POLITECNICO
MILANO 1863

Artificial Neural Networks and Deep Learning

- Neural Networks Training and Overfitting-

Matteo Matteucci, PhD (matteo.matteucci@polimi.it)
Artificial Intelligence and Robotics Laboratory
Politecnico di Milano

AIRLAB
ARTIFICIAL INTELLIGENCE AND ROBOTICS LAB

Neural Networks are Universal Approximators

"A single hidden layer feedforward neural network with S shaped activation functions can approximate any measurable function to any desired degree of accuracy on a compact set "

Universal approximation theorem (Kurt Hornik, 1991)

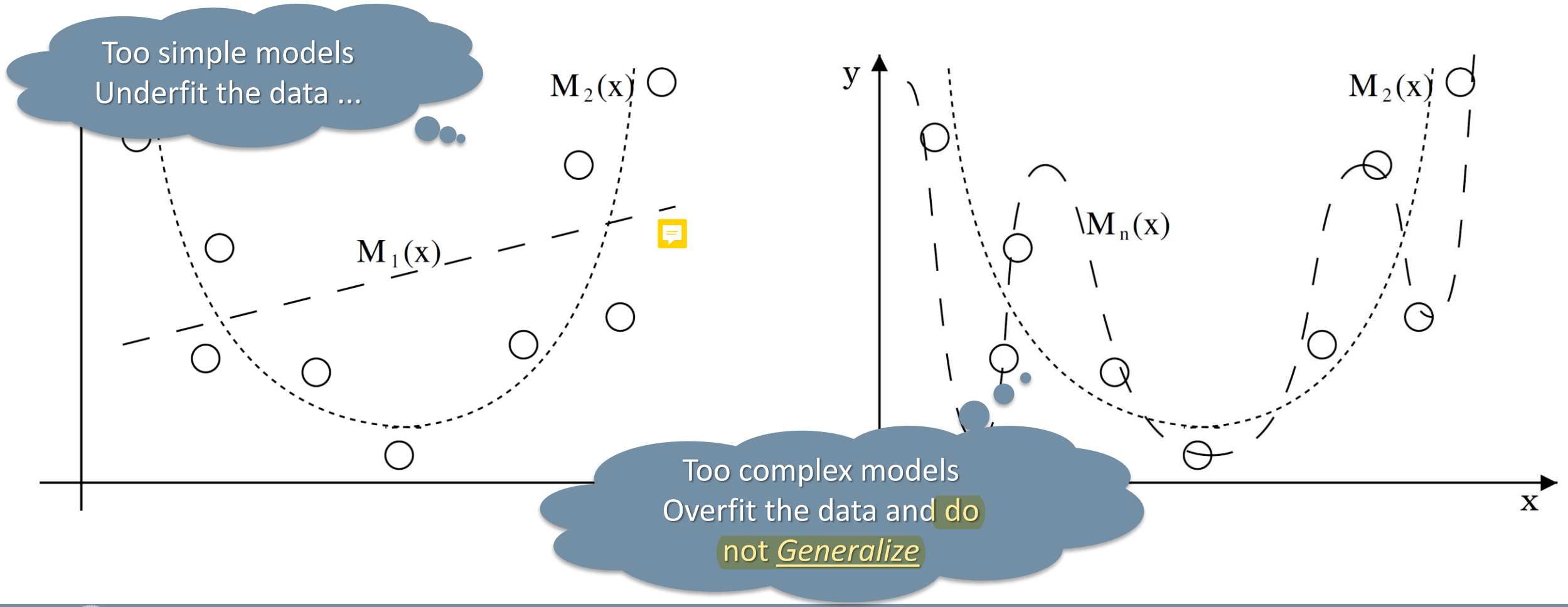
Regardless of what function we are learning; a single layer can do it …

- … but it doesn't mean we can find the necessary weights!
- … but an exponential number of hidden units may be required
- … but it might be useless in practice if it does not generalize!

"Entia non sunt multiplicanda praeter necessitatem"
William of Ockham (c 1285 – 1349)

Model Complexity

Inductive Hypothesis: A solution approximating the target function over a sufficiently large set of training examples will also approximate it over unobserved examples



How to Measure Generalization?

Training error/loss is not a good indicator of performance on future data:

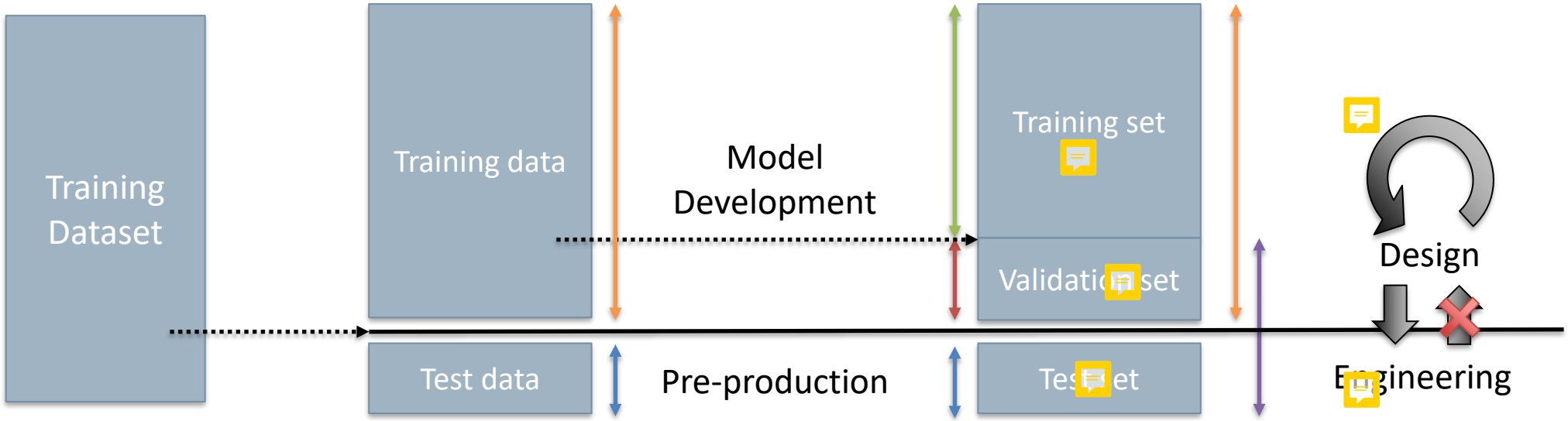
- The classifier has been learned from the very same training data, any estimate based on that data will be optimistic
- New data will probably not be exactly the same as training data
- You can find patterns even in random data

We need to test on an independent new test set

- Someone provides you a new dataset
- Split the data and hide some of them for later evaluation
- Perform random subsampling (with replacement) of the dataset

In classification preserve class distribution, i.e., stratified sampling!

