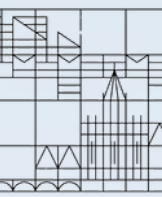


04 | Training Neural Networks

Giordano De Marzo

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

Deep Learning for the Social Sciences

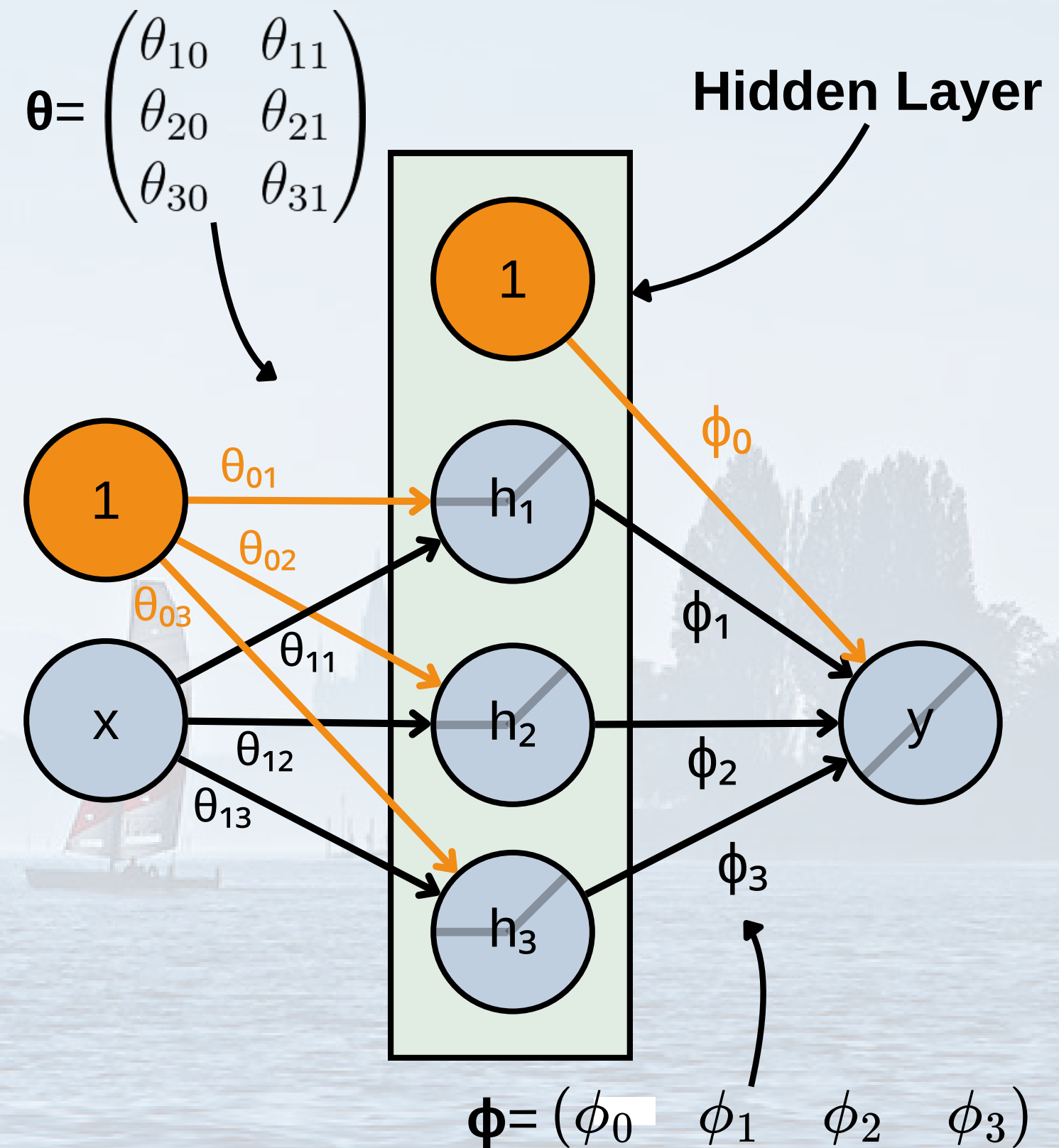


Combining Perceptrons

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

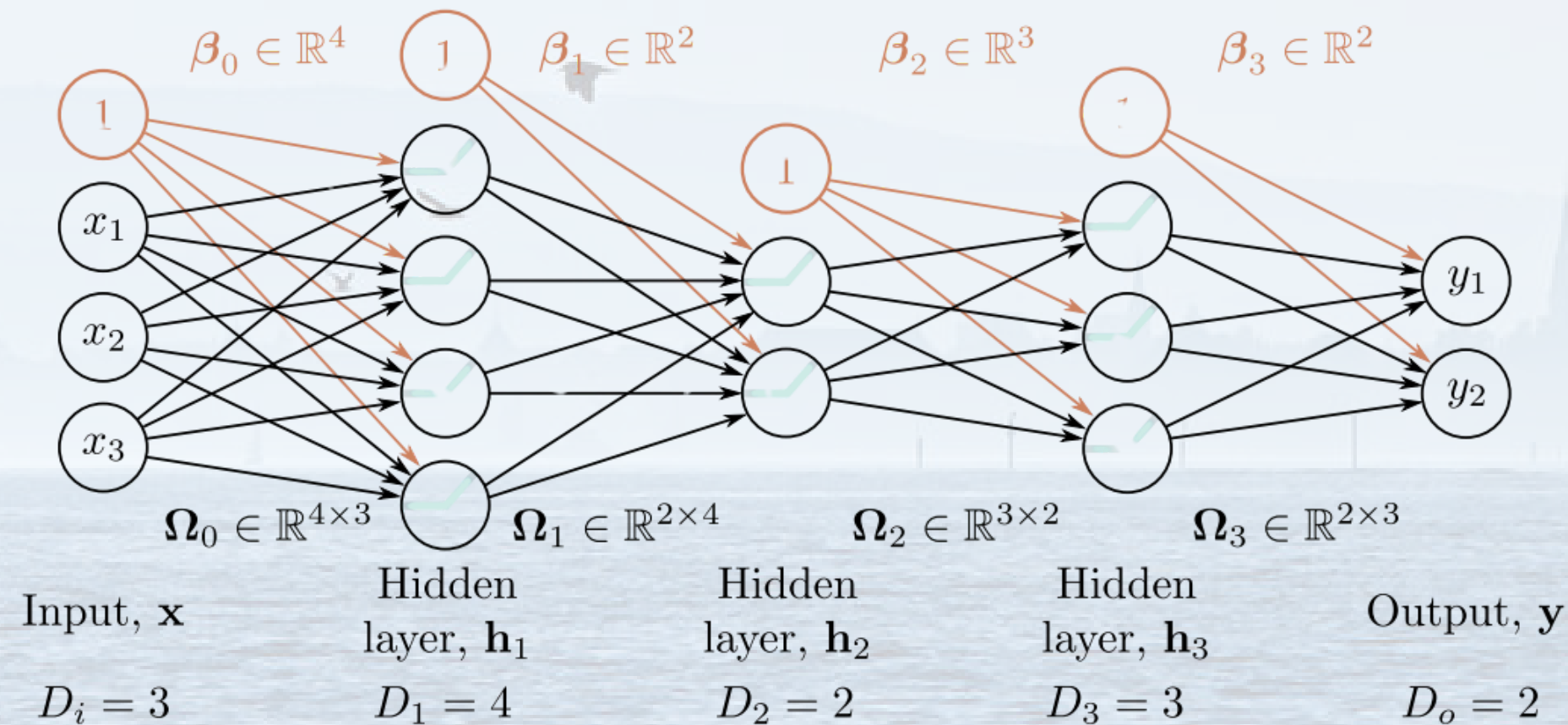
In this way we are adding an **hidden layer** to the neural network



Recap: Multilayer Perceptron

A general Multilayer Perceptron consists of K hidden layers $\mathbf{h}_1 \dots \mathbf{h}_K$, for a total of $K+1$ weight matrices $\mathbf{\Omega}_0 \dots \mathbf{\Omega}_K$ and bias vectors $\beta_0 \dots \beta_K$.

Matrix notation is useful to represent large neural networks in a compact notation.



$$\begin{aligned}
 \mathbf{h}_1 &= \mathbf{a}[\beta_0 + \mathbf{\Omega}_0 \mathbf{x}] \\
 \mathbf{h}_2 &= \mathbf{a}[\beta_1 + \mathbf{\Omega}_1 \mathbf{h}_1] \\
 \mathbf{h}_3 &= \mathbf{a}[\beta_2 + \mathbf{\Omega}_2 \mathbf{h}_2] \\
 &\vdots \\
 \mathbf{h}_K &= \mathbf{a}[\beta_{K-1} + \mathbf{\Omega}_{K-1} \mathbf{h}_{K-1}] \\
 \mathbf{y} &= \beta_K + \mathbf{\Omega}_K \mathbf{h}_K.
 \end{aligned}$$



Let's Make it Easy

This may seem 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?

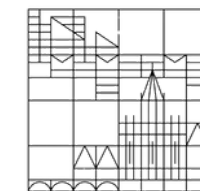
- possible in theory
- impossible in practice
- **DNN** are just a very **computationally efficient** way to represent and fit arbitrary complex functions

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).

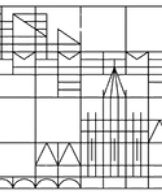


Outline

1. Training Deep Neural Networks
2. Backpropagation Algorithm
3. Regularizing Neural Networks



Training Deep Neural Networks

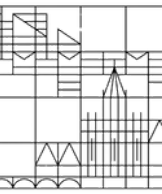


Training a Multilayer Perceptron

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:

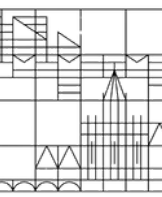
- a **Loss Function** that quantifies how good is the MLP in performing the desired task (how close is its output to the desired output)
- an **Optimization Algorithm** that determines how the parameters of the MLP should be updated to make the Loss decrease
- the **Backpropagation** algorithm, used 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 $\Omega_0 \dots \Omega_k$ and biases $\beta_0 \dots \beta_k$ 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



