Universität Konstanz

04 | Multilayer Perceptron

Giordano De Marzo

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

Deep Learning for the Social Sciences

Recap

- Supervised and Unsupervised Learning
- Regression and Classification
- Model Parameters and Loss Functions
- Shallow Neural Networks
- Deep Neural Networks

Outline

1. Multilayer Perceptron Architecture
2. Training Deep Neural Networks
3. Improving Performances
4. Autoencoder Architecture

Universität
Konstanz

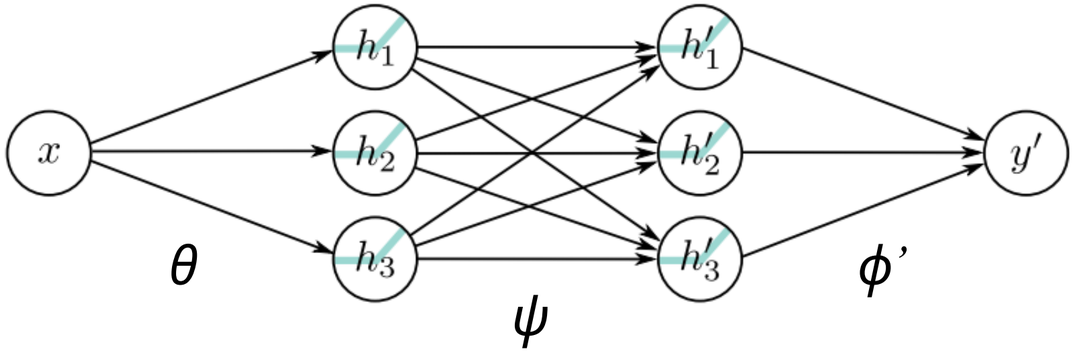
Multilayer Perceptron Architecture

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

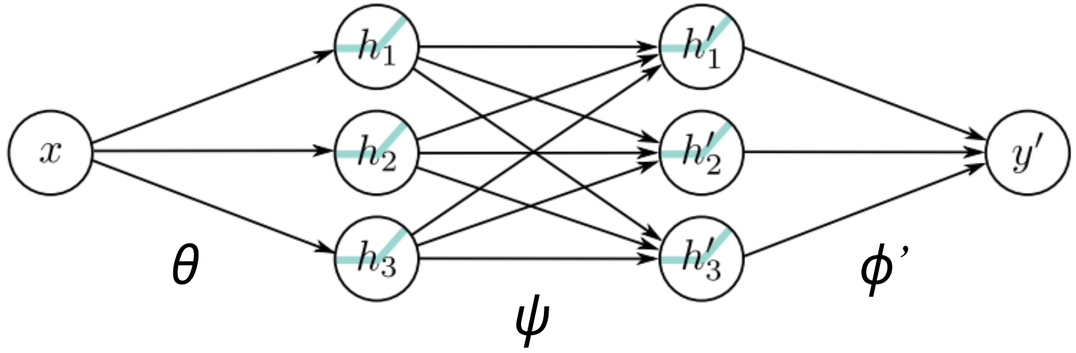
$$
\begin{aligned}
h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\
h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\
h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x],
\end{aligned}
$$

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

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

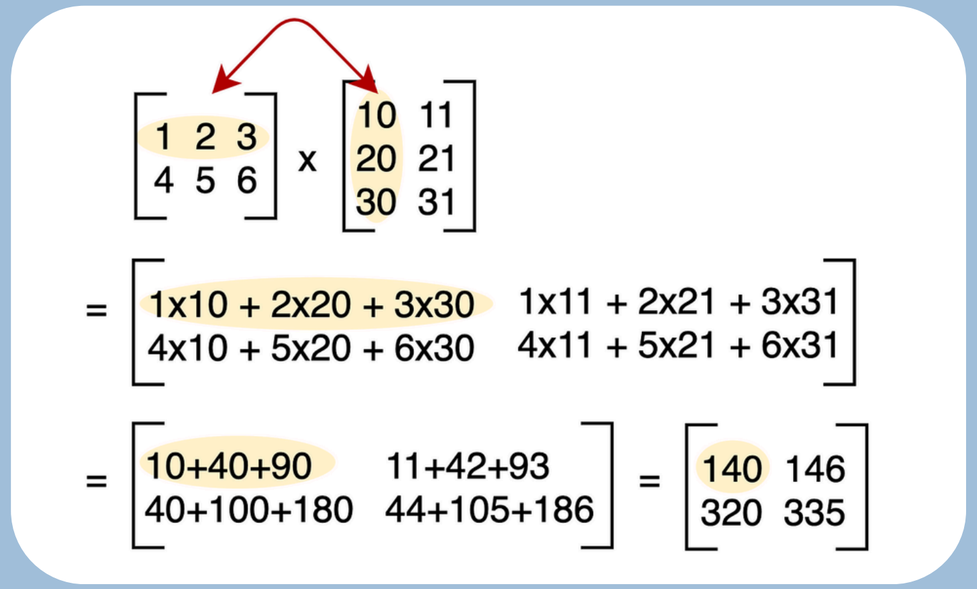


Math Recap

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

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

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

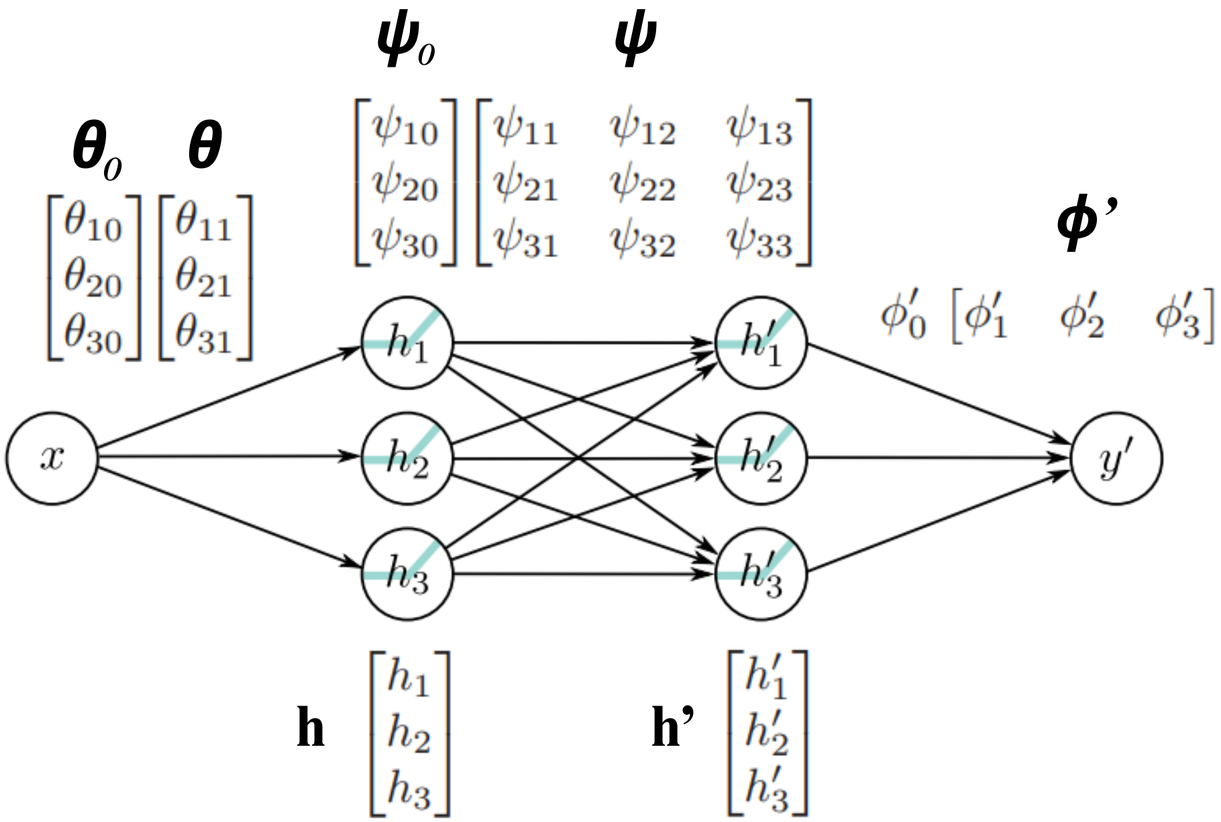
The output layer computes the output processing the result of the second hidden layer