Clearing the terms …

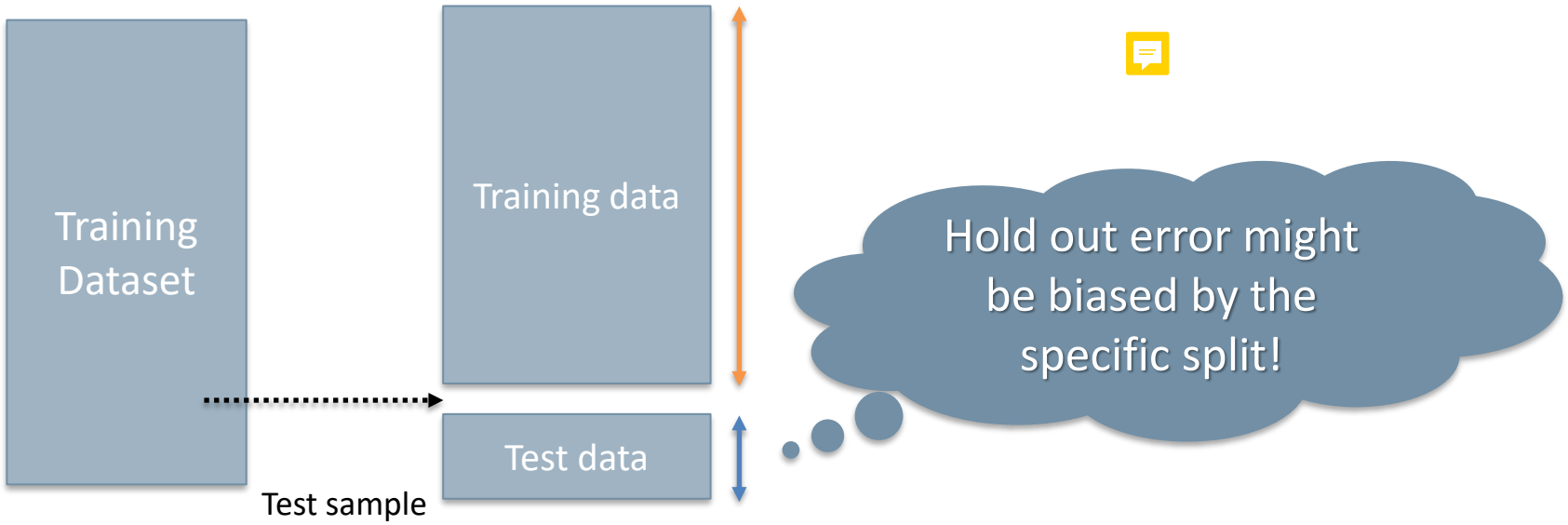


- **Training dataset**: the available data
- Training set: the data used to learn model parameters
- Test set: the data used to perform final model assessment
- Validation set: the data used to perform model selection
- Training data: used to train the model (fitting + selection)
- Validation data: used to assess the model quality (selection + assessment)

Cross-Validation

Cross-validation is the use of the training dataset to both train the model (parameter fitting + model selection) and estimate its error on new data

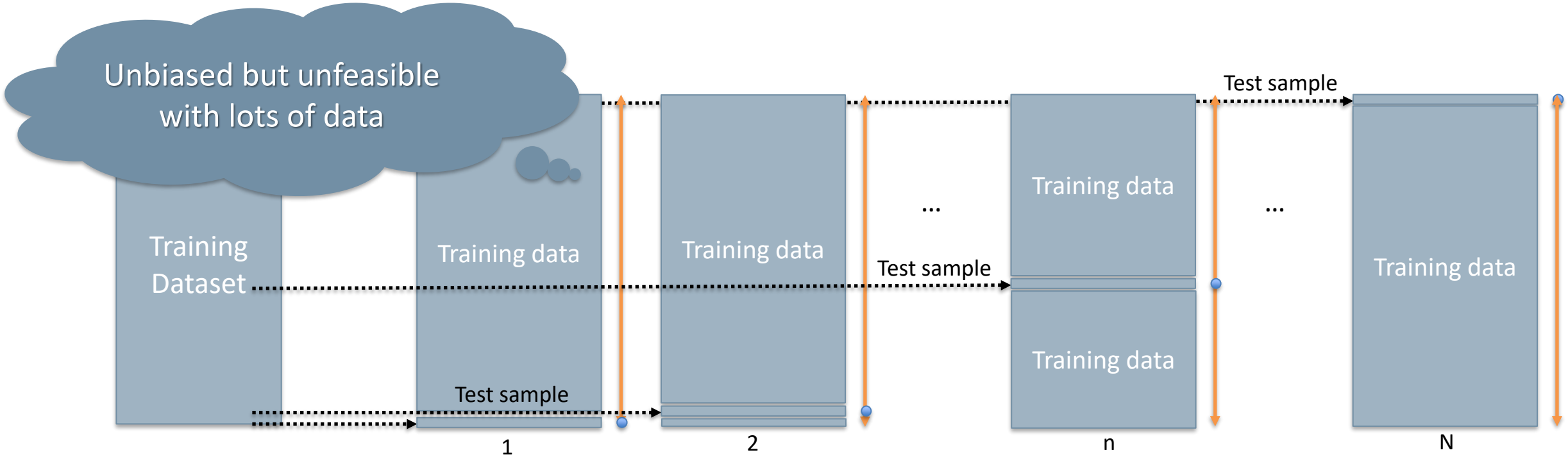
- When lots of data are available use a Hold Out set and perform validation



Cross-Validation

Cross-validation is the use of the training dataset to both train the model (parameter fitting + model selection) and estimate its error on new data

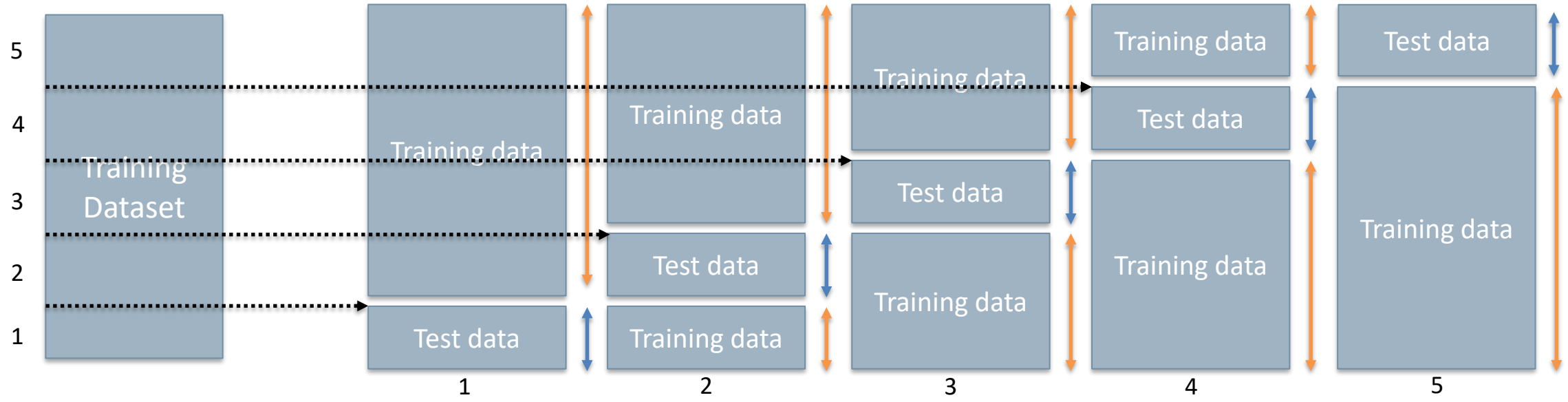
- When lots of data are available use a Hold Out set and perform validation
- When having few data available use Leave-One-Out Cross-Validation (LOOCV)



