + \phi_2 h_2 + \phi_0$$



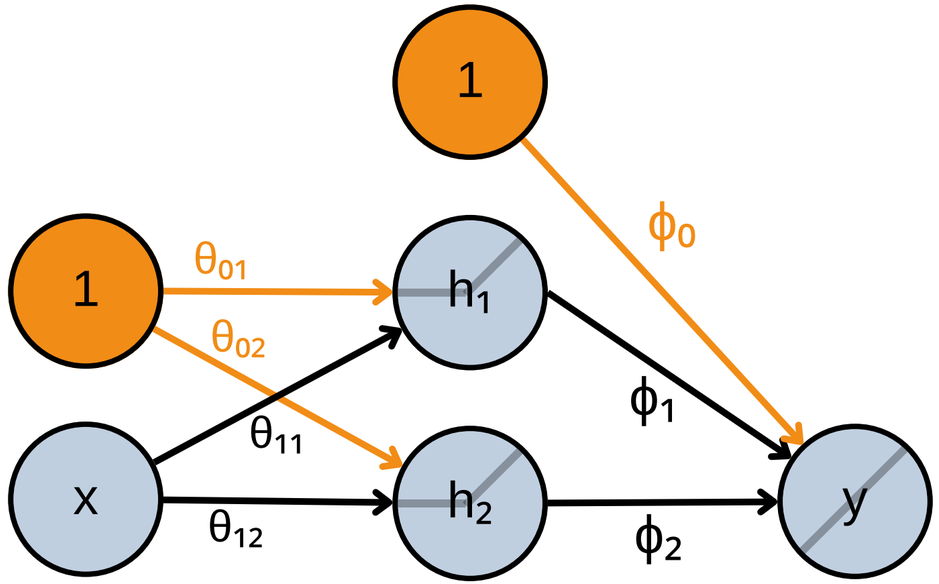
Example: Forward Pass

First we perform the forward pass and we compute the following quantities

- preactivations f
- hidden units h
- outputs y
- Loss L

Finally we use the MSE as loss. We denote by $\bar{y}$ the ground truth, so that

$$\mathcal{L} = (y - \bar{y})^2$$



Example: Backward Pass

Now we can start derive the loss with respect to the parameters of the neural network. We start with the weights of the last layer

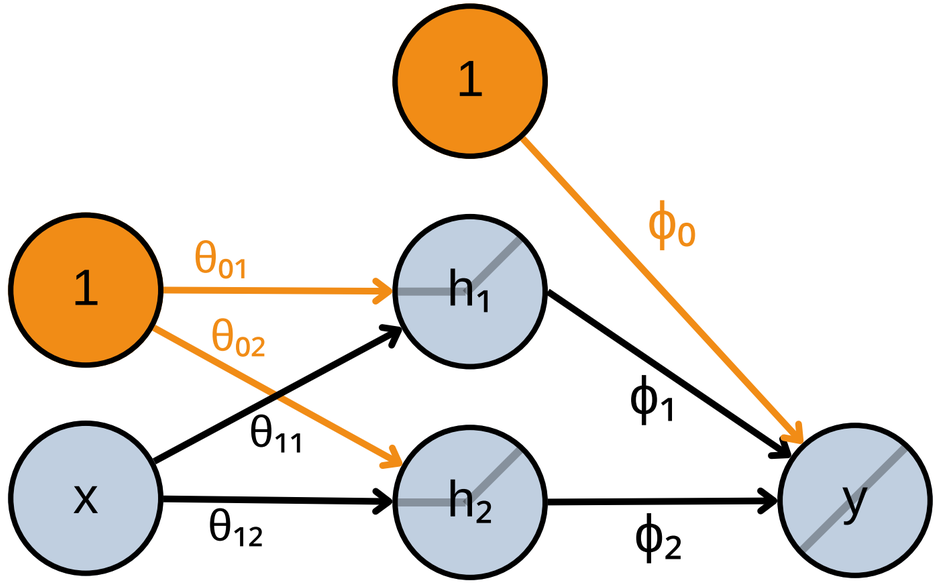
$$\frac{d\mathcal{L}}{d\phi_1} = \frac{d(y-\bar{y})^2}{d\phi_1} = 2(y-\bar{y}) \cdot \frac{dy}{d\phi_1}$$