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

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

Matrix Notation

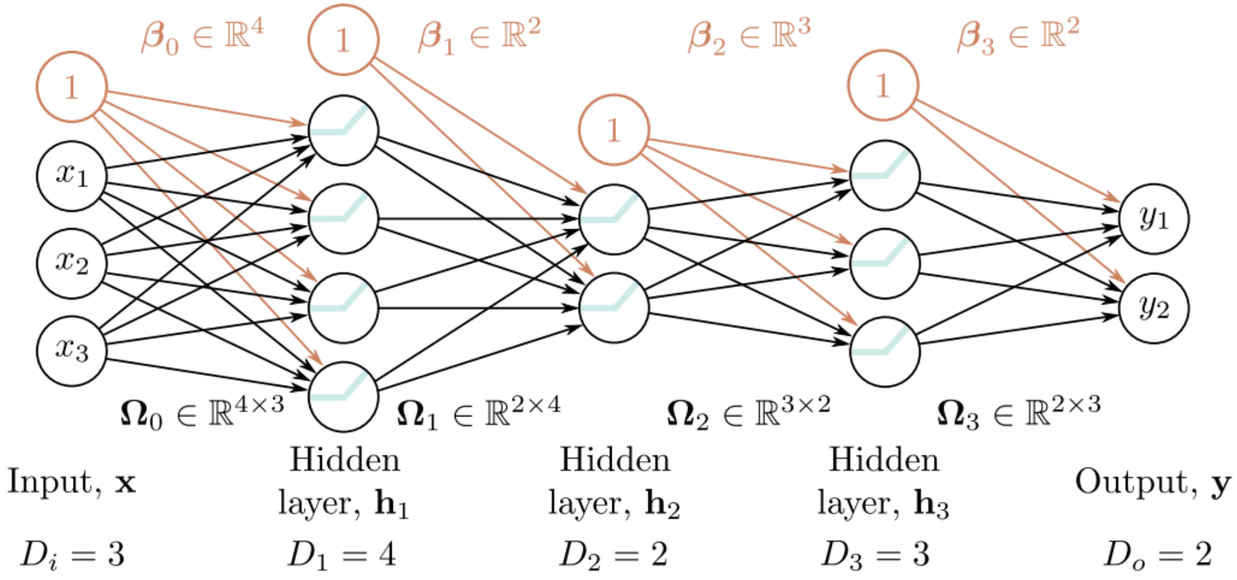
We can write this same neural network in matrix notation. Bold letters denote vectors and mtatrices.



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

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.

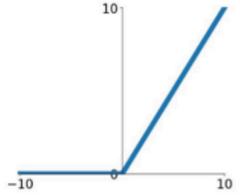


$$
\begin{aligned}
\mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}] \\
\mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1] \\
\mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2] \\
&\vdots \\
\mathbf{h}_K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

Activation Functions for Hidden Layers

ReLU

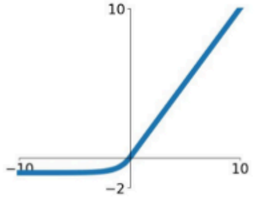
$\max(0, x)$



- Standard option
- Fast computation
- No gradient vanishing

ELU

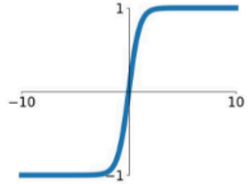
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



- Possible replacement for ReLU
- No gradient sparsity

tanh

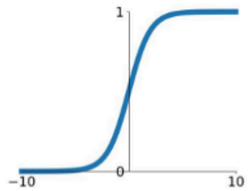
$\tanh(x)$



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

Sigmoid

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

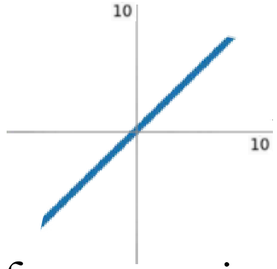


- Similar uses and problems of tanh

Activation Functions for Output Layer

Linear

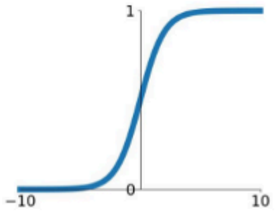
x



- 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

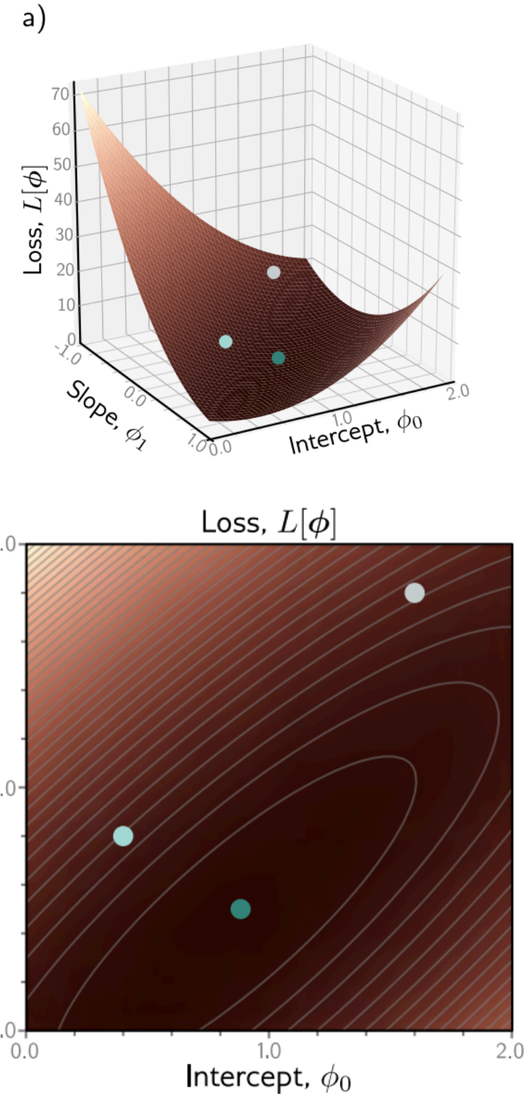
Before learning how to train a MLP we have to distinguish between the Network's Parameters and Hyperparameters

- **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 $\hat{\mathbf{y}}_i$ the ground truth. Note that in this case the output can be a vector