Cross-Validation

Cross-validation is the use of the training dataset to both train the model (parameter fitting + model selection) and estimate its error on new data

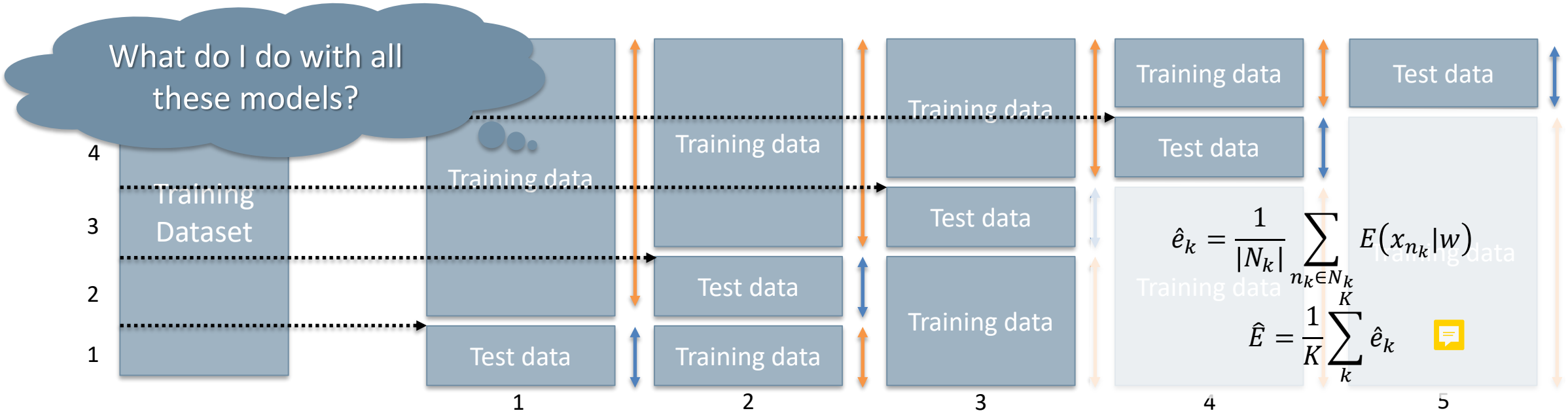
- When lots of data are available use a Hold Out set and perform validation
- When having few data available use Leave-One-Out Cross-Validation (LOOCV)
- K-fold Cross-Validation is a good trade-off (sometime better than LOOCV)



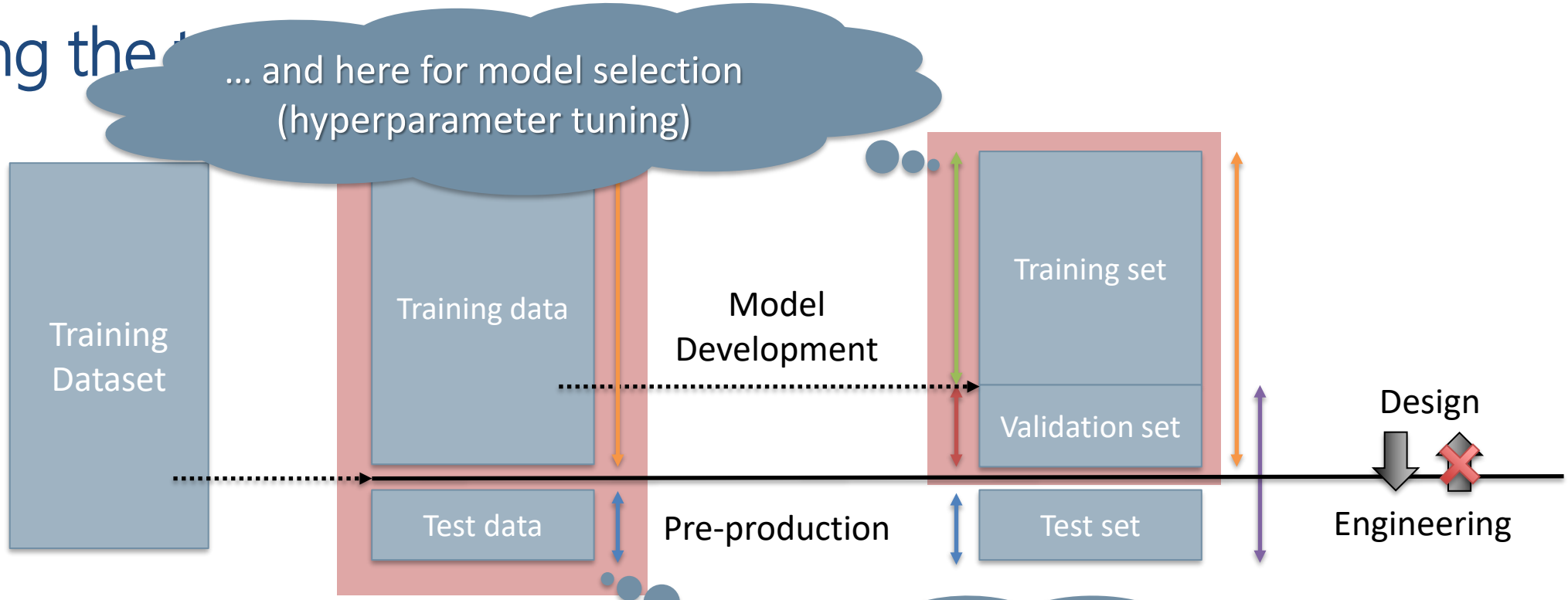
Cross-Validation

Cross-validation is the use of the training dataset to both train the model (parameter fitting + model selection) and estimate its error on new data

- When lots of data are available use a Hold Out set and perform validation
- When having few data available use Leave-One-Out Cross-Validation (LOOCV)
- K-fold Cross-Validation is a good trade-off (sometime better than LOOCV)



Clearing the



- **Training dataset**: the available
- **Training set**: the data used
- **Test set**: the data used to perform final assessment
- **Validation set**: the data used to perform model selection

We can apply K-fold Cross-Validation at this level for model assessment …

Beware the number of models you get and how much it cost to train!