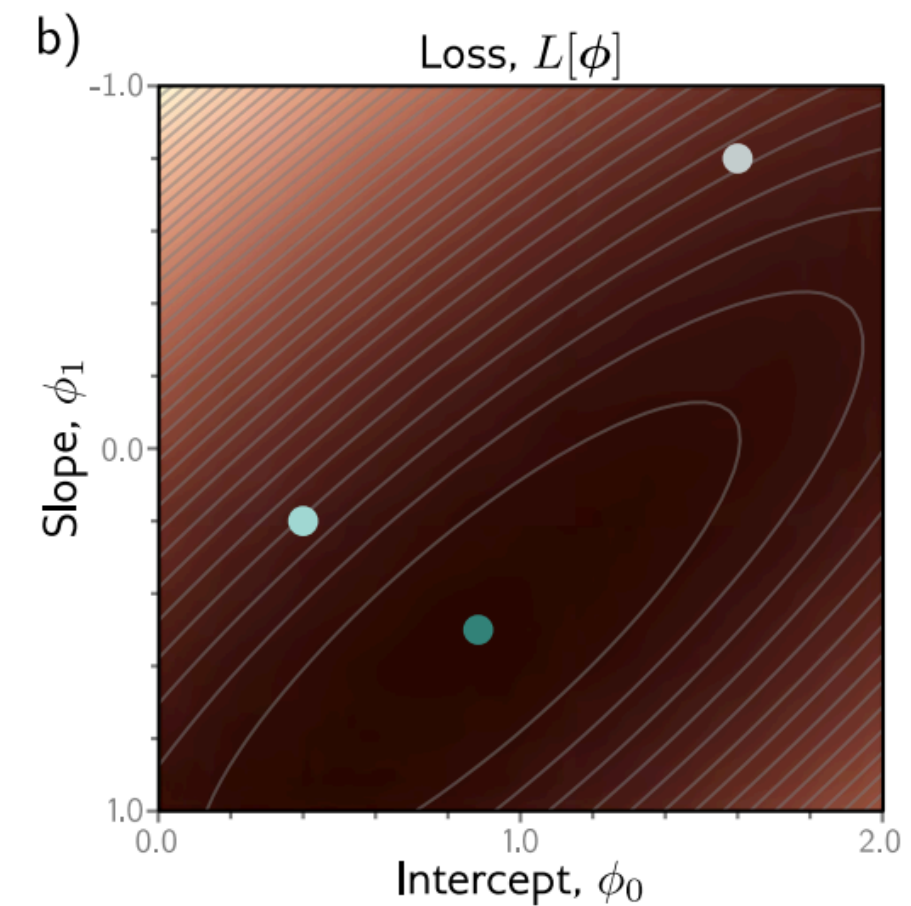
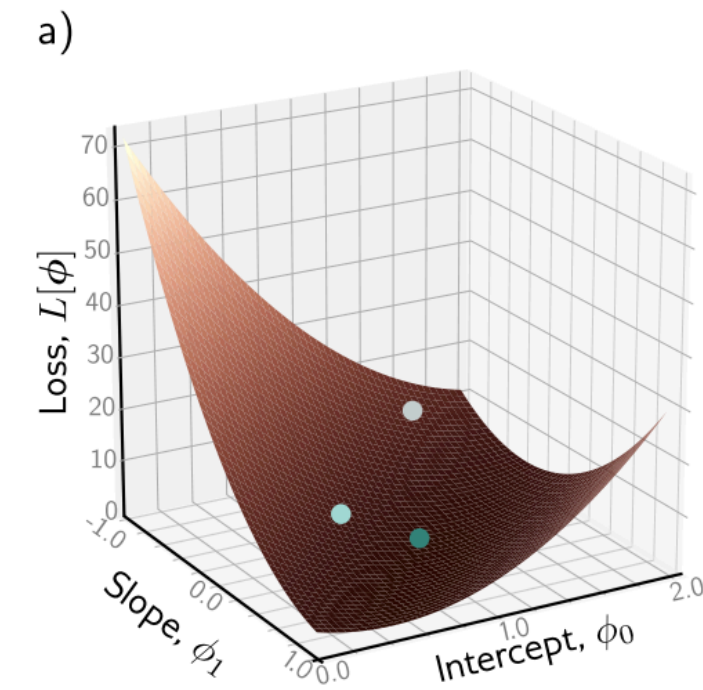
Loss Function

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

$$L = L[\Omega_0 \dots \Omega_k, \beta_0 \dots \beta_k]$$

- 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 $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots)$ and is tasked with predicting a continuous value. We denote by $y_i = y_i(\mathbf{x}_i | \boldsymbol{\Omega}_0 \dots \boldsymbol{\Omega}_k, \beta_0 \dots \beta_k)$ 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

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

- **Mean Absolute Error**

- Good alternative
- Less importance to outliers

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



Loss Functions for Classification

In a classification task the neural network receives an input vector $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots)$ and is tasked with predicting to which class the input belongs to. We denote by $\mathbf{y}_i = \mathbf{y}_i(\mathbf{x}_i | \boldsymbol{\Omega}_0 \dots \boldsymbol{\Omega}_k, \beta_0 \dots \beta_k)$ 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)

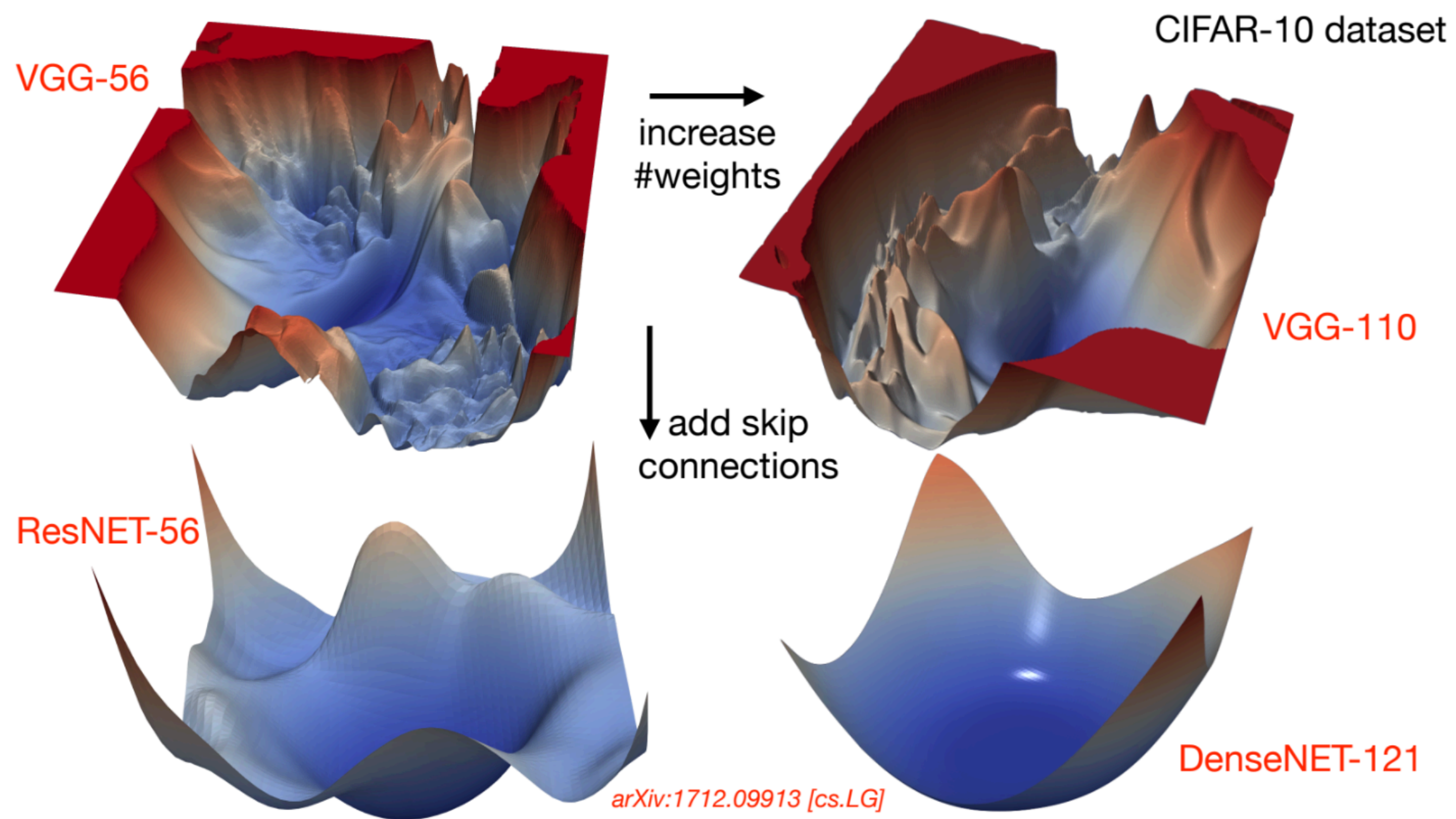
$$L[\boldsymbol{\Omega}_0 \dots \boldsymbol{\Omega}_k, \beta_0 \dots \beta_k] = -\frac{1}{N} \sum_i^N [\hat{y}_i \log(y_i) + (1 - \hat{y}_i) \log(1 - y_i)]$$

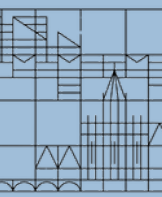
- **Categorical Cross Entropy**

- Most popular option for multi-class classification tasks
- Multiple neurons output + softmax

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

Why going Deep?





Math Recap

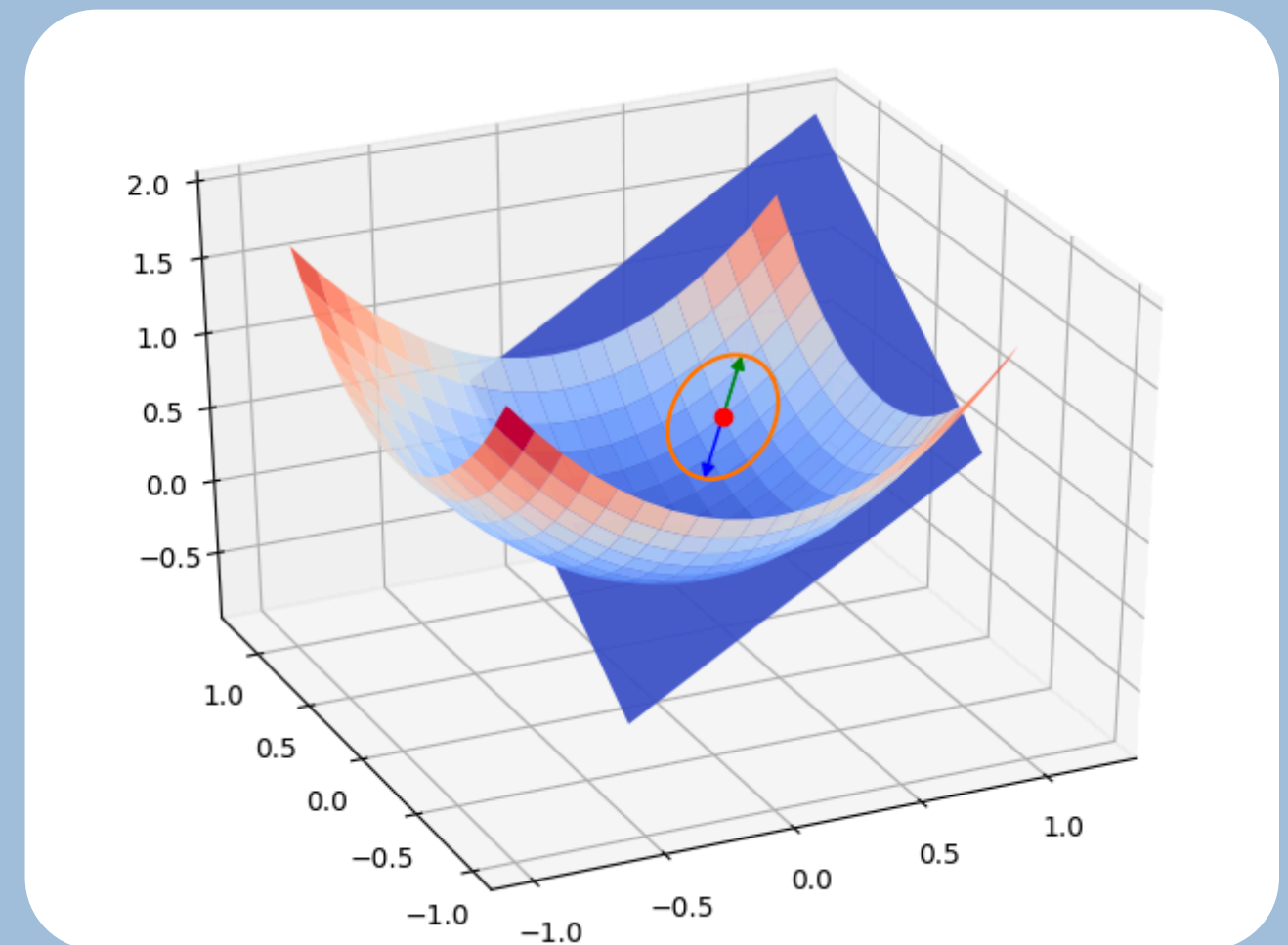
Gradient of a Function

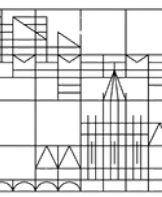
Let us consider a function $f(x_1, x_2, \dots, x_n)$ from \mathbb{R}^n (input in n dimensions) to \mathbb{R} (output is a real number)

- 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

$$\nabla f = \left(\frac{df}{dx_1}, \frac{df}{dx_2}, \dots, \frac{df}{dx_n} \right)$$