We already computed the expression for y

$$y = \phi_1 h_1 + \phi_2 h_2 + \phi_0 \Rightarrow \frac{dy}{d\phi_1} = h_1$$

Putting all back together

$$\frac{d\mathcal{L}}{d\phi_1} = 2(y-\bar{y}) \cdot h_1 = 2\Delta y \cdot h_1$$



Example: Backward Pass

Let's see what happens when we try do derive with respect to the weights in the input layer

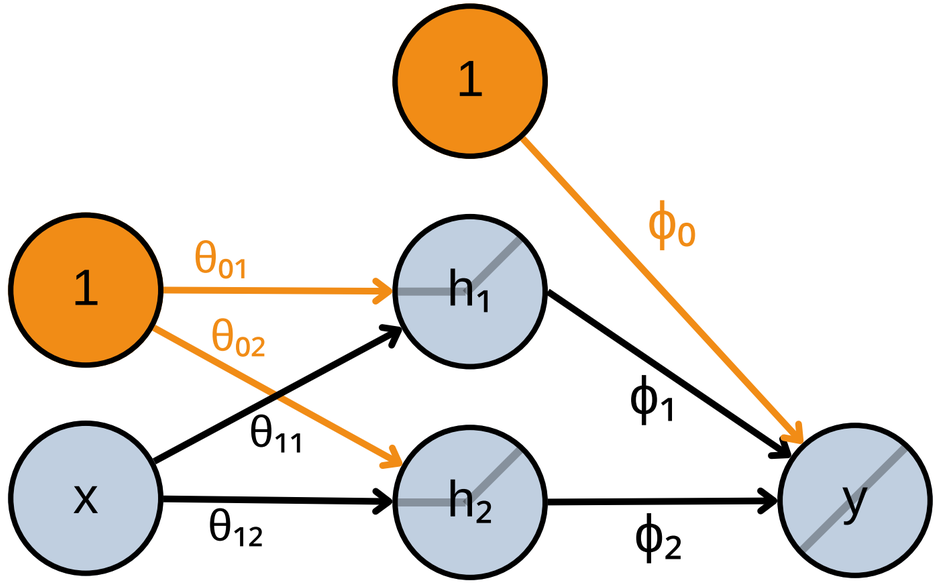
$$\frac{d\mathcal{L}}{d\theta_{11}} = \frac{d\mathcal{L}}{dy} \cdot \frac{dy}{dh_1} \cdot \frac{dh_1}{df_1} \cdot \frac{df_1}{d\theta_{11}}$$

We need to compute these 4 derivatives

- $\frac{d\mathcal{L}}{dy} = 2\Delta y$
- $\frac{dy}{dh_1} = \phi_1$
- $h_1 = a(f_1) \Rightarrow \frac{dh_1}{df_1} = a'(f_1)$
- $f_1 = \theta_{11}x + \theta_{01} \Rightarrow \frac{df_1}{d\theta_{11}} = x$

Putting all back together

$$\frac{d\mathcal{L}}{d\theta_{11}} = 2\Delta y \cdot \phi_1 \cdot a'(f_1) \cdot x$$



Example: Backward Pass

Similar results are obtained also for the biases. The idea is always the same, using the chain rule.