he model (fitting + selection)

used to assess the model quality (selection + assessment)

POLITECNICO
MILANO 1863

Artificial Neural Networks and Deep Learning

- Preventing Neural Networks Overfitting -

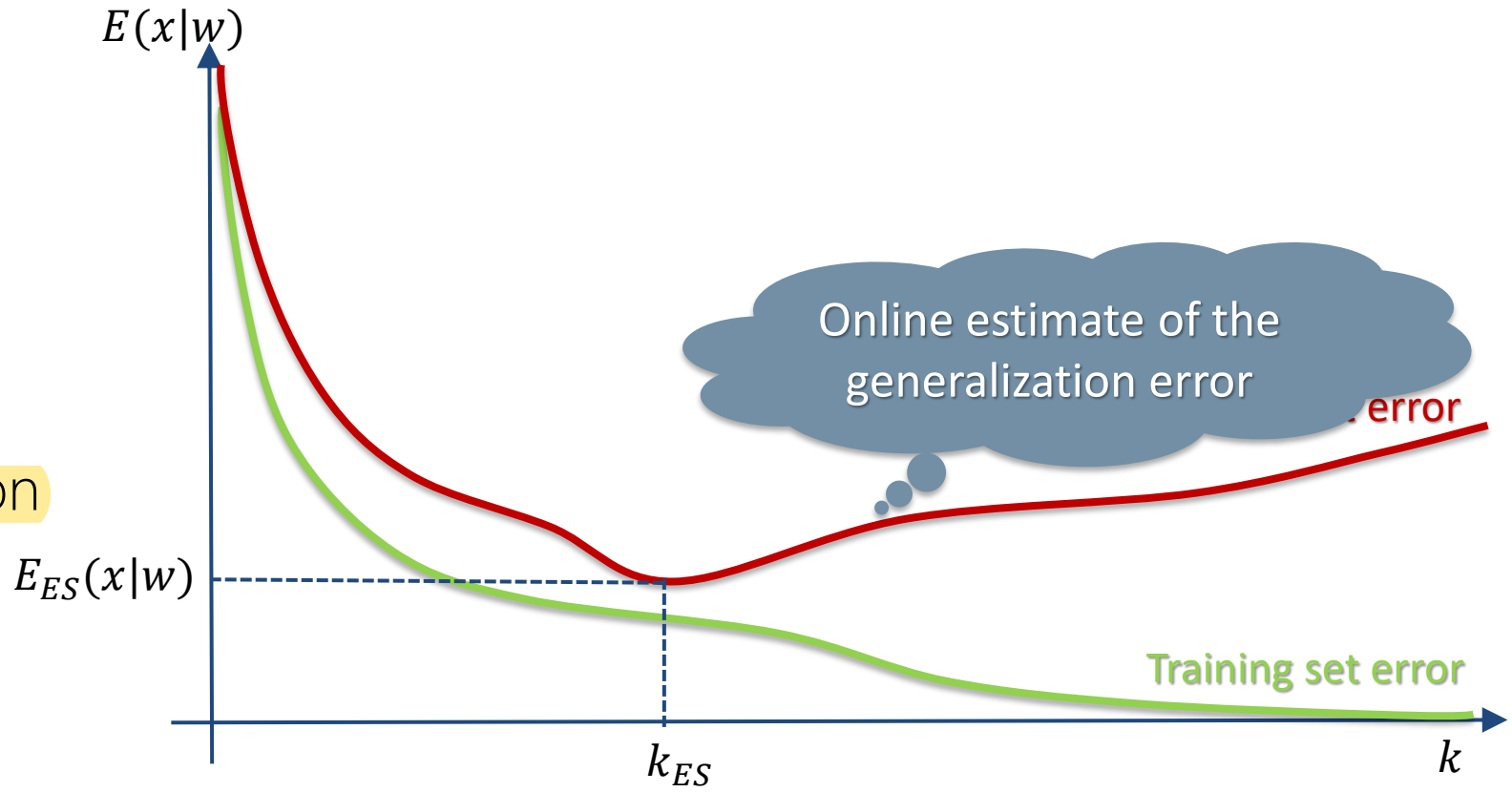
Matteo Matteucci, PhD (matteo.matteucci@polimi.it)
Artificial Intelligence and Robotics Laboratory
Politecnico di Milano

AIRLAB
ARTIFICIAL INTELLIGENCE AND ROBOTICS LAB

Early Stopping: Limiting Overfitting by Cross-validation

Overfitting networks show a monotone *training error* trend (on average with SGD) as the number of gradient descent iterations k, but they lose generalization at some point …

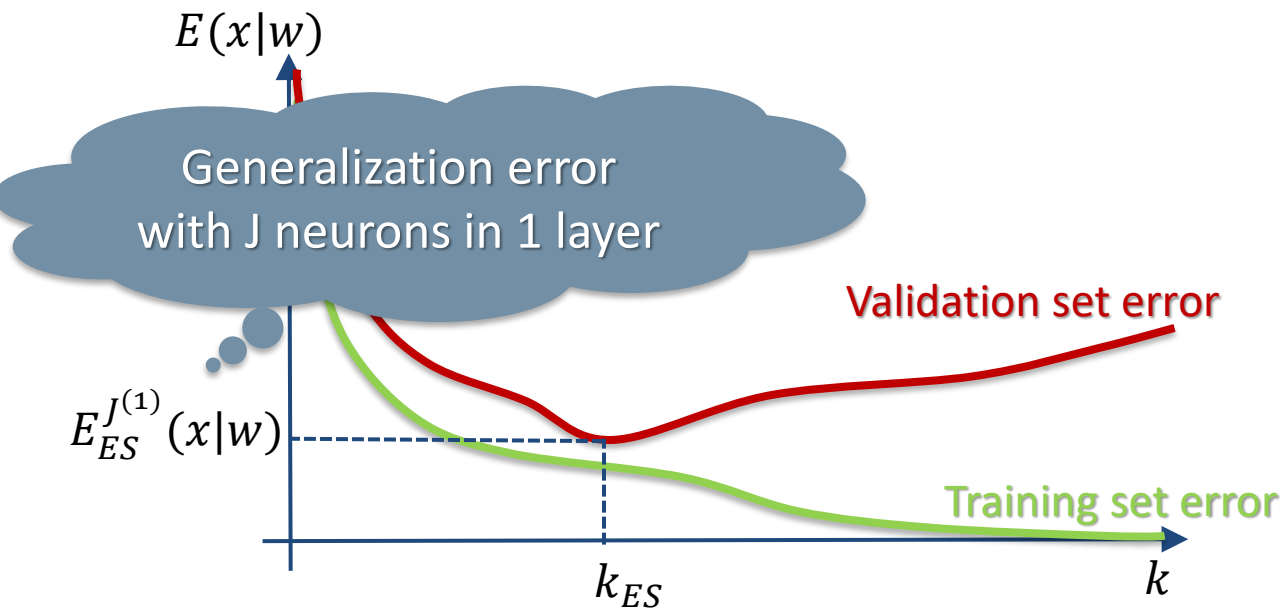
- Hold out some data
- Train on the training set
- Perform cross-validation on the hold out set
- Stop train when validation error increases