- 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

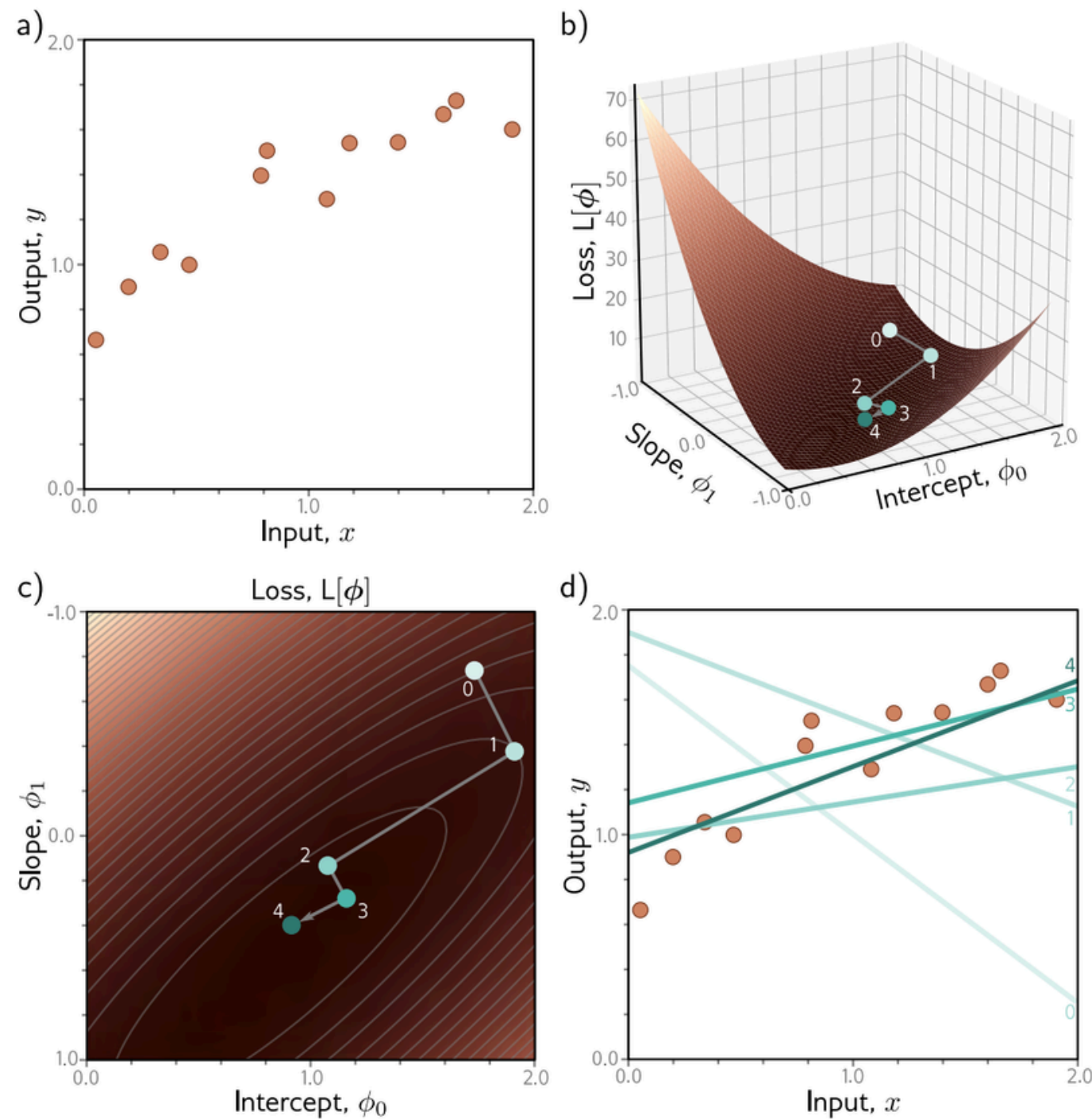
- we denote by \mathbf{W}_t the set of all weights and biases at iteration t and we compute the gradient of the Loss in \mathbf{W}_t

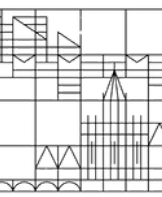
$$\nabla L(\mathbf{W}_t) = \left(\frac{dL}{d\Omega_{0,0}}, \frac{dL}{d\Omega_{0,1}}, \dots, \frac{dL}{d\beta_{0,0}}, \frac{dL}{d\beta_{0,1}}, \dots \right)_{\mathbf{W}_t}$$

- 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

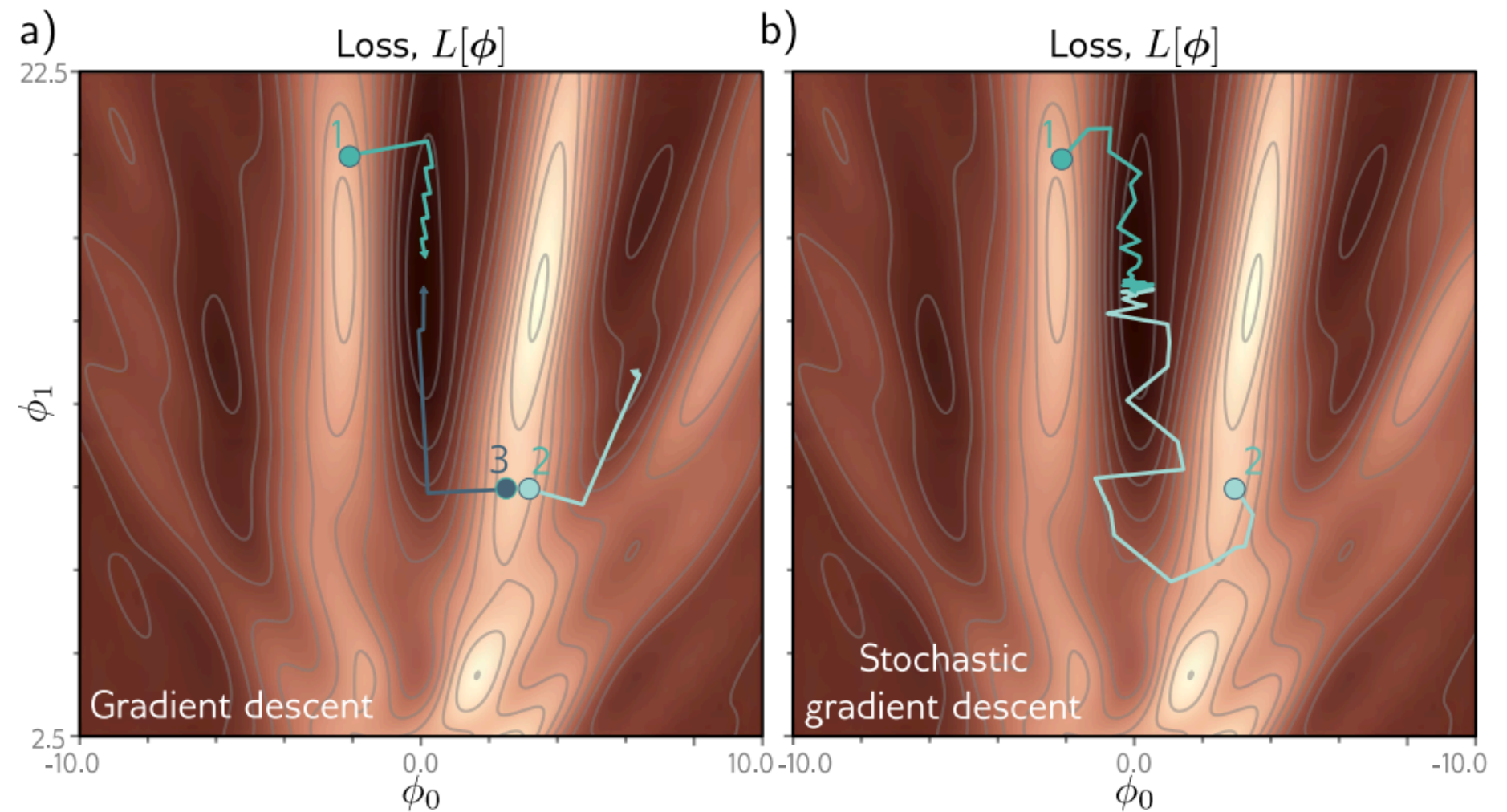
The Gradient Descent has some limits:

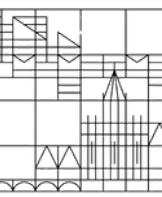
- it becomes computationally inefficient for large datasets
- it may get stuck in local minima

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





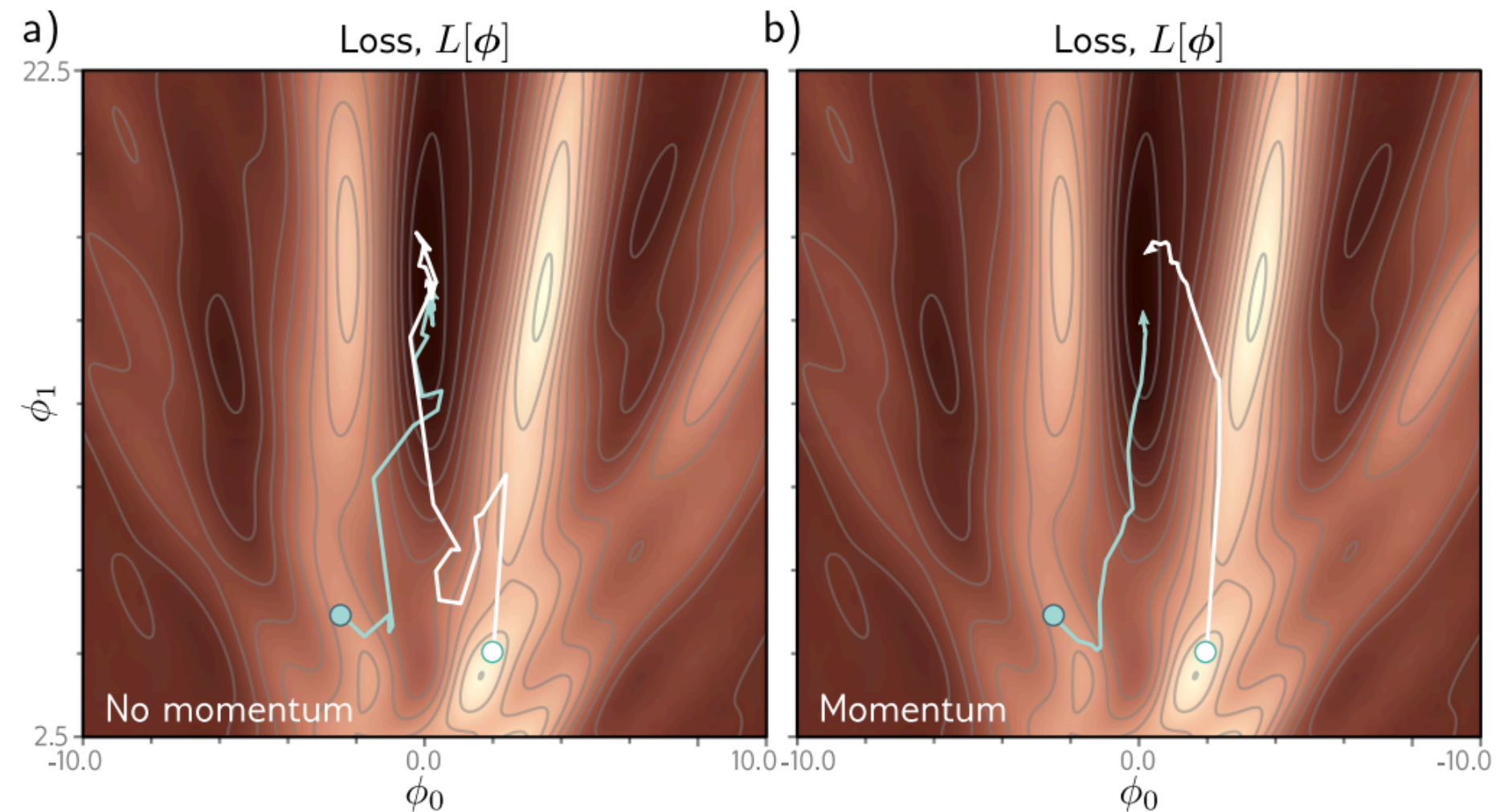
Momentum

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 $\alpha \sim 0.9-0.99$. This makes the convergence faster and more regular.



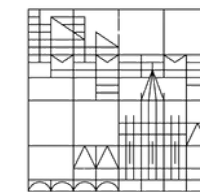


Other Optimization Algorithms

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.