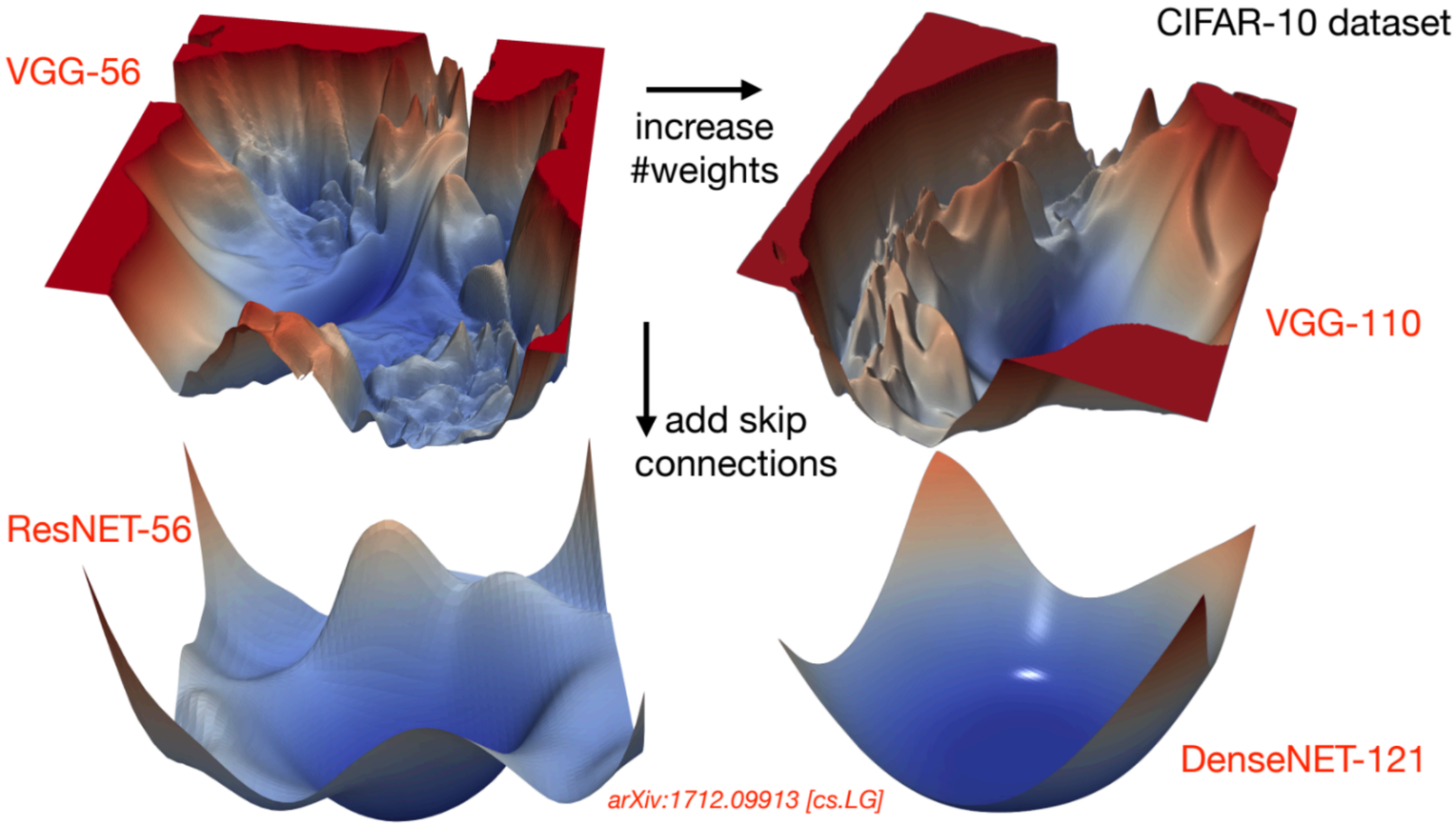
- **Binary Cross Entropy**
  - Most popular option for binary classification tasks
  - Single neuron output + sigmoid (or similar)

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

- **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

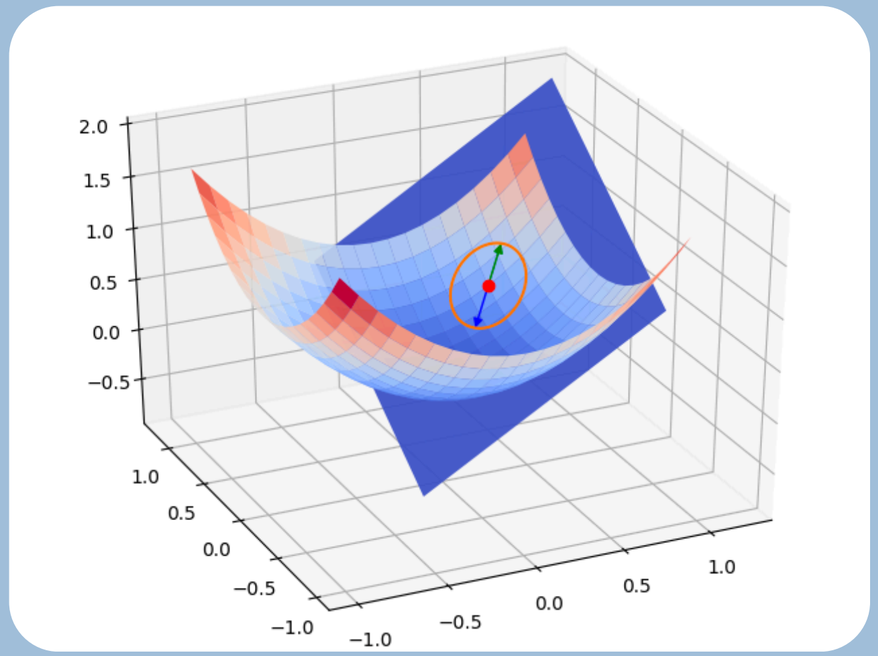
Gradient of a Function

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)

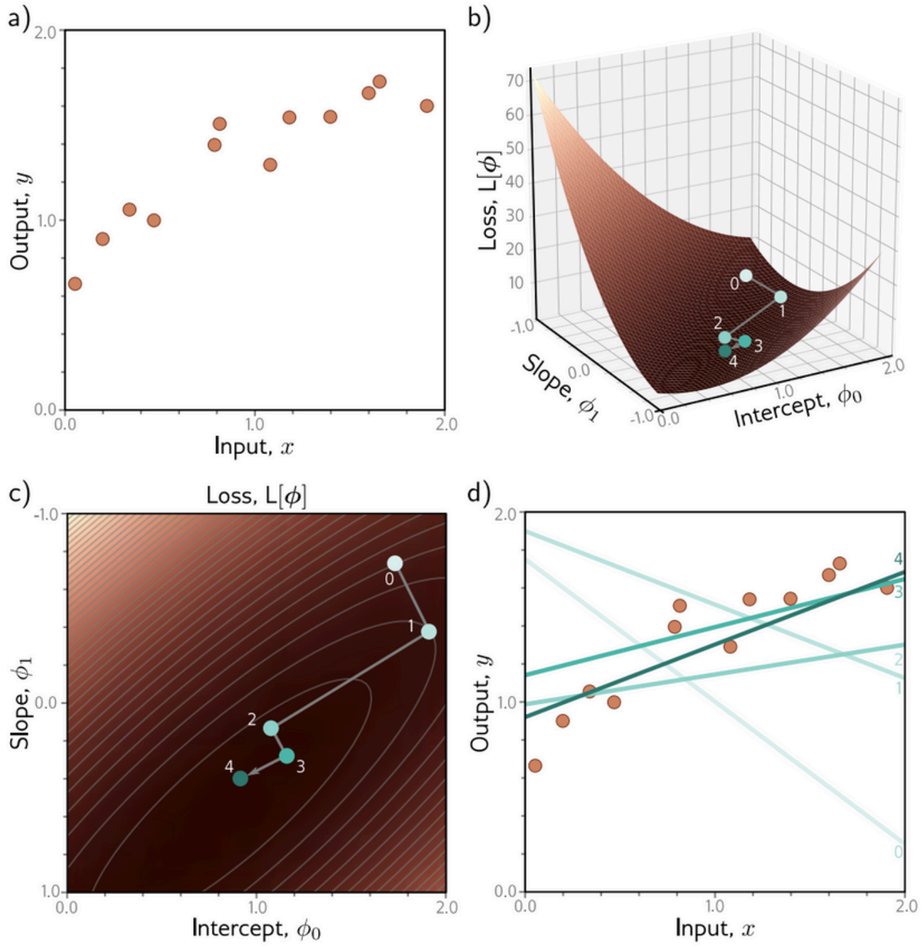
- the gradient is a generalization of the derivative for higher dimensions
- it is a vector that contains all the derivatives of the function with respect to each of its n input variables