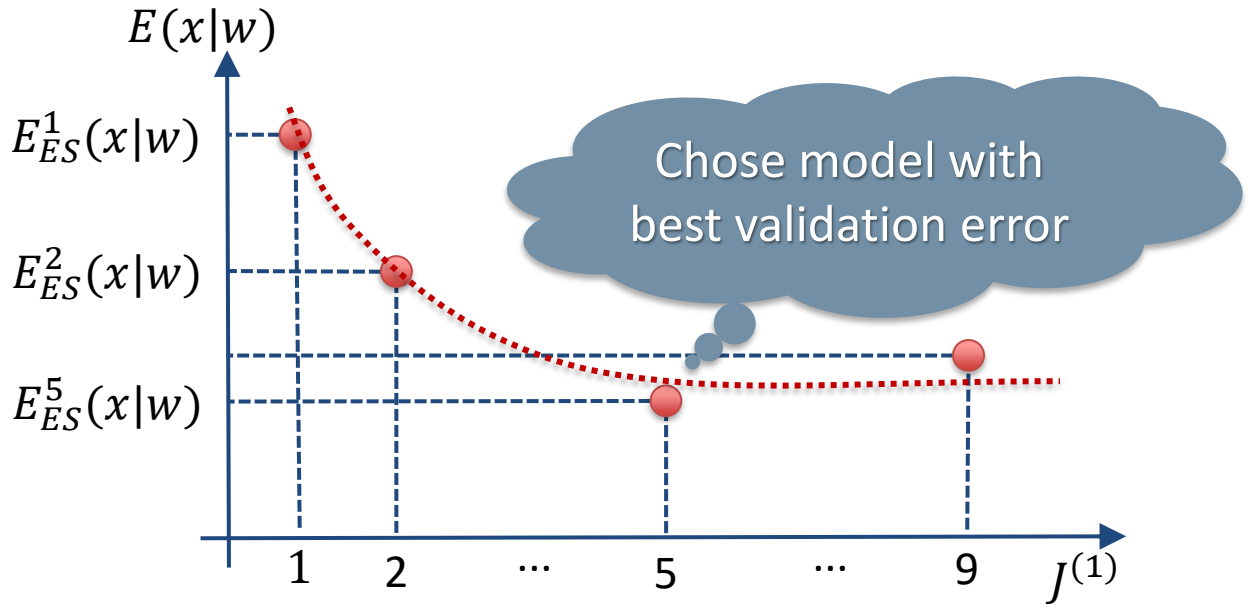


Cross-validation and Hyperparameters Tuning

Model selection and evaluation happens at different levels:

- Parameters level, i.e, when we learn the weights $\boldsymbol{w}$ for a neural network
- Hyperparameters level, i.e., when we chose the number of layers $\boldsymbol{L}$ or the number of hidden neurons $\boldsymbol{J^{(l)}}$ for a given layer





Weight Decay: Limiting Overfitting by Weights Regularization

Regularization is about constraining the model «freedom», based on a-priori assumption on the model, to reduce overfitting

So far we have maximized the data likelihood:

Maximum Likelihood

$$w_{MLE} = argmax_w \, P(D|w)$$

We can reduce model «freedom» by using a Bay

Make assumption on parameters (a-priori) distribution

Maximum A-Posteriori

$$w_{MAP} = argmax_w \, P(w|D)$$
$$= argmax_w \, P(D|w) \cdot P(w)$$

Small weights observed to improve generalization of neural networks:

$$P(w) \sim N(0, \sigma_w^2)$$

Weight Decay: Limiting Overfitting by Weights Regularization

$$\hat{w} = argmax_w\, P(w|D) = argmax_w\, P(D|w)\, P(w)$$

$$= argmax_w \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\left(t_n - g(x_n|w)\right)^2}{2\sigma^2}} \prod_{q=1}^{Q} \frac{1}{\sqrt{2\pi}\sigma_w} e^{-\frac{\left(w_q\right)^2}{2\sigma_w^2}}$$

$$= argmin_w \sum_{n=1}^{N} \frac{\left(t_n - g(x_n|w)\right)^2}{2\sigma^2} + \sum_{q=1}^{Q} \frac{\left(w_q\right)^2}{2\sigma_w^2}$$

$$= argmin_w \underbrace{\sum_{n=1}^{N} \left(t_n - g(x_n|w)\right)^2}_{\text{Fitting}} + \underbrace{\gamma \sum_{q=1}^{Q} \left(w_q\right)^2}_{\text{Regularization}}$$

Here it comes another loss function!!!

Recall Cross-validation and Hyperparameters Tuning

You can use cross-validaton to select the proper γ:

- Split data in training and validation sets
- Minimize for different values of γ

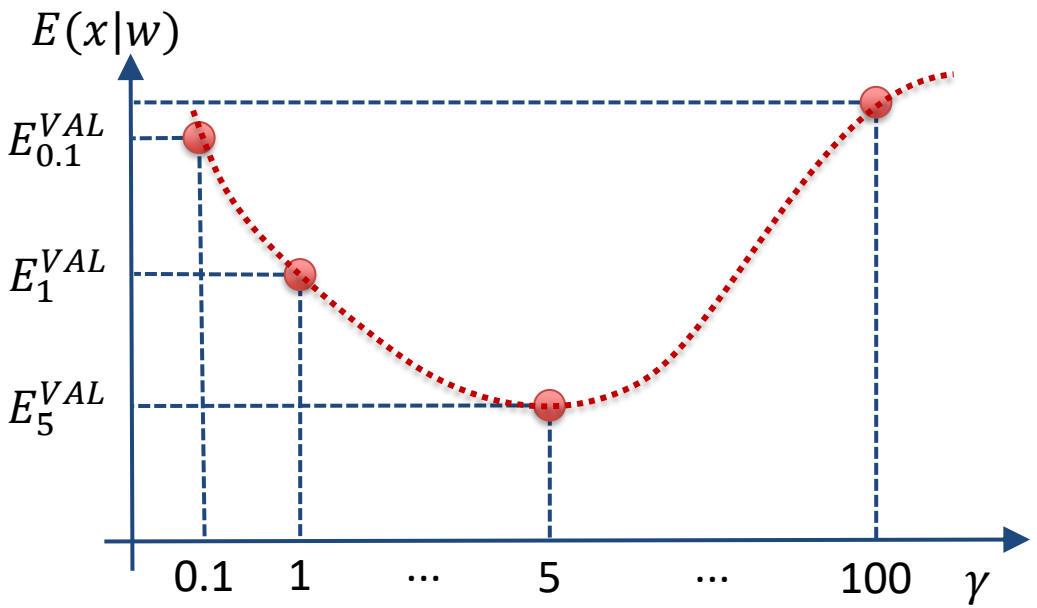
$$E_{\gamma}^{TRAIN} = \sum_{n=1}^{N_{TRAIN}} \left(t_n - g(x_n|w)\right)^2 + \gamma \sum_{q=1}^{Q} \left(w_q\right)^2$$