Backpropagation Algorithm



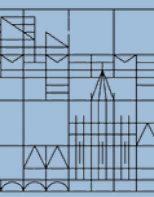
Deriving the Loss

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 ($\sim 10x$ model size).



Math Recap

Chain Rule for Derivatives

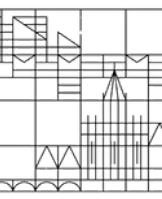
The chain rule is a simple mathematical rule for computing the function of composite functions. Let us consider two functions $f(x)$ and $g(x)$, we want to compose these functions to get $F=f(g(x))$ and then compute the derivative of this composed function with respect to x .

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 $F(x)=\sin(\log(x))$

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

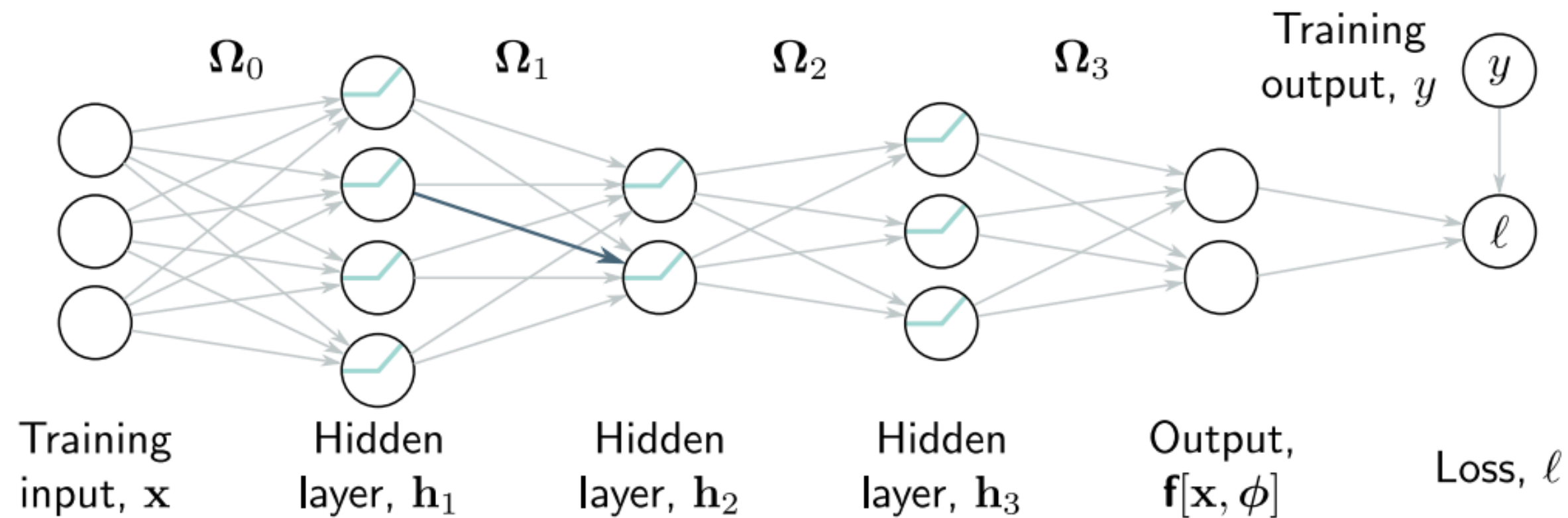


Forward Pass

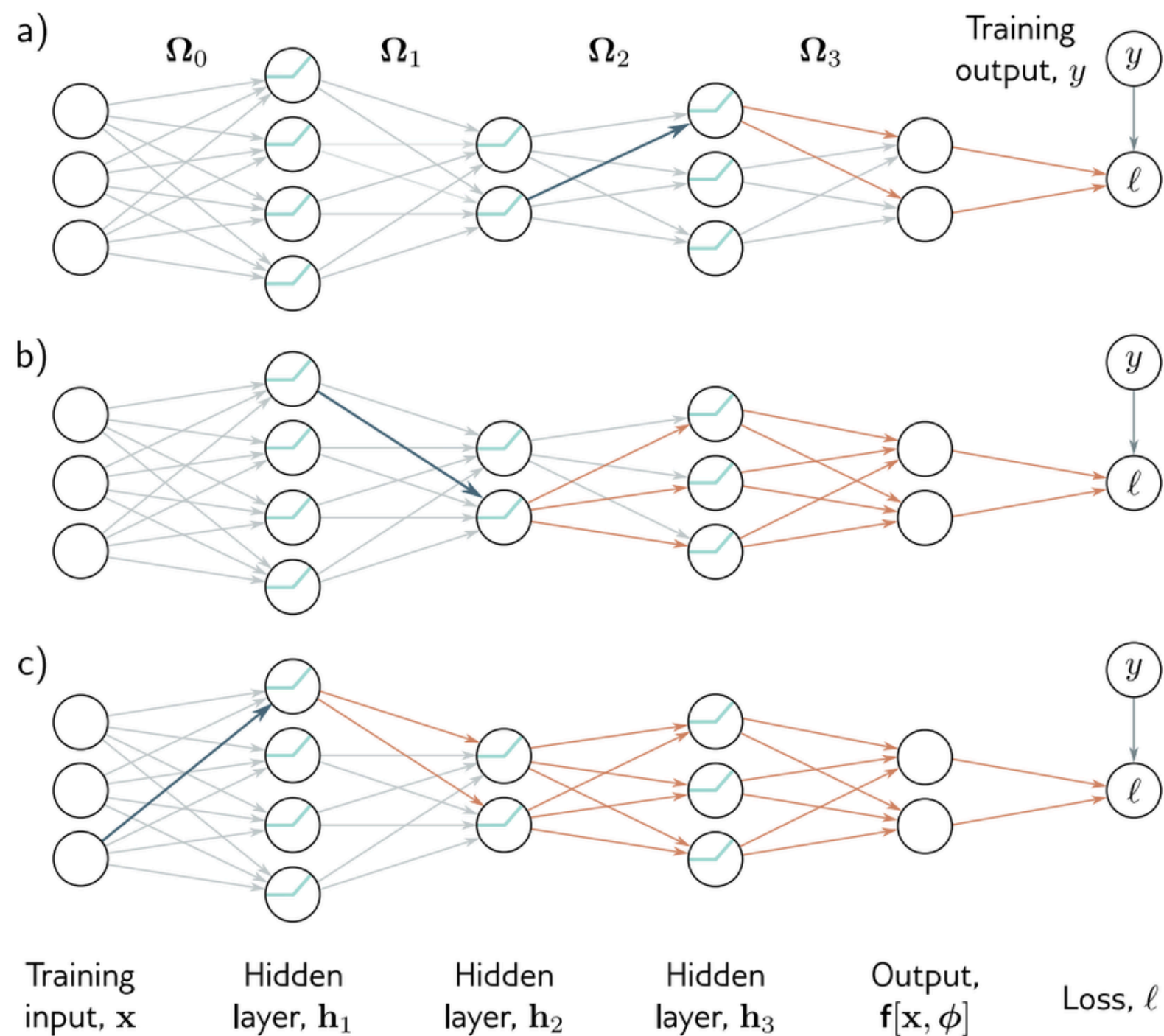
In the forward pass the DNN is feed with the training examples and its output is computed.

Moreover, for each neuron, the following quantities are stored

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



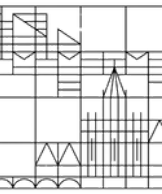
Backward Pass



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

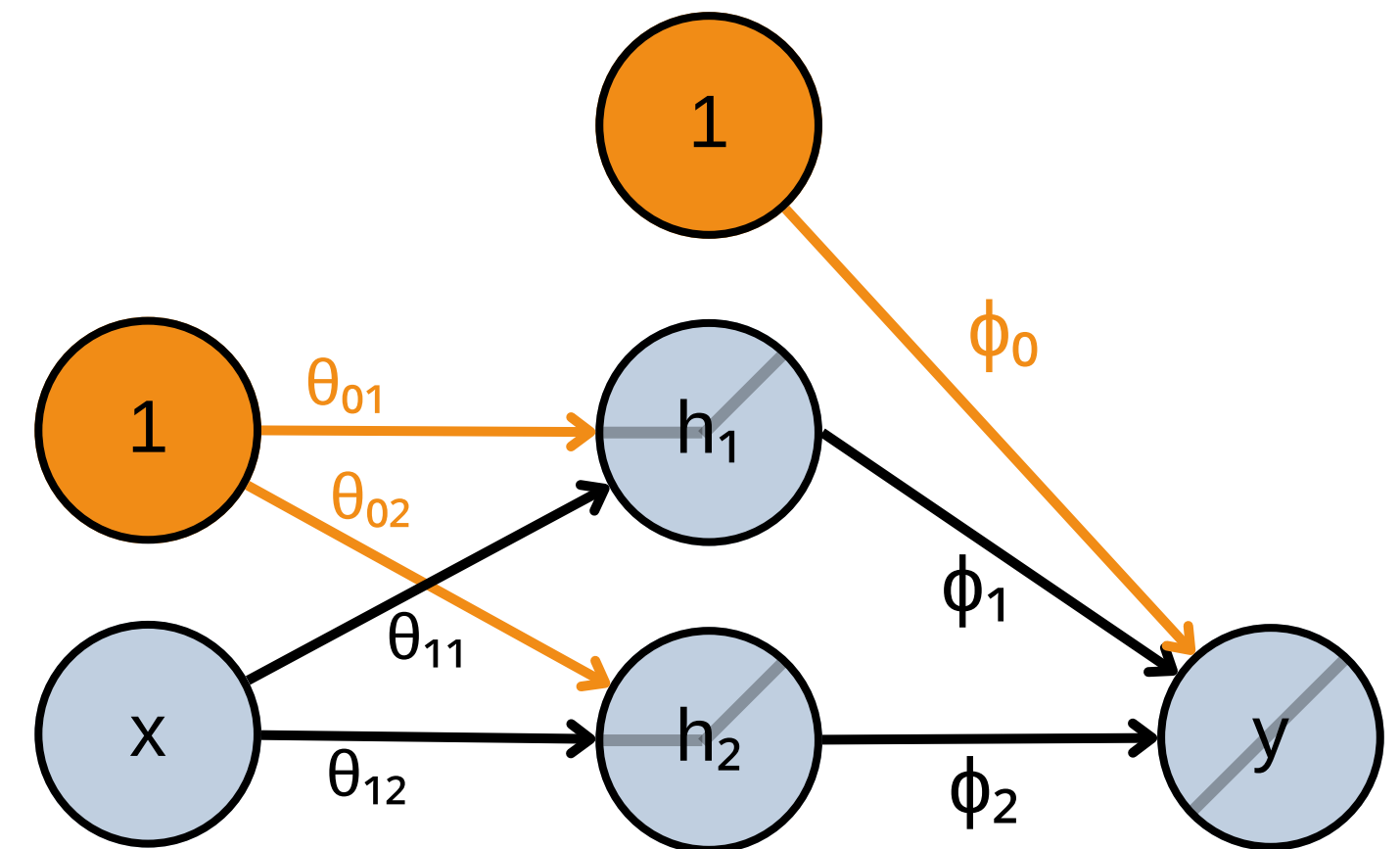
First we perform the forward pass and we compute the following quantities

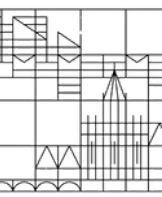
- preactivations f
- hidden units h
- outputs y
- Loss L