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

- 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

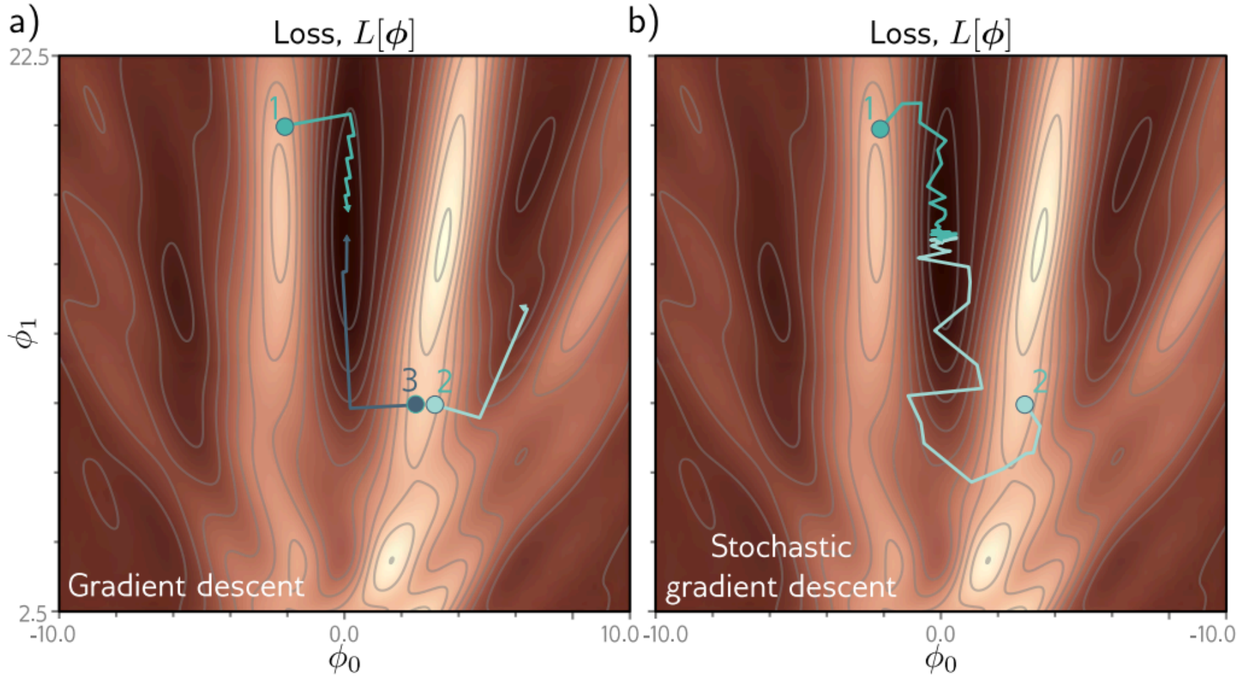
The Gradient Descent has some limits:

- 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



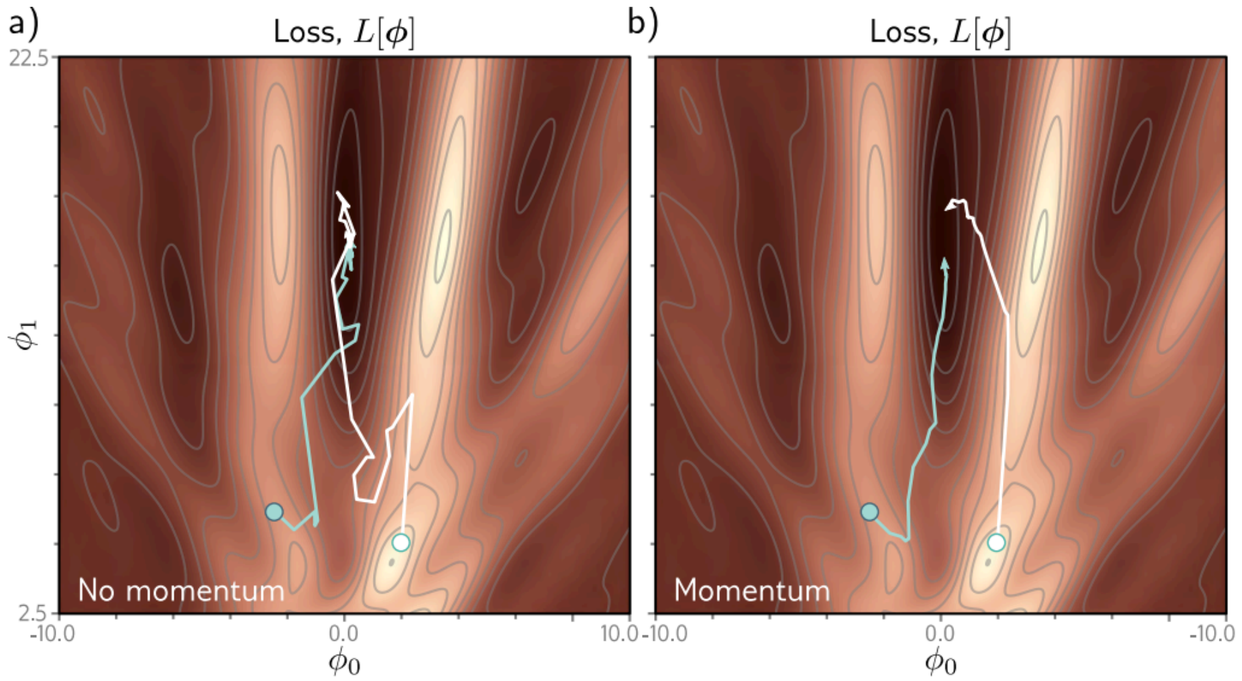
Momentum

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.

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))

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

Backpropagation Algorithm

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
- it consists of two steps:
  - a forward pass
  - a backward pass

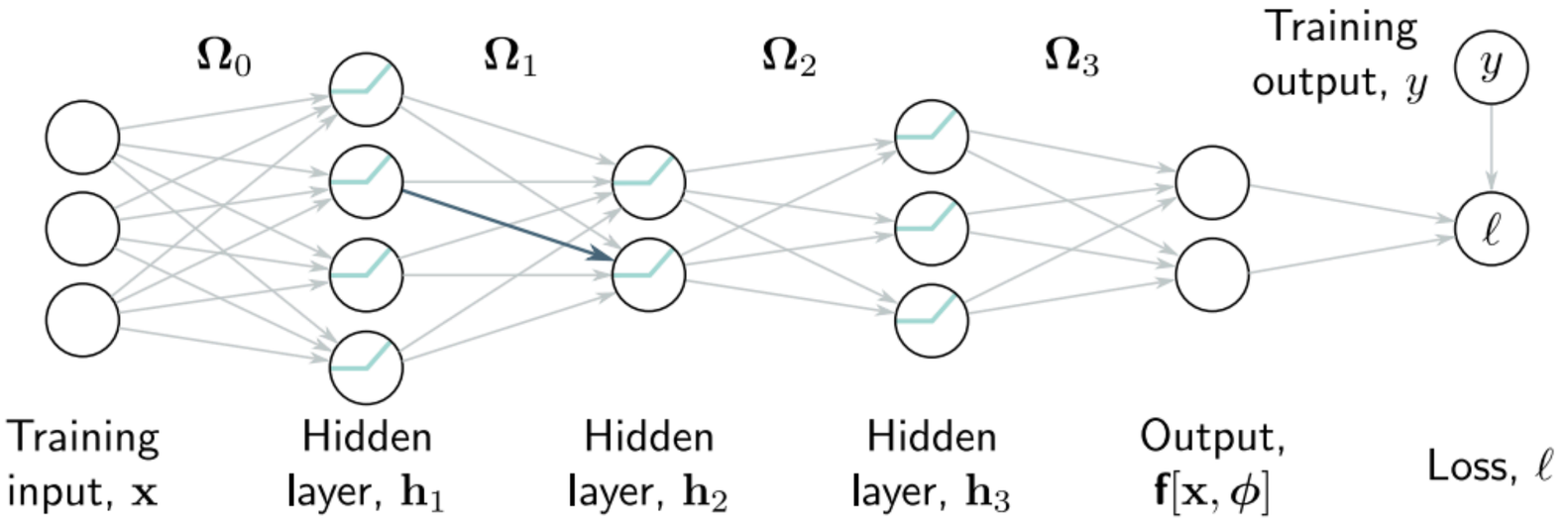
Using the Backpropagation makes computing the gradients as time consuming as computing the Loss, but a lot of memory is required (~10x model size).