- Evaluate the model

$$E_{\gamma}^{VAL} = \sum_{n=1}^{N_{VAL}} \left(t_n - g(x_n|w)\right)^2$$

- Chose the γ^* with the best validation error
- Put back all data together and minimize

$$E_{\gamma^*} = \sum_{n=1}^{N} \left(t_n - g(x_n|w)\right)^2 + \gamma^* \sum_{q=1}^{Q} \left(w_q\right)^2$$

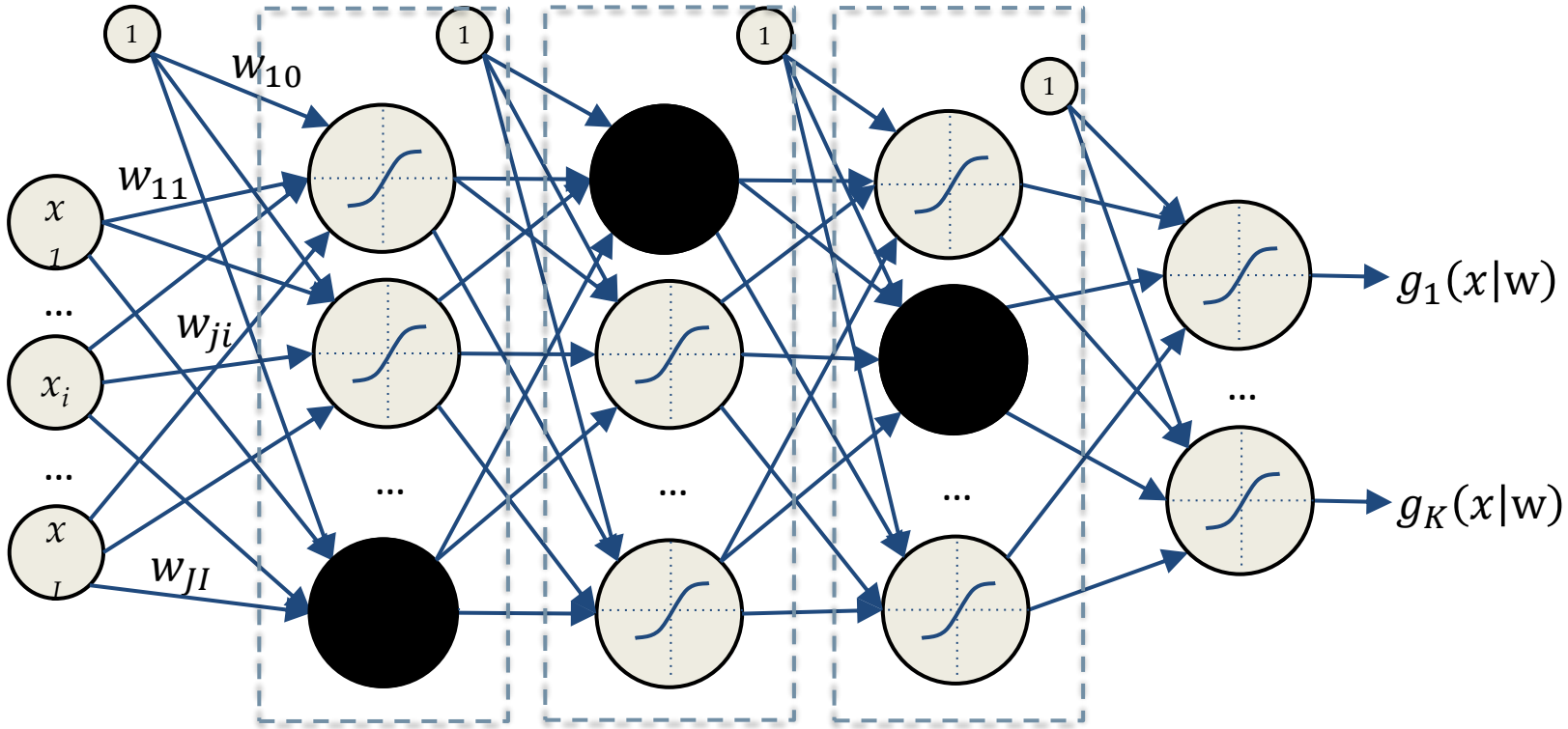


Chose $\gamma^* = 5$ with best validation error

Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force to learn an independent feature preventing hidden units to rely on other units (co-adaptation):

- Each hidden unit is set to zero with $p_j^{(l)}$ probability, e.g., $p_j^{(l)} = 0.3$



$$m^{(l)} = [m_1^{(l)}, \dots m_{J(l)}^{(l)}]$$

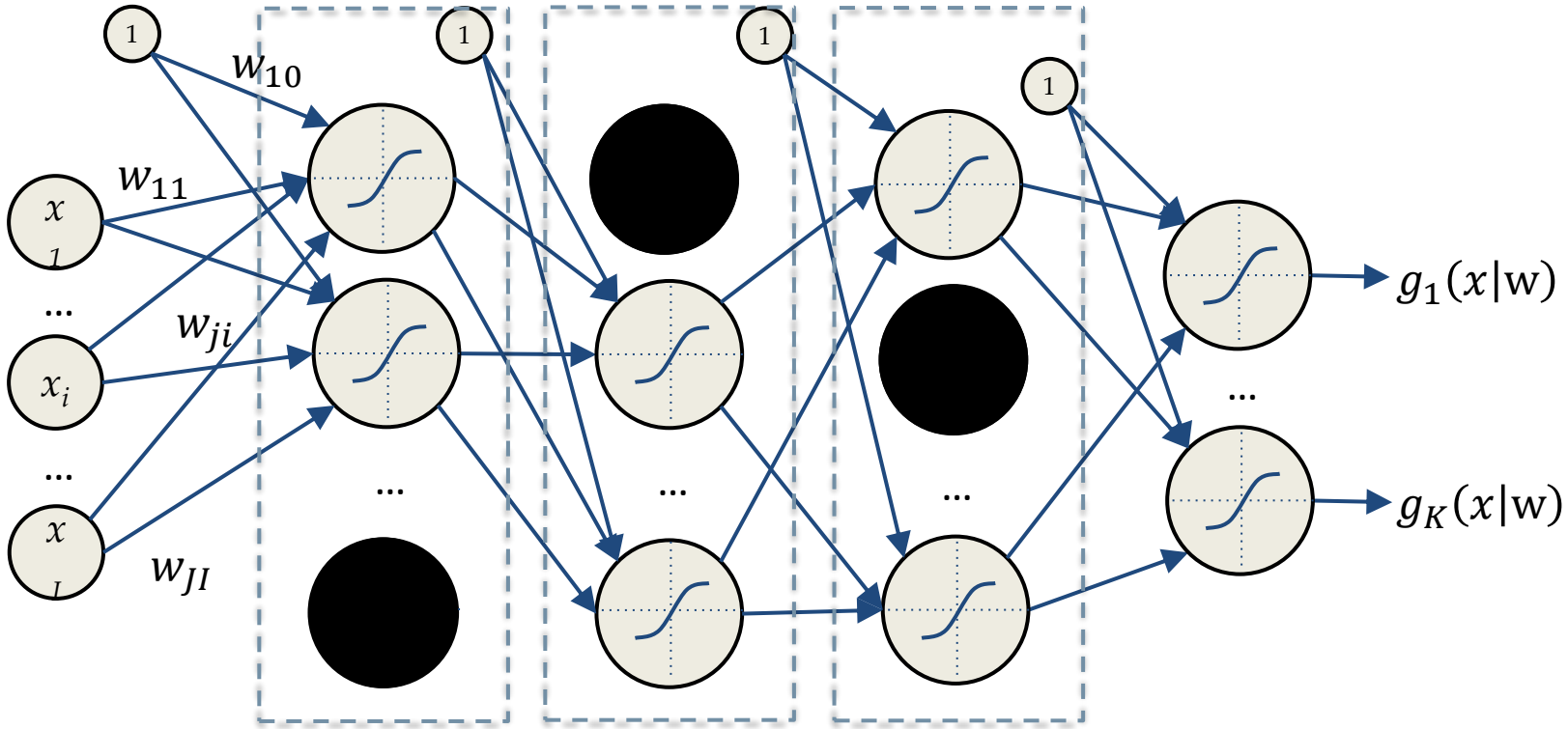
$$m_j^{(l)} \sim Be\left(p_j^{(l)}\right)$$