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

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

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





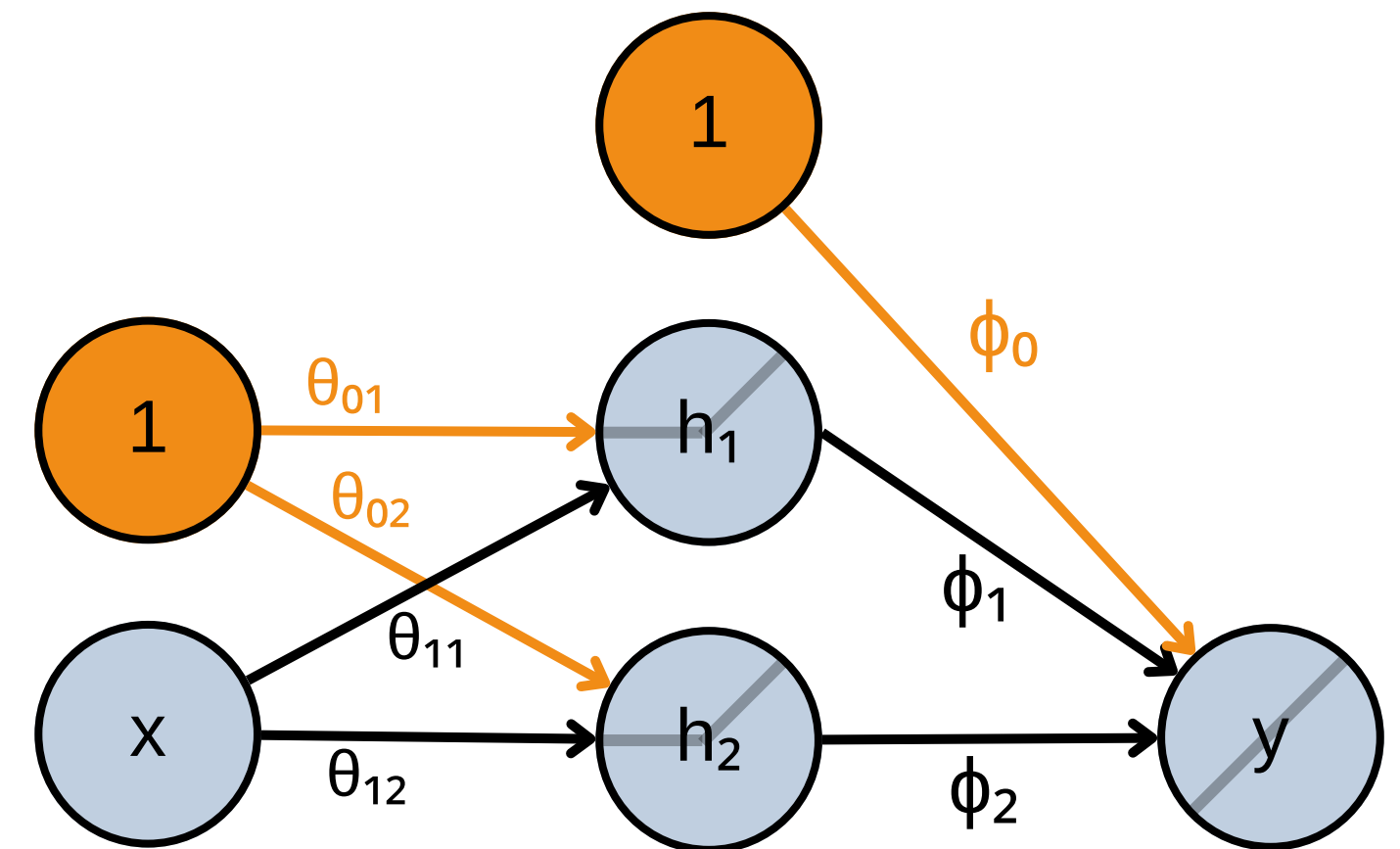
Example: Forward Pass

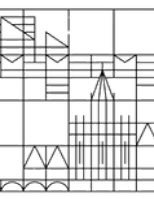
First we perform the forward pass and we compute the following quantities

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

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

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





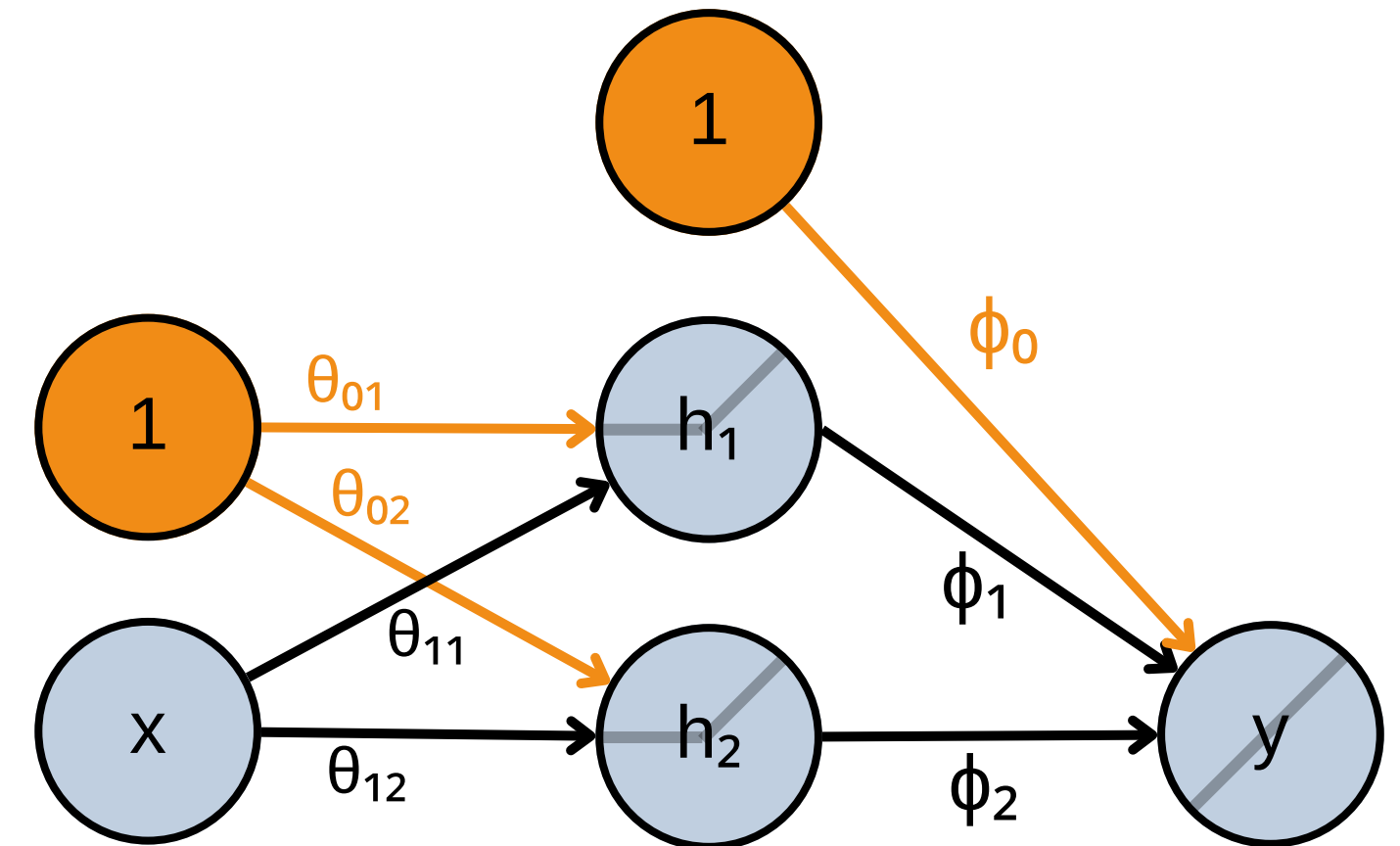
Example: Forward Pass

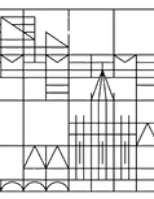
First we perform the forward pass and we compute the following quantities

- 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

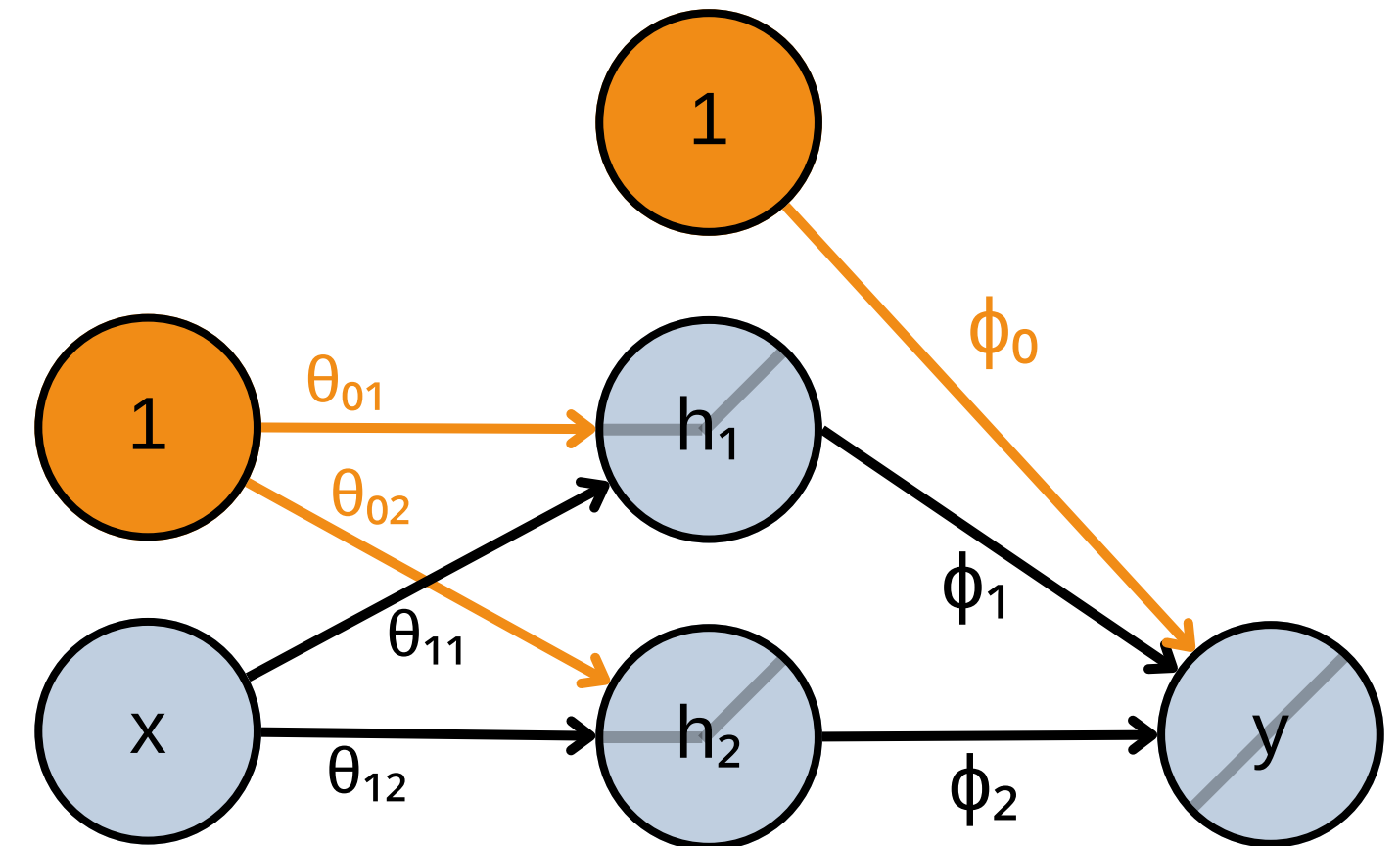
$$\frac{d\mathcal{L}}{d\phi_1} = \frac{d(y - \bar{y})^2}{d\phi_1} = 2(y - \bar{y}) \cdot \frac{dy}{d\phi_1}$$

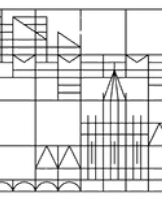
We already computed the expression for y

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

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 to derive with respect to the weights in the input layer