In summary we get for the first neuron

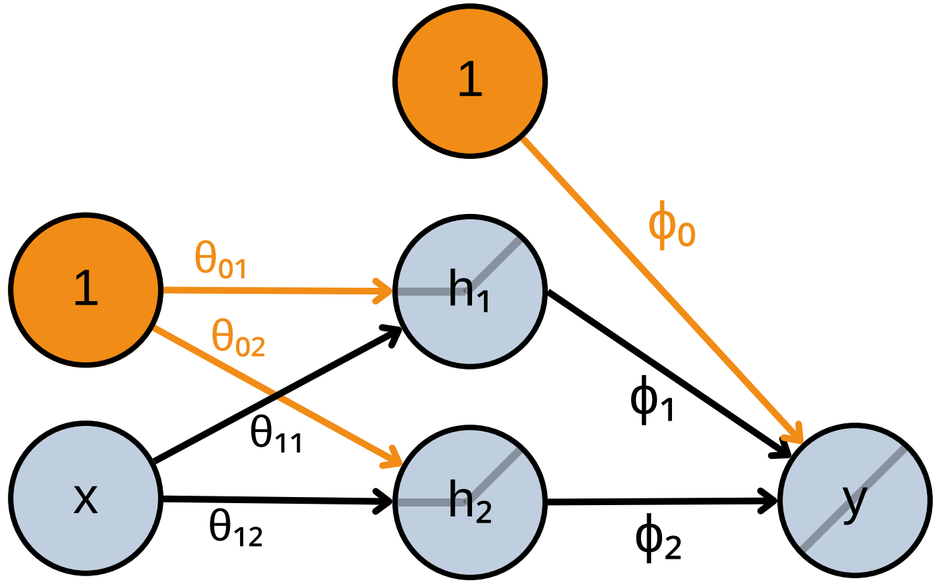
$$\frac{d\mathcal{L}}{d\phi_1} = 2\Delta y \cdot h_1$$

$$\frac{d\mathcal{L}}{d\phi_0} = 2\Delta y$$

$$\frac{d\mathcal{L}}{d\theta_{11}} = 2\Delta y \cdot \phi_1 \cdot a'(f_1) \cdot x$$

$$\frac{d\mathcal{L}}{d\theta_{01}} = 2\Delta y \cdot \phi_1 \cdot a'(f_1)$$

Try to derive these results also for the remaining parameters in the neural network. You can also try to add an hidden layer to see how things progress



Learning Curves

We now have all ingredients for training a DNN

- The training process is split in **epochs** (time steps).
- At each epoch the DNN is shown the full dataset and its parameter are updated using the backpropagation and the optimization algorithm
- In order to understand if the network is learning we have to study the learning curve (how the loss is varying epoch by epoch)
- Typically the loss start at high values and gradually decreases till reaching a plateau, where the network stop to learn

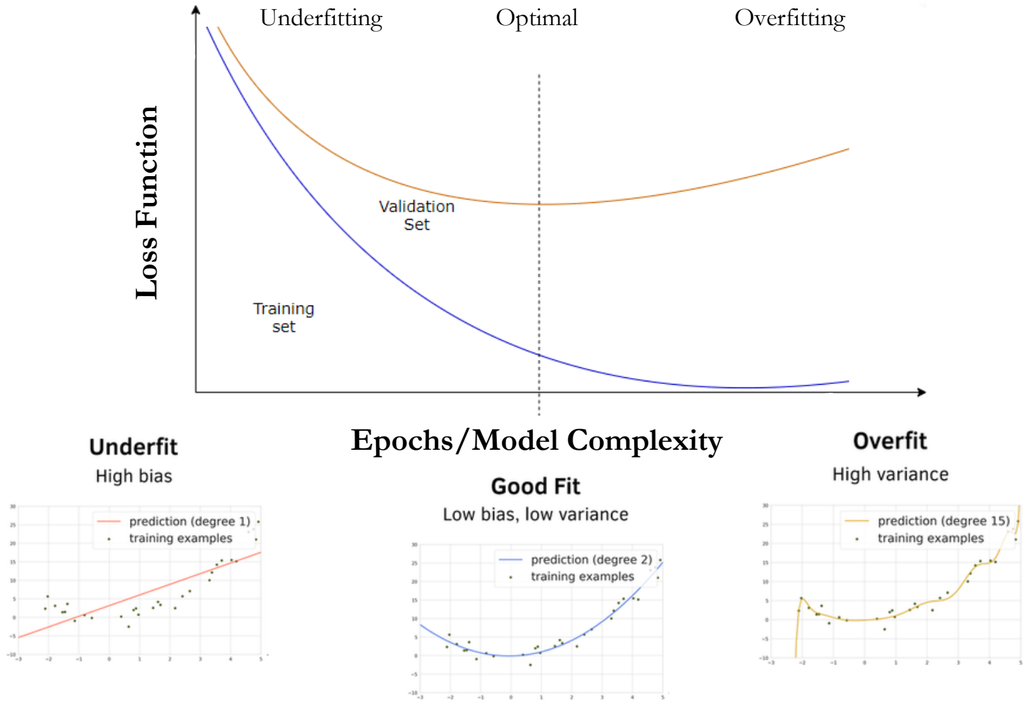
Note, however that the loss curve only tells how good is the DNN on the training data, so we could overfit the model without noticing it! For this reason we need to use also a **validation** and a **test** set.