Forward Pass

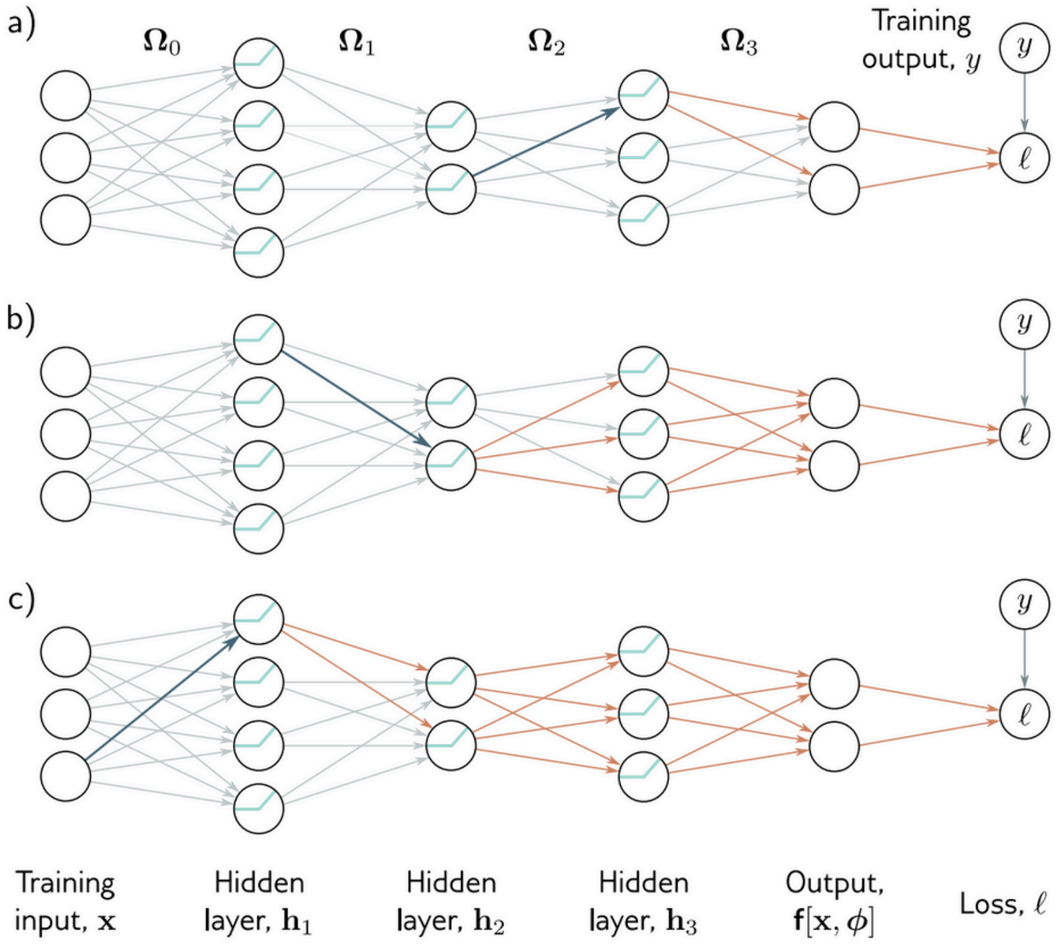
In the forward pass the DNN is feed with the training examples and it output is computed.

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!

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

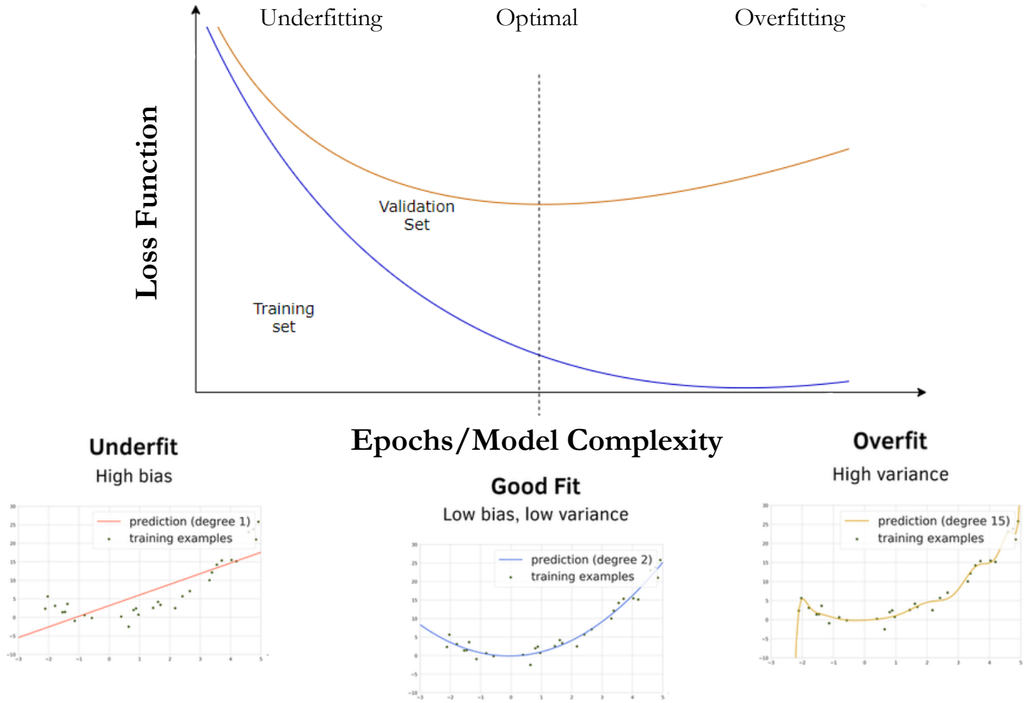
Note, however that the loss curve only tells how good is the DNN on the training data, so we could overfit the model without noticing it! For this reason we need to use also a **validation** and a **test** set.

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!