$$h^{(l)}(W^{(l)} h^{(l-1)} \odot m^{(l)})$$

Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force to learn an independent feature preventing hidden units to rely on other units (co-adaptation):

- Each hidden unit is set to zero with $p_j^{(l)}$ probability, e.g., $p_j^{(l)} = \mathbf{0}.3$



$$m^{(l)} = [m_1^{(l)}, \dots m_{J(l)}^{(l)}]$$

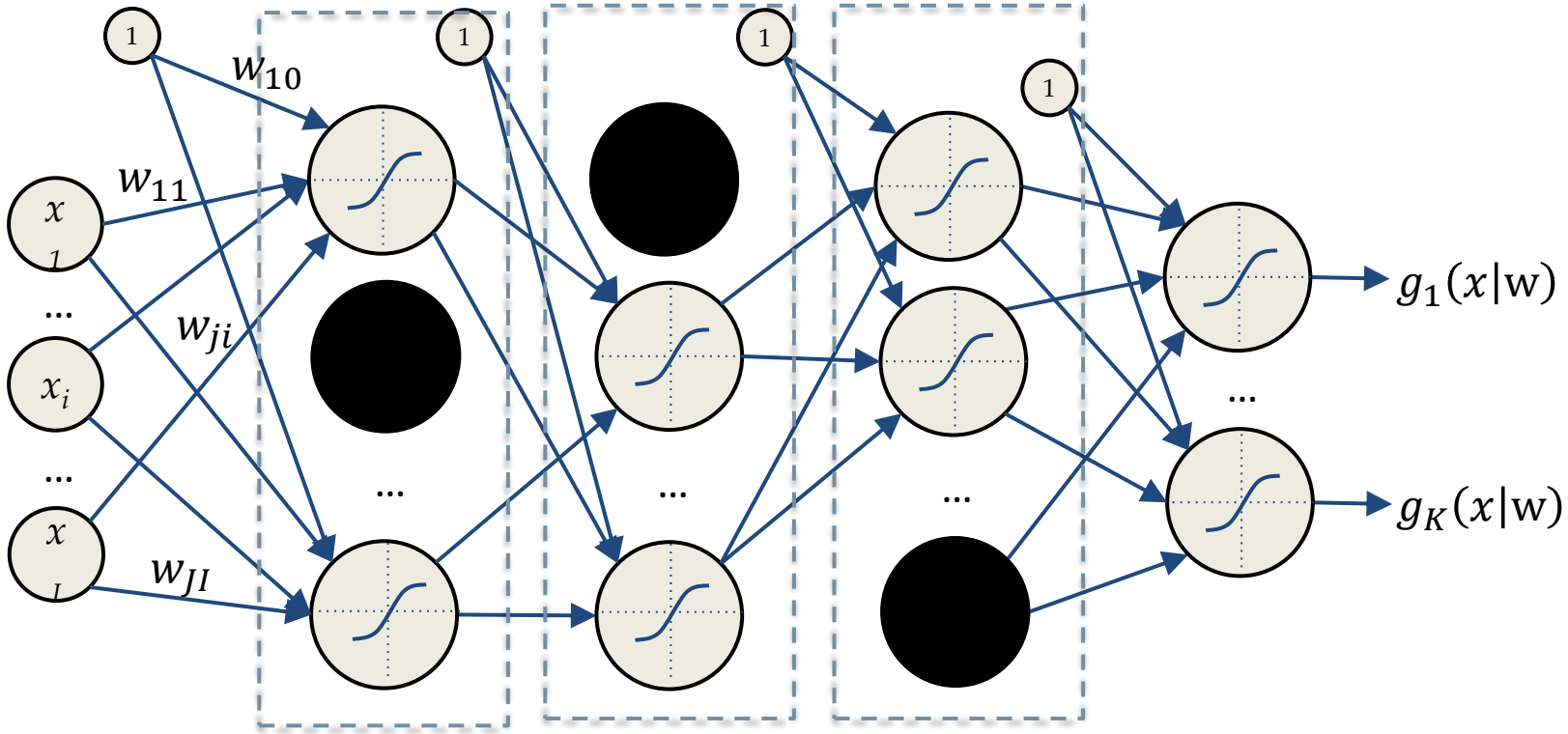
$$m_j^{(l)} \sim Be\left(p_j^{(l)}\right)$$

$$h^{(l)}(W^{(l)} h^{(l-1)} \odot m^{(l)})$$

Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force to learn an independent feature preventing hidden units to rely on other units (co-adaptation):