Underfitting and Overfitting

While we training the DNN we also compute the loss on a validation set, that is not used in the training (data never seen by the DNN)

- when the train and validation loss are both high and close we are in the underfitting regime (too few epochs/parameters)
- when the training loss is low, but the validation loss is high, we overfitted the data (too many epochs/parameters)

Our goal is to find the optimum in between!

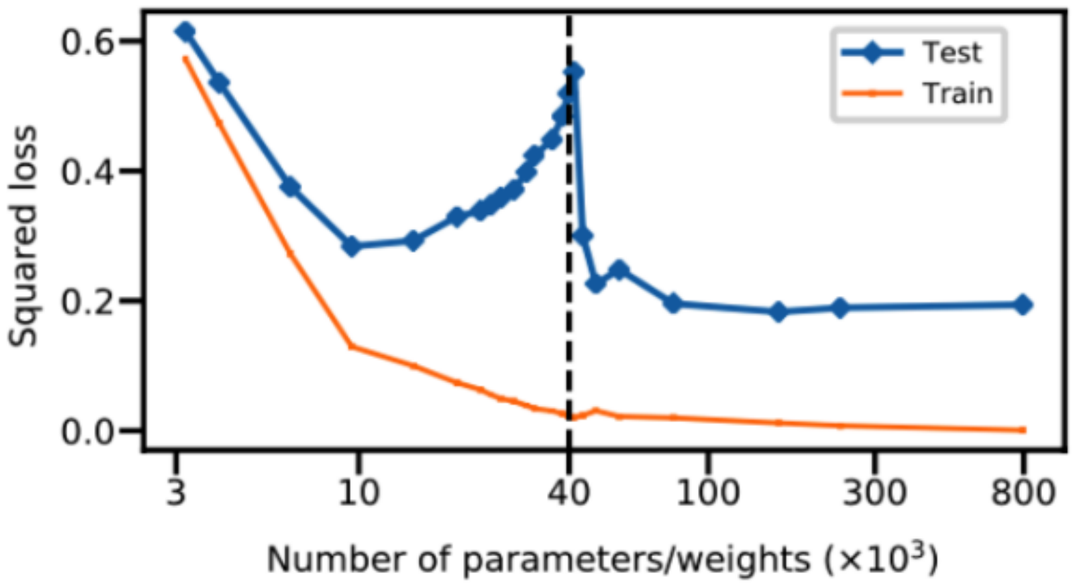


Double Descent

The picture described in the previous slide is the standard one, but it is a bit outdated due to the **Double Descent** phenomenon

- Initially the training and validation/test loss both decrease
- Then we enter the overfitting region where the validation/test loss grows
- However, if we keep increasing the epochs or the model complexity we observe a second descent of the loss

This final region is called **Inference Region.**



https://arxiv.org/abs/1912.02292

Summary

Shallow Neural Networks

Shallow Neural Networks can approximate any arbitrarily complex functions with enough hidden units. We introduce the mathematical formalism to describe these neural networks.

Multilayer Perceptron Architecture

Deep Neural Networks are neural networks with at least two hidden layers. There is nothing conceptually different with respect to shallow neural networks.

Training Deep Neural Networks

We need 3 ingredients for training a deep neural network: i. A loss function ii. An algorithm to minimize the loss (gradient descent) iii. An algorithm to derive the loss (backpropagation).

Backpropagation Algorithm

We can compute the derivative of the loss with respect to the model parameters using the backpropagation algorithm. It simply consists in applying the chain rule for derivatives.

Next Lectures

Tomorrow Coding Lab (24/04)

We will code and train our first neural network, a multilayer perceptron. You will only need your laptop and google colab for this

Next week Lecture (30/04)

We will introduce techniques to improve the training of neural networks and the first unsupervised learning architecture, the Autoencoder. This will be a mixed lecture with both theory and coding (around half and half), so bring your laptop

Following Lectures

Then for the next 2 weeks we will focus on two conceptually similar architectures: Convolutional Neural Networks and Graph Neural Networks.