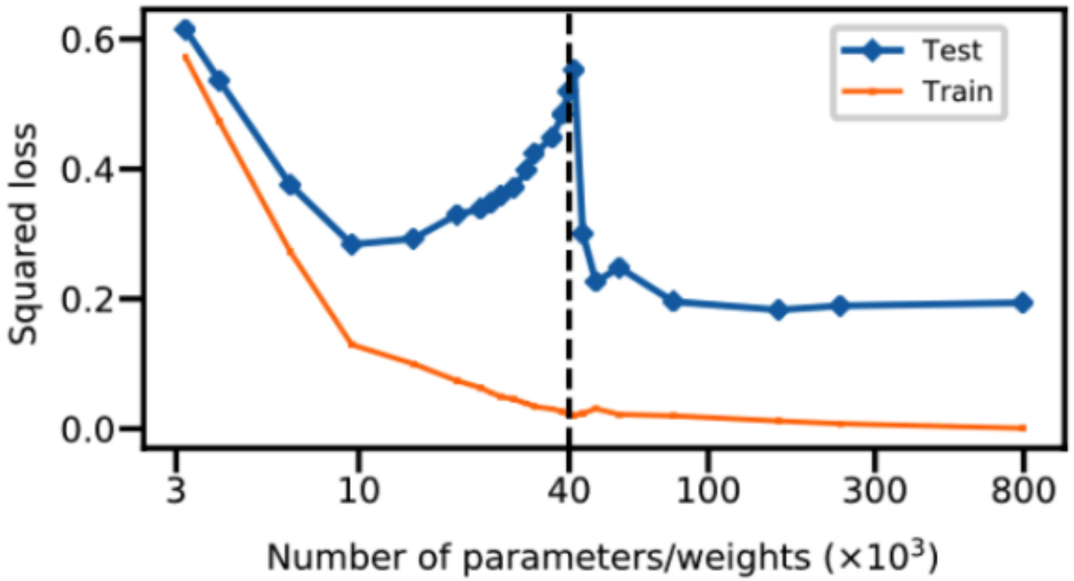


Double Descent

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.**



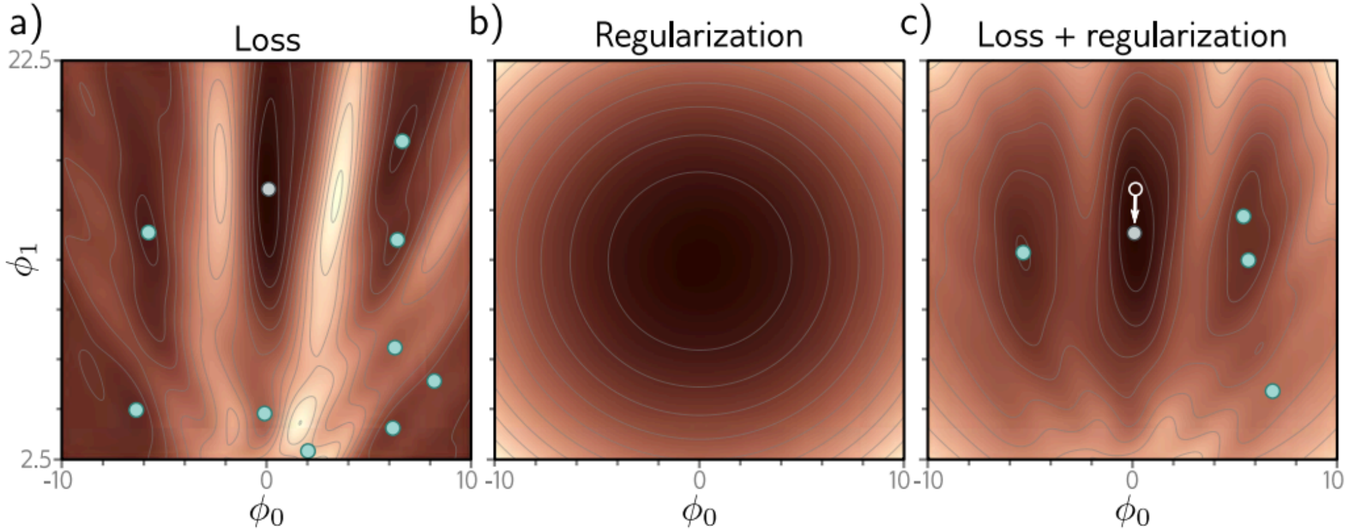
https://arxiv.org/abs/1912.02292

Universität
Konstanz

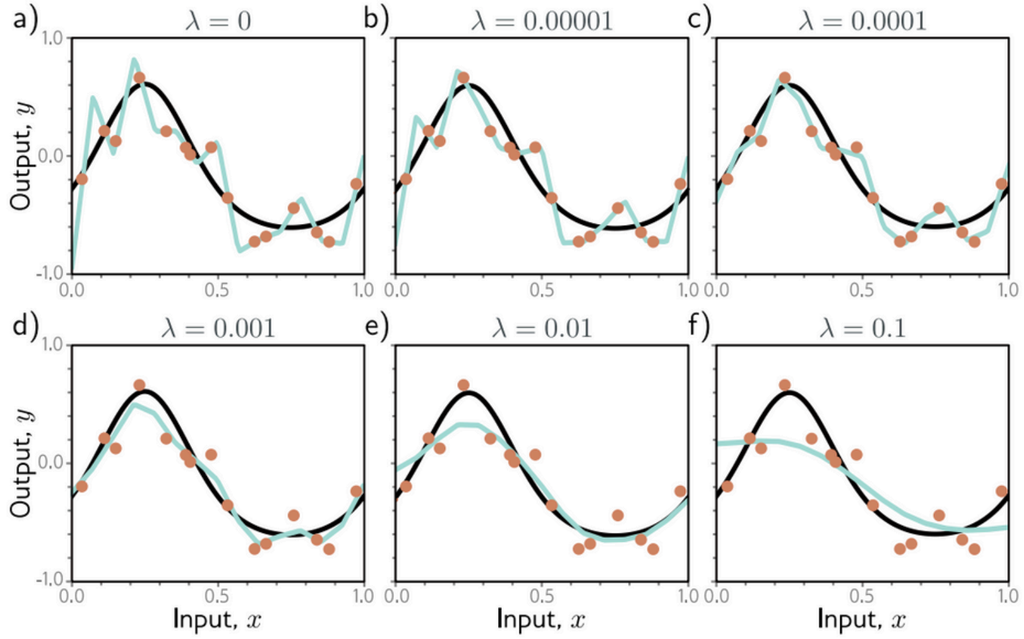
Improving Performances

Regularization Techniques

Theoretically speaking a MLP with two hidden layers and many enough neurons can perform any arbitrary complex regression or classification task. In practice finding the best parameters achieving this is very hard. This is mainly due to the fact that the loss is a very irregular function with many local minima. Regularization techniques are used to make the loss more smooth and to improve the performances of DNN.



L2 Regularization



L2 (and other norm regularizations) work by adding a penalty term to the loss that avoids weights to be too much large

$$L' = L + \lambda \sum w_i^2$$

Benefits:

- Reduces Model Complexity
- Reduces Overfitting

Implementation:

- Modify the Loss
- Train and Test the Model as Usual

Dropout

Dropout prevents overfitting by randomly dropping units (neurons) during training.

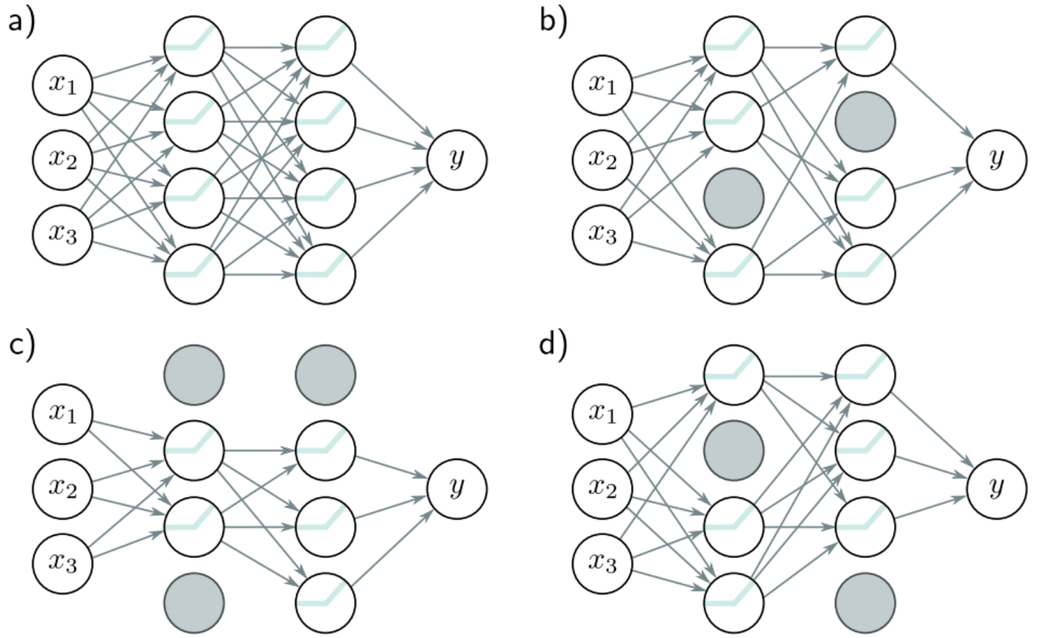
Benefits:

- Reduces Overfitting
- Increases Robustness

Implementation:

1. Apply Dropout:
2. Set Dropout Rate (0.1-0.5)
3. Training Phase Only

Conceptually, dropout effectively simulates training a large number of neural networks with different architectures in parallel.



Model Ensembling

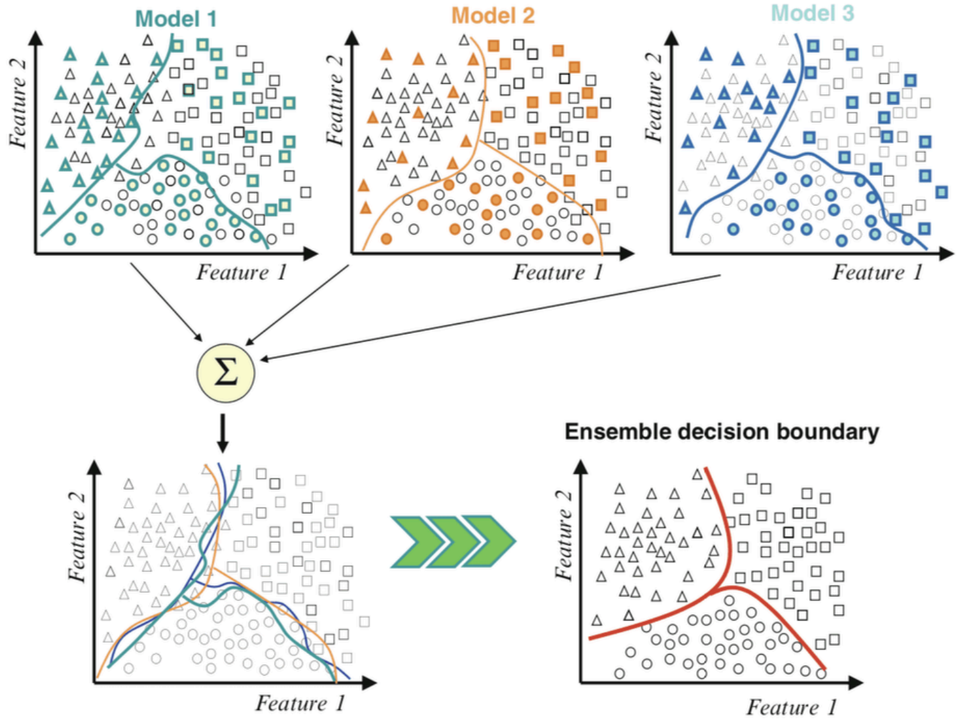
Model ensembling consists in independently training a group of different ML models on the same data. These models are then combined into a single more powerful model (random forest is one example).