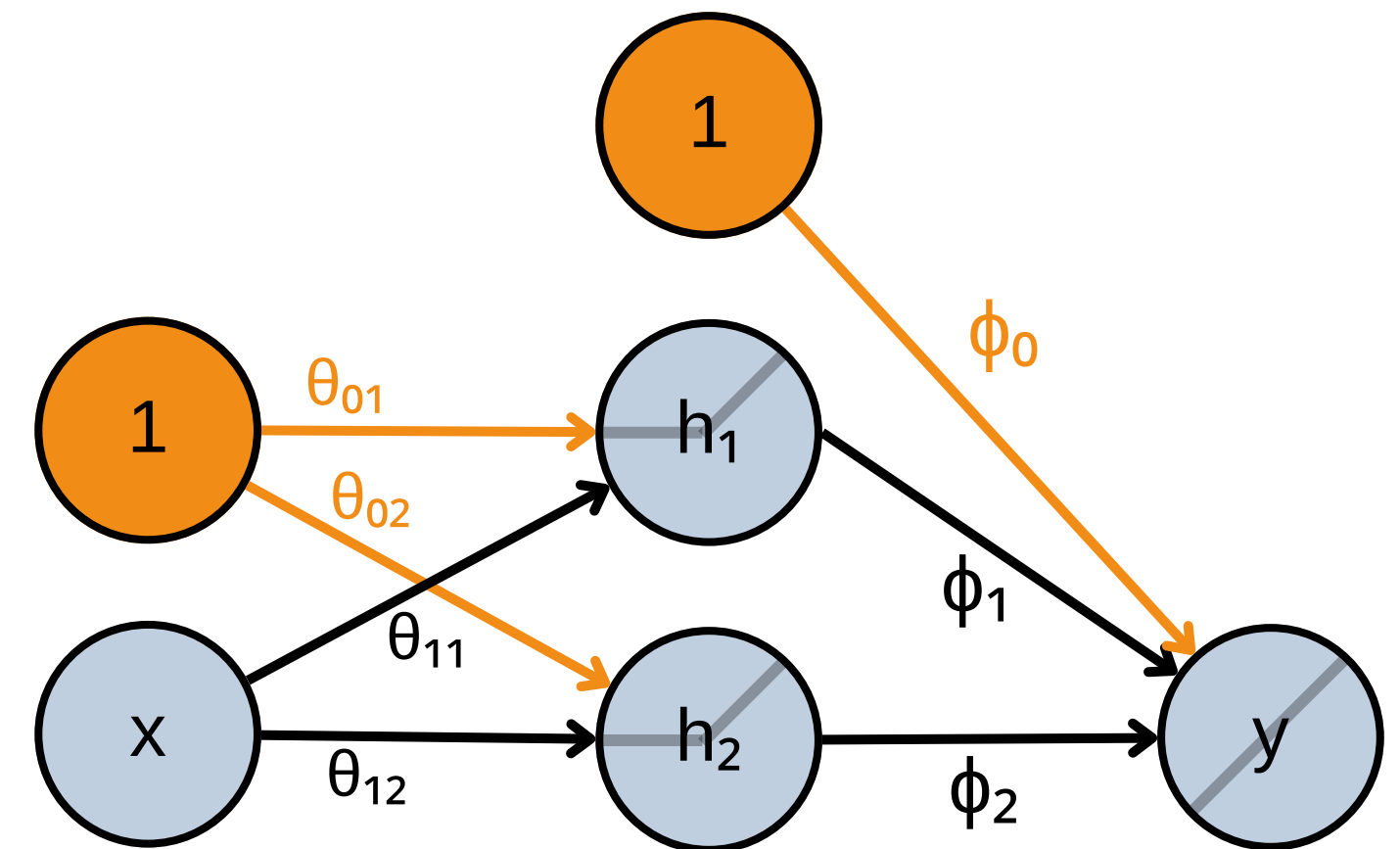
$$\frac{d\mathcal{L}}{d\theta_{11}} = \frac{d\mathcal{L}}{dy} \cdot \frac{dy}{dh_1} \cdot \frac{dh_1}{df_1} \cdot \frac{df_1}{d\theta_{11}}$$

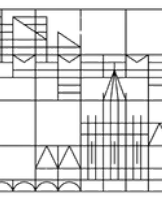
We need to compute these 4 derivatives

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

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

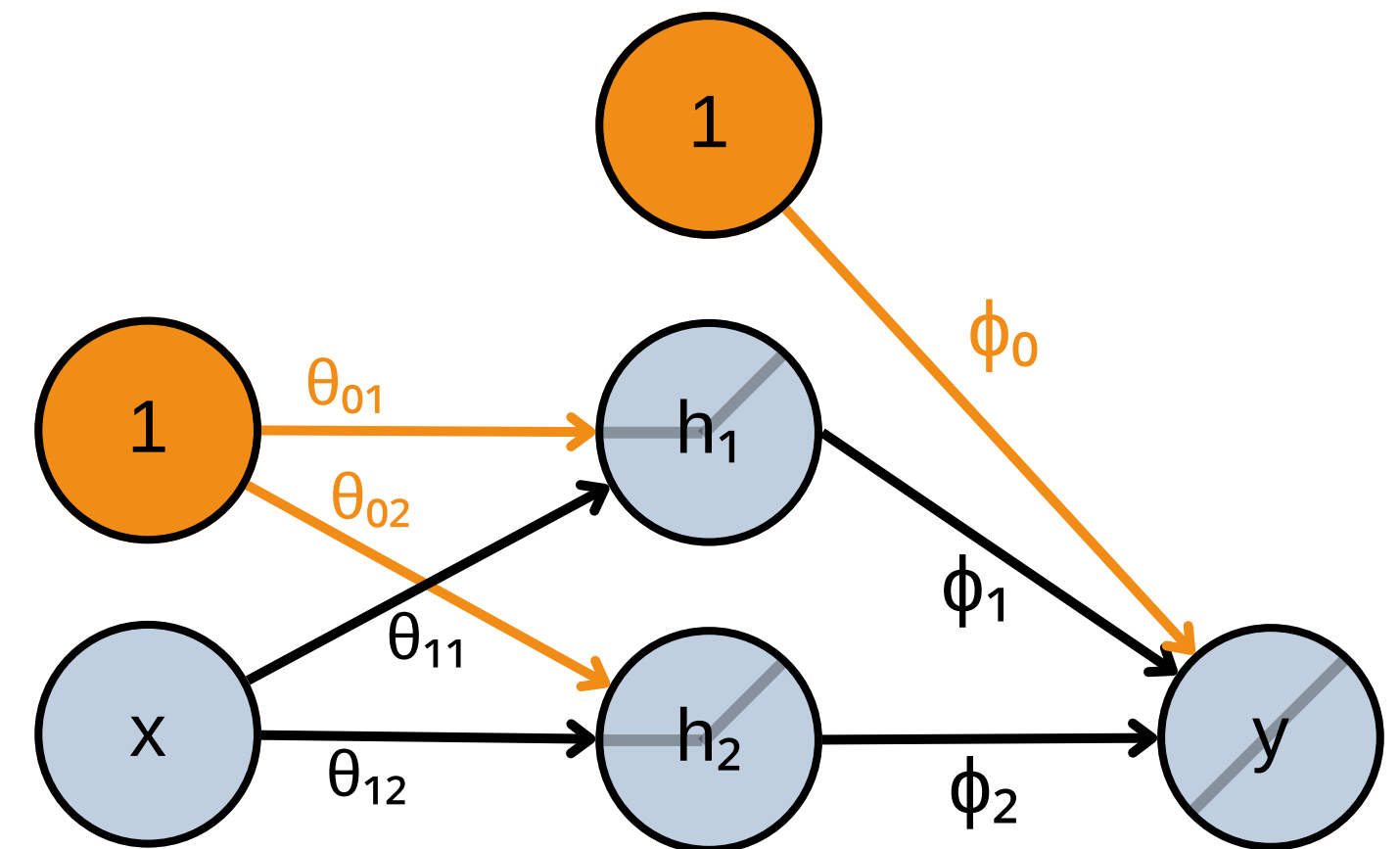
$$\frac{d\mathcal{L}}{d\phi_1} = 2\Delta y \cdot h_1$$

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

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

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

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



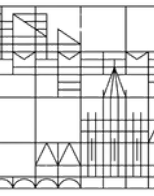


Learning Curves

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.

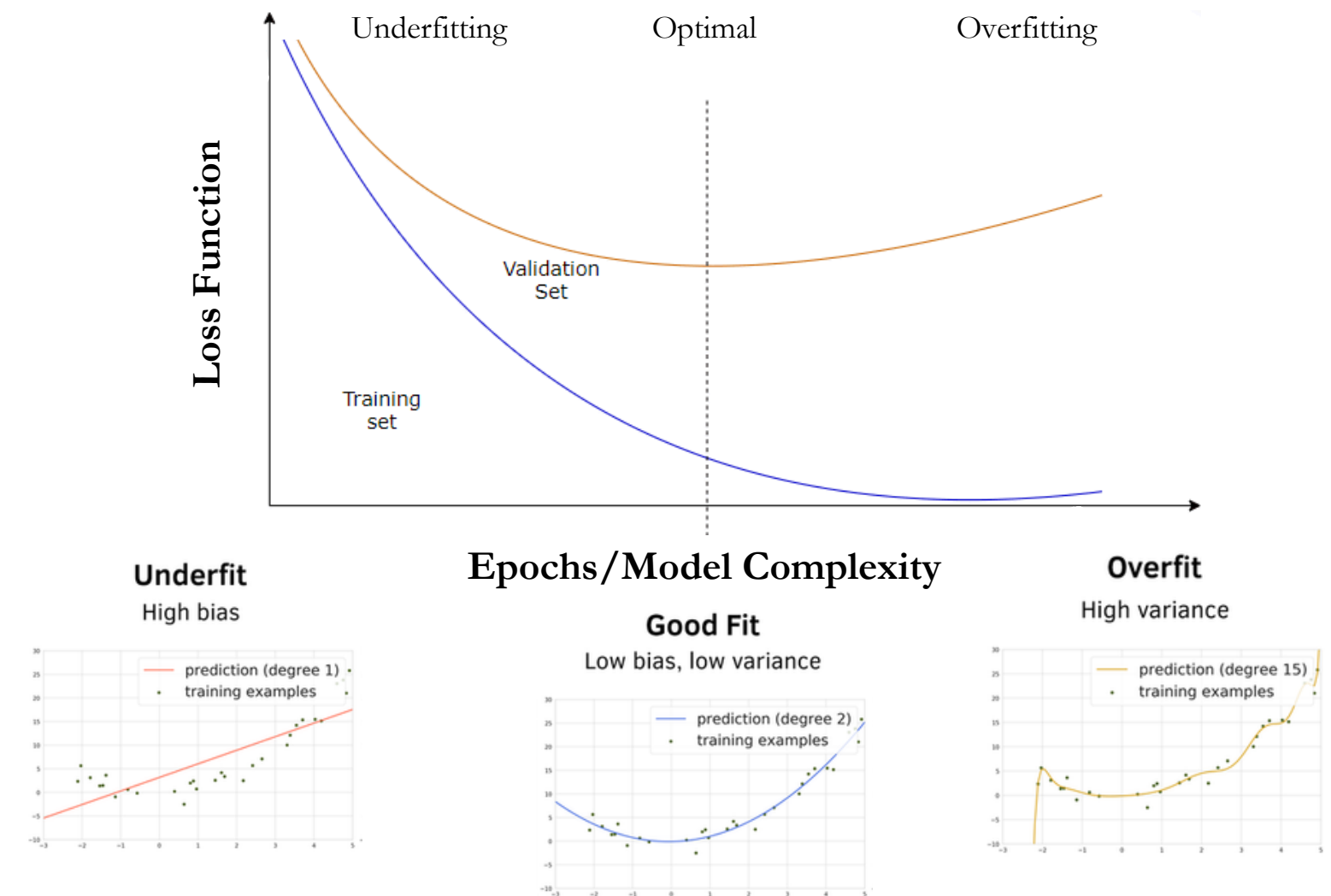


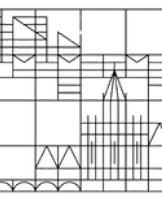
Underfitting and Overfitting

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!



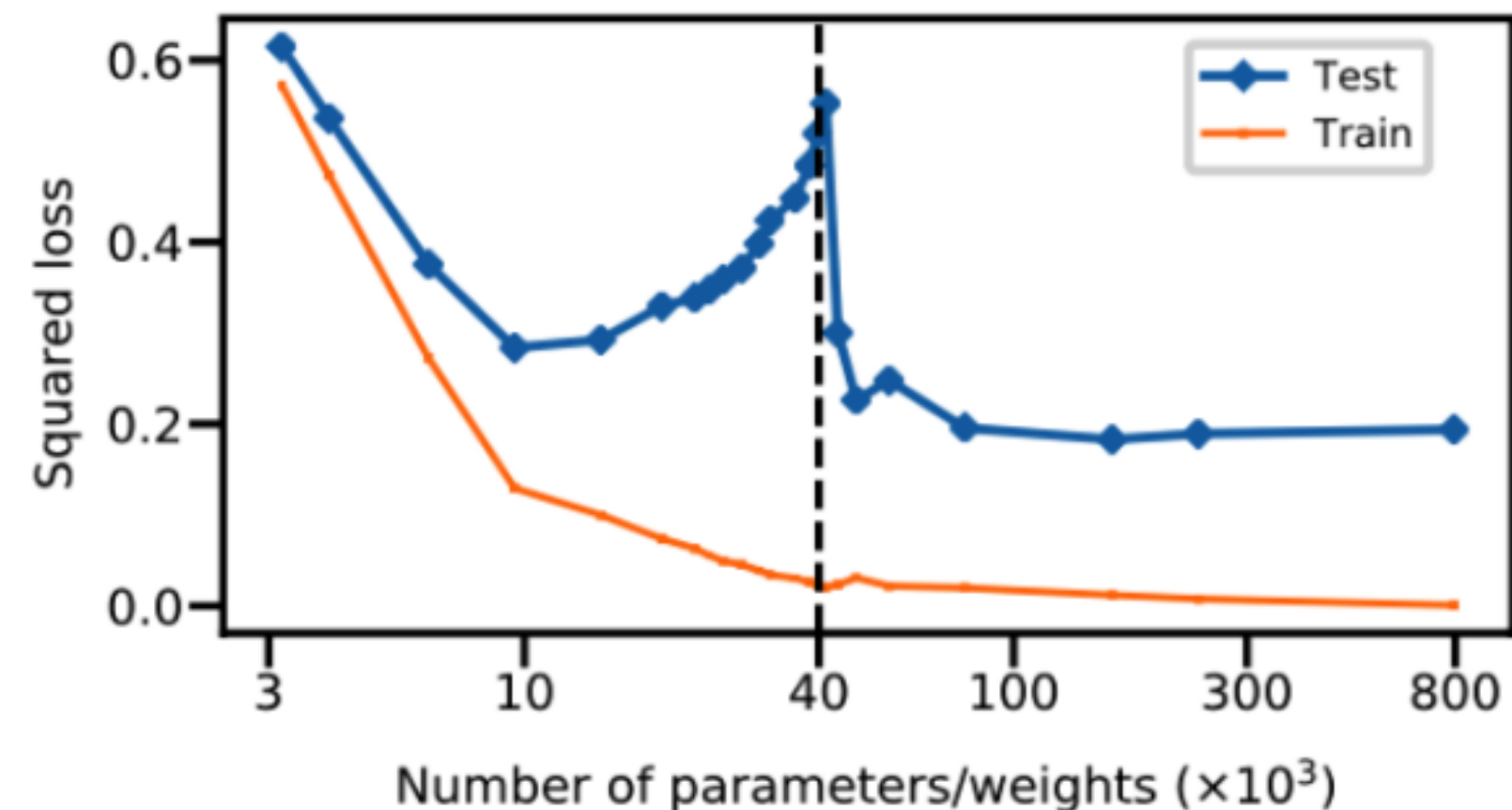


Double Descent

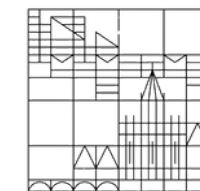
The picture described in the previous slide is the standard one, but it is a bit outdated due to the **Double Descent** phenomenon

- 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**.

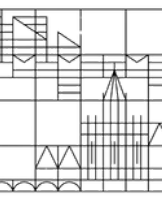


<https://arxiv.org/abs/1912.02292>



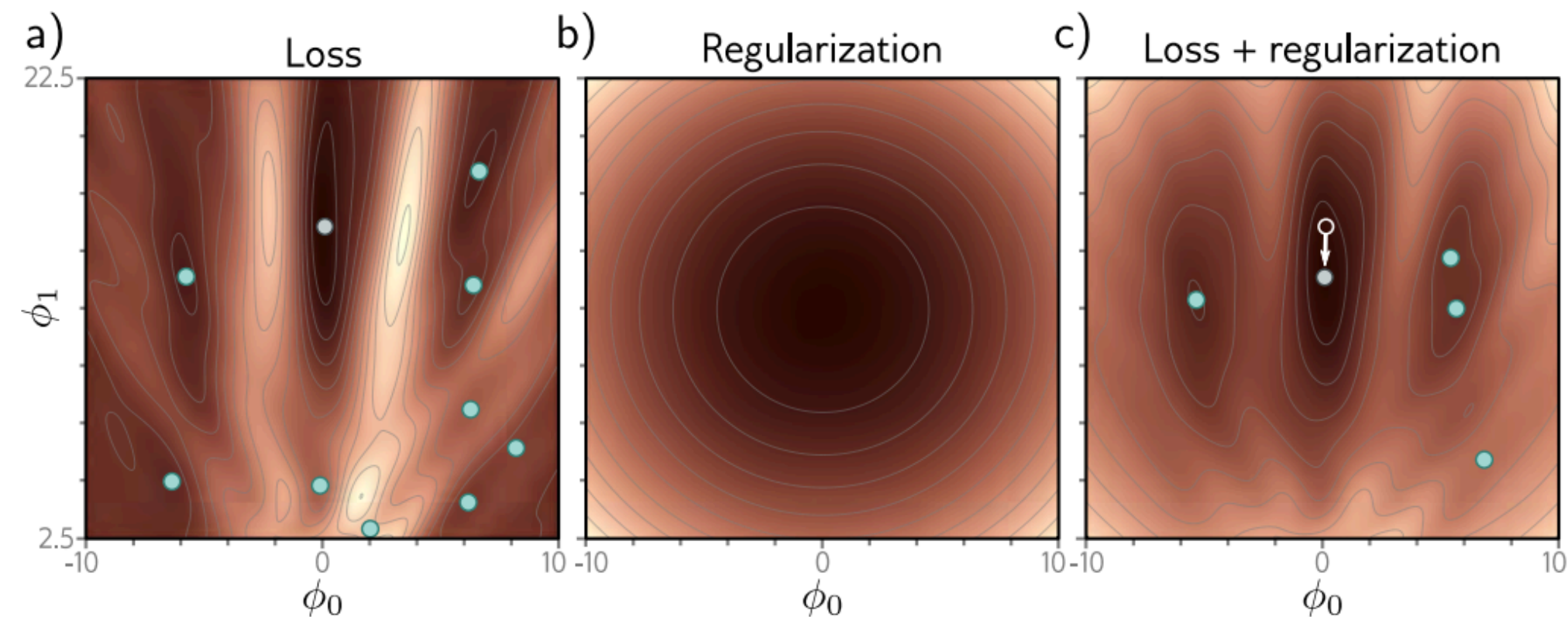
Regularizing Neural Networks

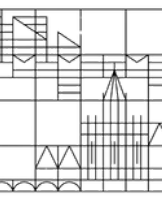




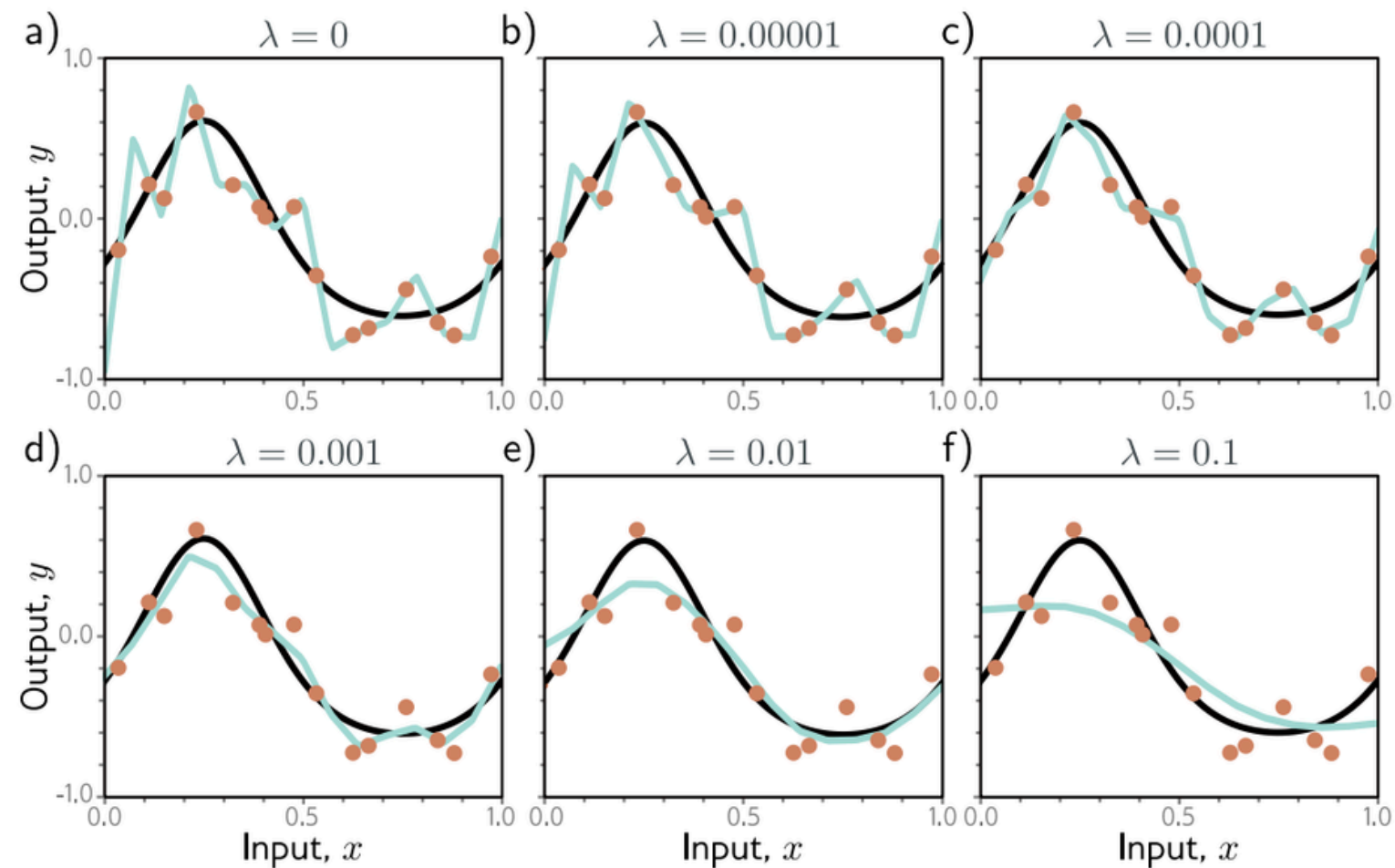
Regularization Techniques

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

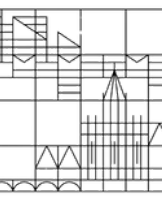
$$L' = L + \lambda \sum w_i^2$$

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.