Benefits:

- Reduces Overfitting
- Increases Robustness

Implementation:

1. Train Different Models
2. Combine Models
3. Test Combined Model



Early Stopping

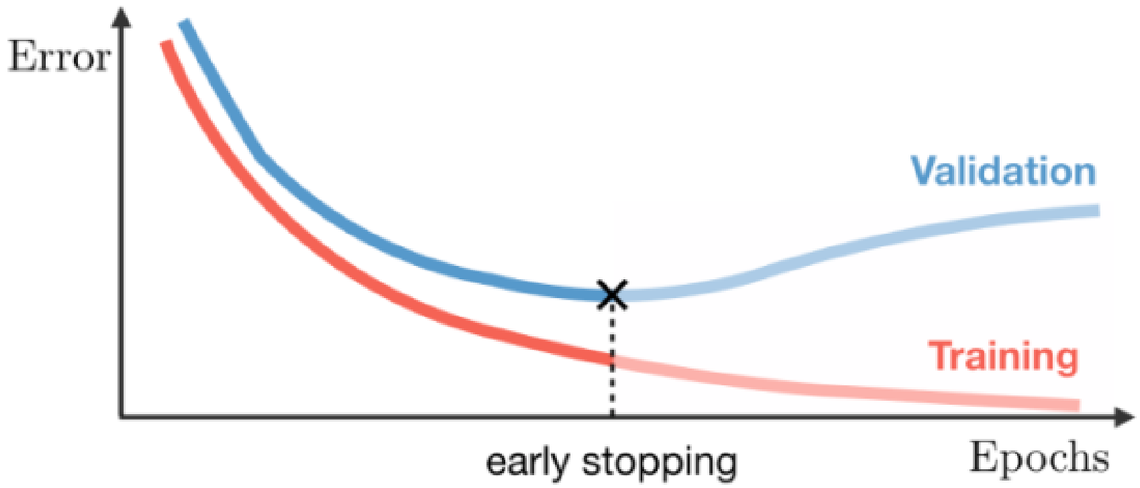
Early stopping prevents overfitting in training DNN. It works by monitoring the model's performance on a validation set during training and stopping the training process if the validation performance begins to degrade

Benefits:

- Prevents Overfitting
- Optimizes Training Time

Implementation:

1. Monitor Performance
2. Patience Counter
3. Model Checkpointing



Noise Injection

Noise addition	Information loss
	
• Addition of noise • More tolerance to quality variation of inputs	• Parts of image ignored • Mimics potential loss of parts of image

Noise injection is a regularization technique that involves adding random noise to the inputs, weights, or activations of a neural network during training. This method helps improve the robustness and generalization of the model by forcing it to learn to cope with small, random changes in the training data.

Benefits:

- Reduces Overfitting
- Enhances Robustness

Implementation:

1. Choose Target for Noise
2. Set Noise Type and Level
3. Training Phase Only

Data Augmentation

Data augmentation is a technique used to increase the diversity and amount of training data by creating modified versions of existing data or artificially generating new data. Important in unbalanced datasets. Some simple techniques are Random Oversampling and SMOTE

Benefits:

- Makes Dataset Balanced
- Enhances Robustness

Implementation:

- Augment Data
- Train on Augmented Data
- Validate and Test on non-Augmented Data

Weights Initialization

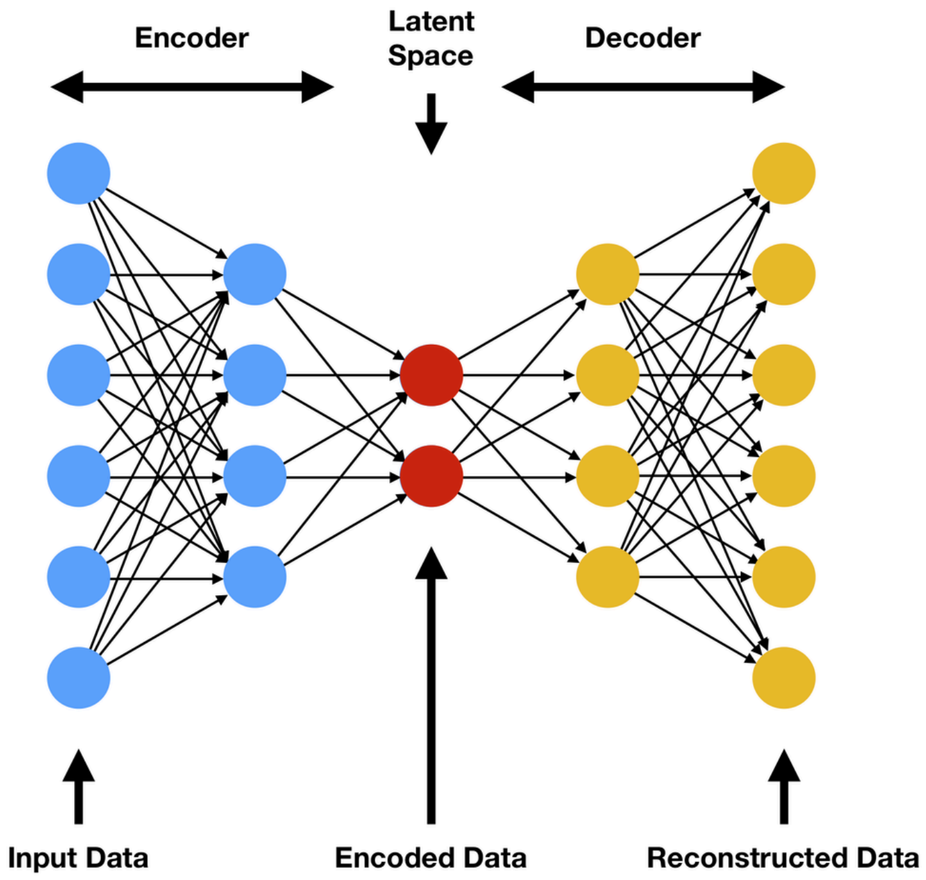
Proper weights initialization is crucial in deep neural networks as it sets the initial state from which training starts. Good initialization can help in breaking symmetry between neurons of the same layer, ensuring that each neuron learns a unique aspect of the data during training. Without this, neurons might end up learning the same features, stalling the learning process. There are several approaches:

- **Zero Initialization:** Generally avoided as it maintains symmetry, causing neurons to learn identical features.
- **Random Initialization:** Assigns weights with small random numbers. Common methods include using a Gaussian or uniform distribution, often scaled by the size of the previous layer (e.g., Xavier/Glorot or He initialization).
    - **Xavier/Glorot Initialization:** Ideal for layers with Sigmoid or Tanh activations.
    - **He Initialization:** Designed for ReLU activations, considering the variance of neurons in the network to maintain activation scales.