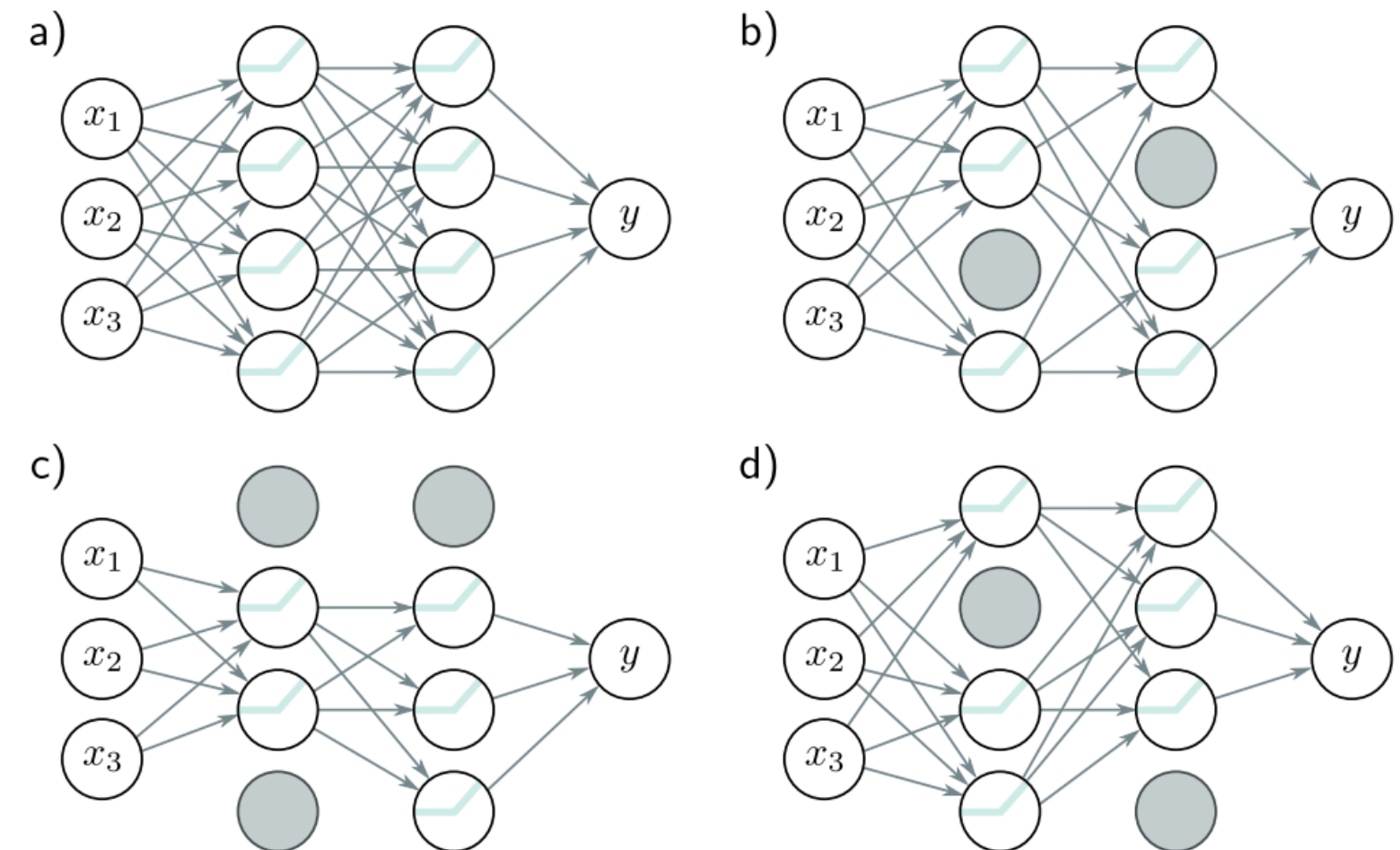
Benefits:

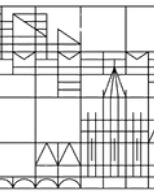
- Reduces Overfitting
- Increases Robustness

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

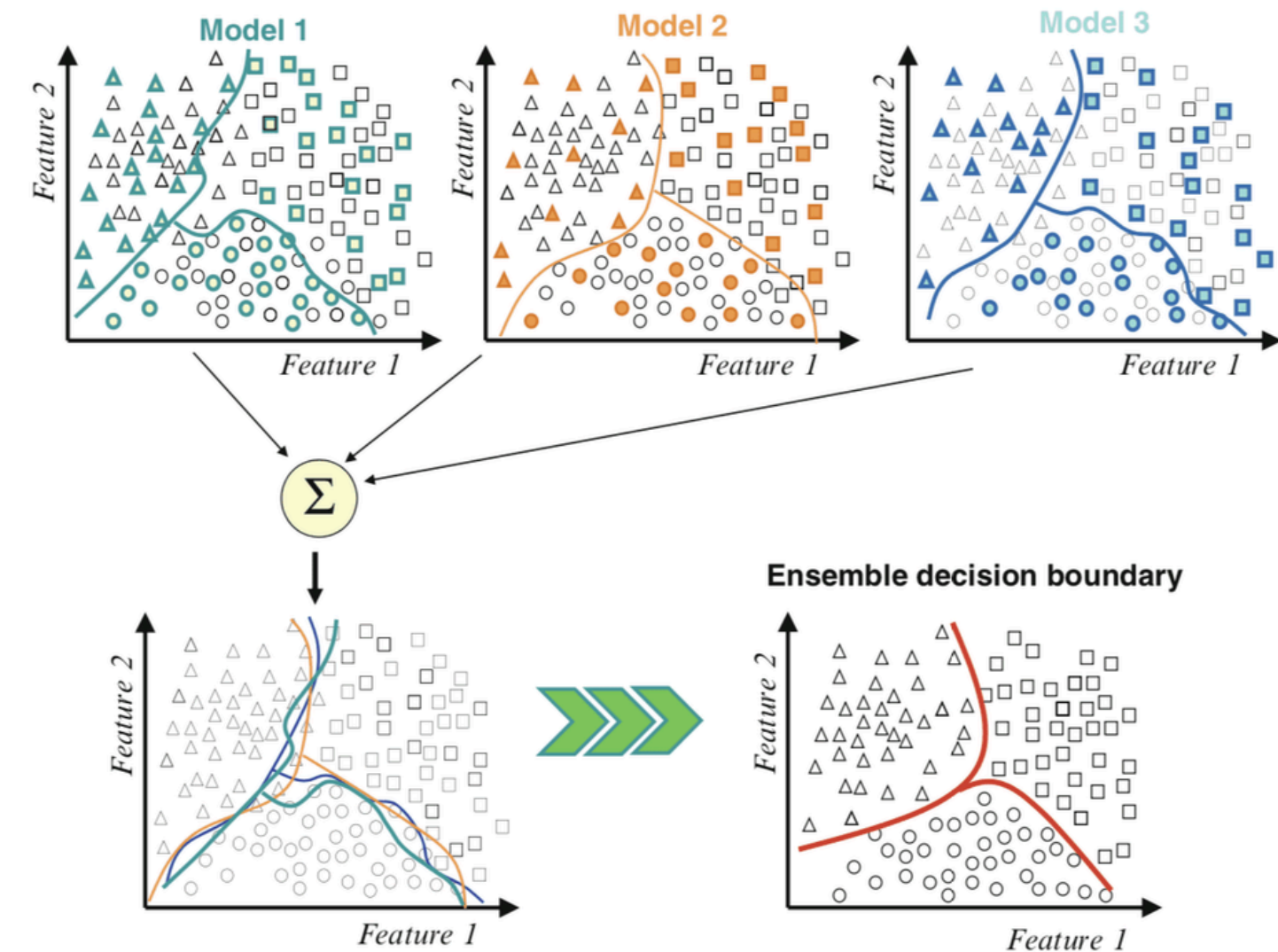
Model ensembling consists in independently training a group of different ML models on the same data. These models are then combined into a single more powerful model (random forest is one example).

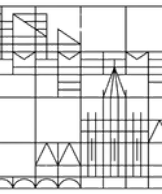
Benefits:

- Reduces Overfitting
- Increases Robustness

Implementation:

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





Early Stopping

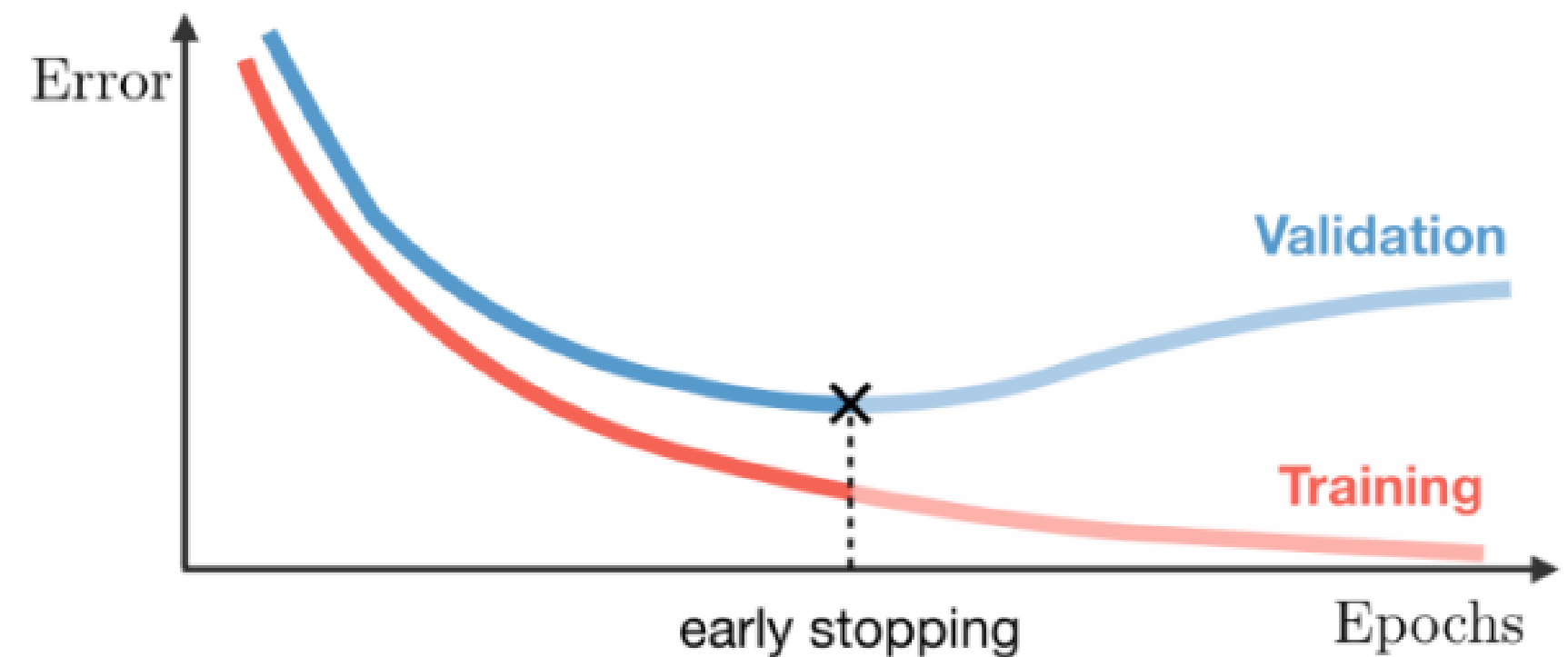
Early stopping prevents overfitting in training DNN. It works by monitoring the model's performance on a validation set during training and stopping the training process if the validation performance begins to degrade

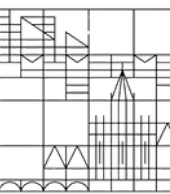
Benefits:

- Prevents Overfitting
- Optimizes Training Time

Implementation:

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





Noise Injection

Noise 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

Noise addition

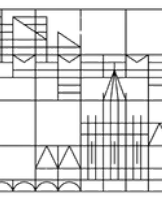


- Addition of noise
- More tolerance to quality variation of inputs

Information loss



- Parts of image ignored
- Mimics potential loss of parts of image



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

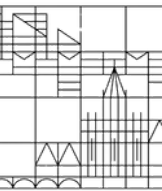
Benefits:

- Makes Dataset Balanced
- Enhances Robustness

Implementation:

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

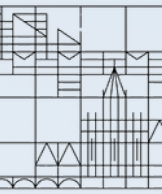
Original	Flip	Rotation
		
<ul style="list-style-type: none"> • Image without any modification 	<ul style="list-style-type: none"> • Flipped with respect to an axis for which the meaning of the image is preserved 	<ul style="list-style-type: none"> • Rotation with a slight angle • Simulates incorrect horizon calibration
Random crop	Color shift	Contrast change
		
<ul style="list-style-type: none"> • Random focus on one part of the image • Several random crops can be done in a row 	<ul style="list-style-type: none"> • Nuances of RGB is slightly changed • Captures noise that can occur with light exposure 	<ul style="list-style-type: none"> • Luminosity changes • Controls difference in exposition due to time of day



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.



Summary

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.

Improving Performances

The loss function is typically very rough and full of local minima, but there are several possible approaches to regularize the loss, they can be used alone or together