Autoencoder Architecture

Universität
Konstanz

The Autoencoder



The Autoencoder is one of the most important MLP architectures for unsupervised learning. It is composed of three sections:

- **Encoder** Encodes the data into a latent representation
- **Latent Space** Space where the encoded data live
- **Decoder** Convert back the data from the latent space to the standard representation

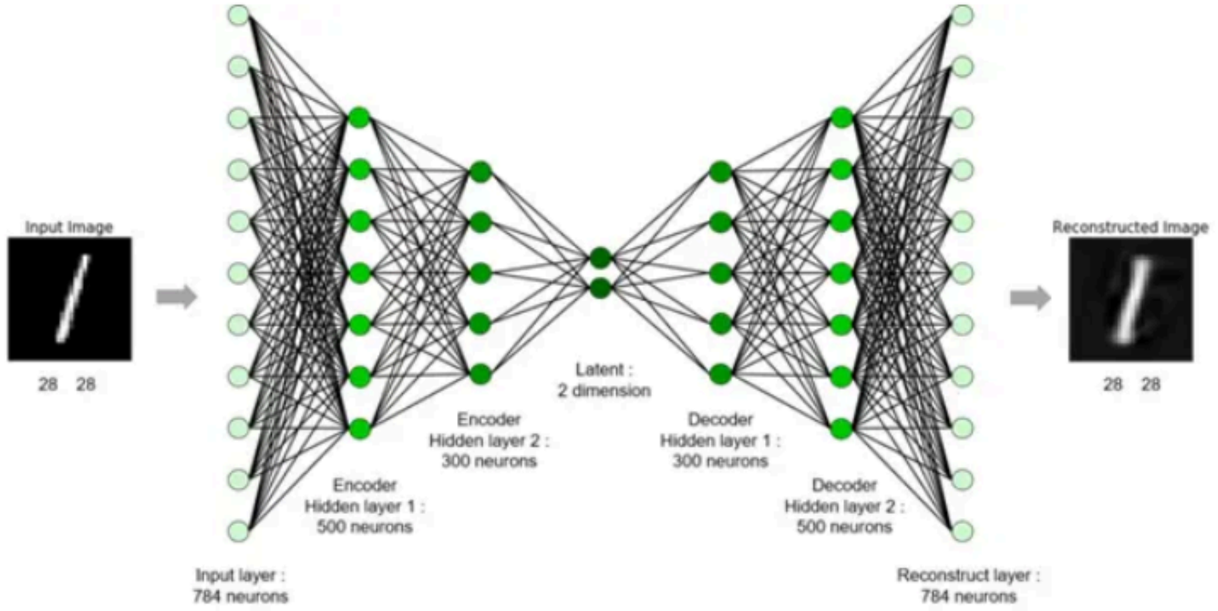
The Autoencoder can be used for several tasks

- Dimensionality Reduction
- Anomaly Detection
- Denoising

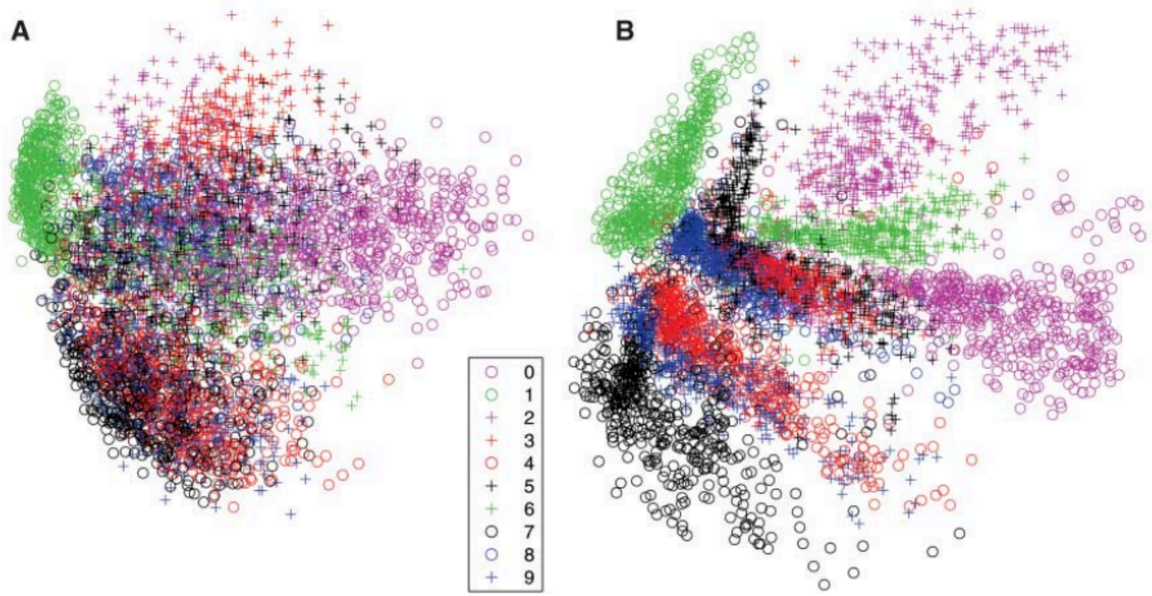
Unsupervised Learning

The Autoencoder is trained in an unsupervised way using a reconstruction loss

- it is trained to reconstruct in output exactly what it is given in input
- the loss quantifies how different is the output from the input
- practically speaking it is like a supervised regression, but there is no need of labelled data



Dimensionality Reduction



The Autoencoder architecture has a bottleneck, the latent space:

- if the reconstruction loss is low, the autoencoder can successfully reconstruct the input
- this means that the latent space contains enough information to reconstruct the input
- the latent space contains a low dimensional representation of the input data

Therefore we can train an autoencoder and then use it encoder to get a dimensionality reduction of the data (similar to PCA or T-SNE)

Anomaly Detection

Autoencoders can be used to perform anomaly detection

- the autoencoder will correctly reconstruct most of the data with low error
- outliers and anomalies, that are rarely (or never) seen in the training data will be harder to reconstruct

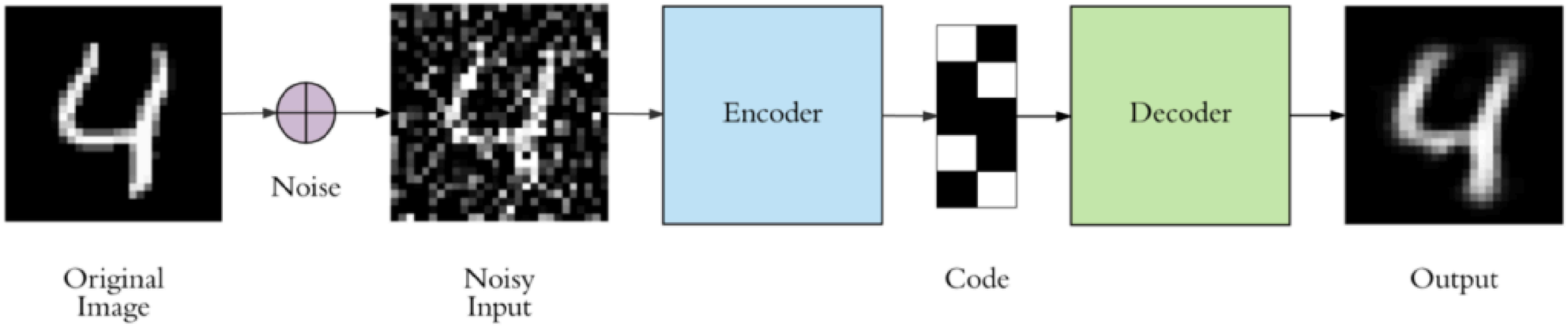
The idea is thus to determine if an event is or not an anomaly (e.g. credit card fraud) based on its reconstruction error. Note that no labels are needed (unsupervised anomaly detection).

Denoising

The Autoencoder can be used to remove noise from data

- training data are corrupted with some noise
- the Autoencoder is feed with the noisy data, but it is trained to output the original data without noise
- since noise is random and contains no information, the latent space does not learn it

As a consequence, when given in input noisy data or images, a denoising autoencoder can remove all noise returning a clean image. Again no supervised learning is required.



Summary

Multilayer Perceptron Architecture

- Deep Neural Network with layers of neurons connected by non-linear activation functions. Equivalent to a combination of matrix multiplications and activations

Training Deep Neural Networks

- Loss function to measure performances of the DNN
- Optimization algorithm to find the best parameters
- Backpropagation to compute the gradient of the loss

Improving Performances

- Several possible approaches to regularize the loss, they can be used alone or together

Autoencoder Architecture

- Unsupervised learning model trained to reconstruct the input
- Bottleneck structure to extract low-dimension latent representation of the input