- Each hidden unit is set to zero with $p_j^{(l)}$ probability, e.g., $p_j^{(l)} = \mathbf{0}.3$



$$m^{(l)} = [m_1^{(l)}, \dots m_{J^{(l)}}^{(l)}]$$

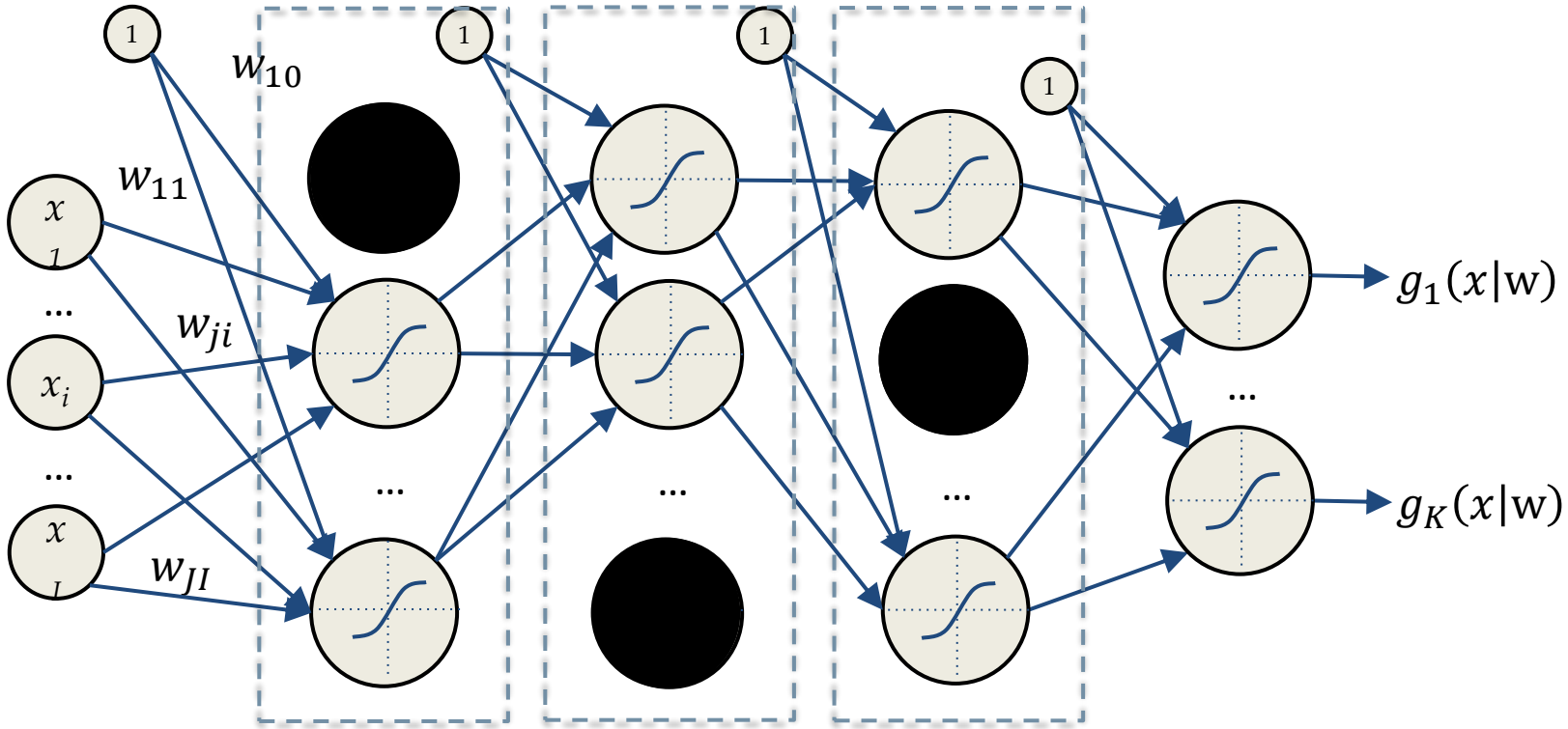
$$m_j^{(l)} \sim Be\left(p_j^{(l)}\right)$$

$$h^{(l)}(W^{(l)} h^{(l-1)} \odot m^{(l)})$$

Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force to learn an independent feature preventing hidden units to rely on other units (co-adaptation):

- Each hidden unit is set to zero with $p_j^{(l)}$ probability, e.g., $p_j^{(l)} = \mathbf{0}.3$



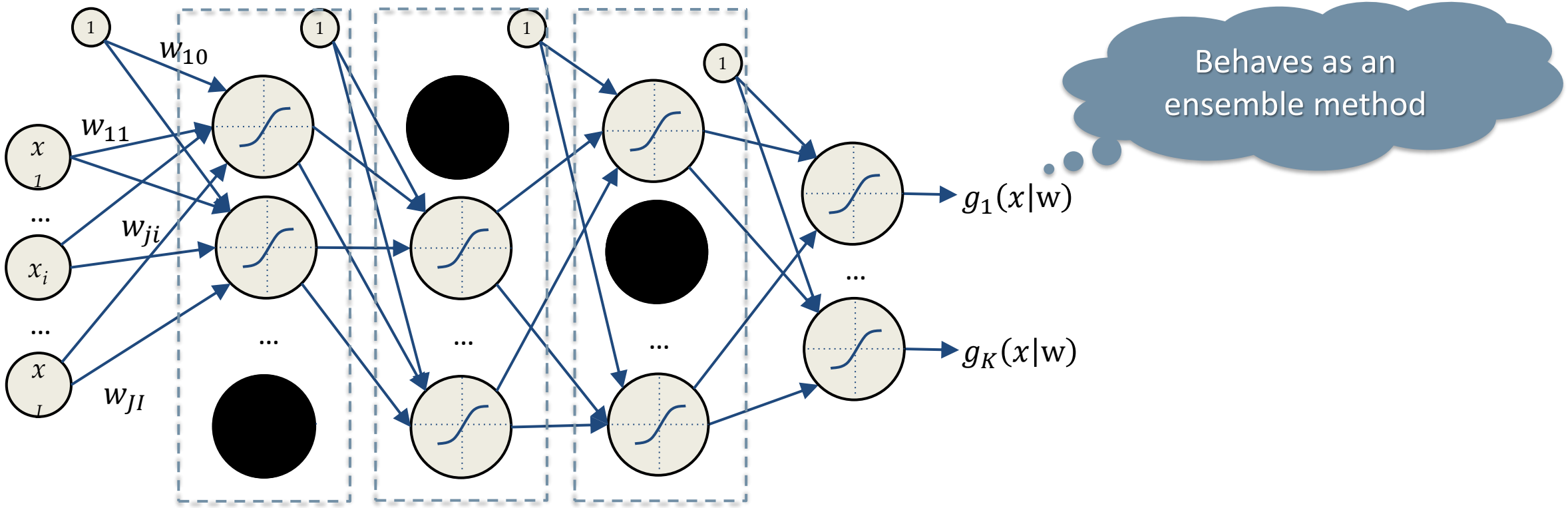
$$m^{(l)} = [m_1^{(l)}, \dots m_{J(l)}^{(l)}]$$

$$m_j^{(l)} \sim Be\left(p_j^{(l)}\right)$$

$$h^{(l)}(W^{(l)} h^{(l-1)} \odot m^{(l)})$$

Dropout: Limiting Overfitting by Stochastic Regularization

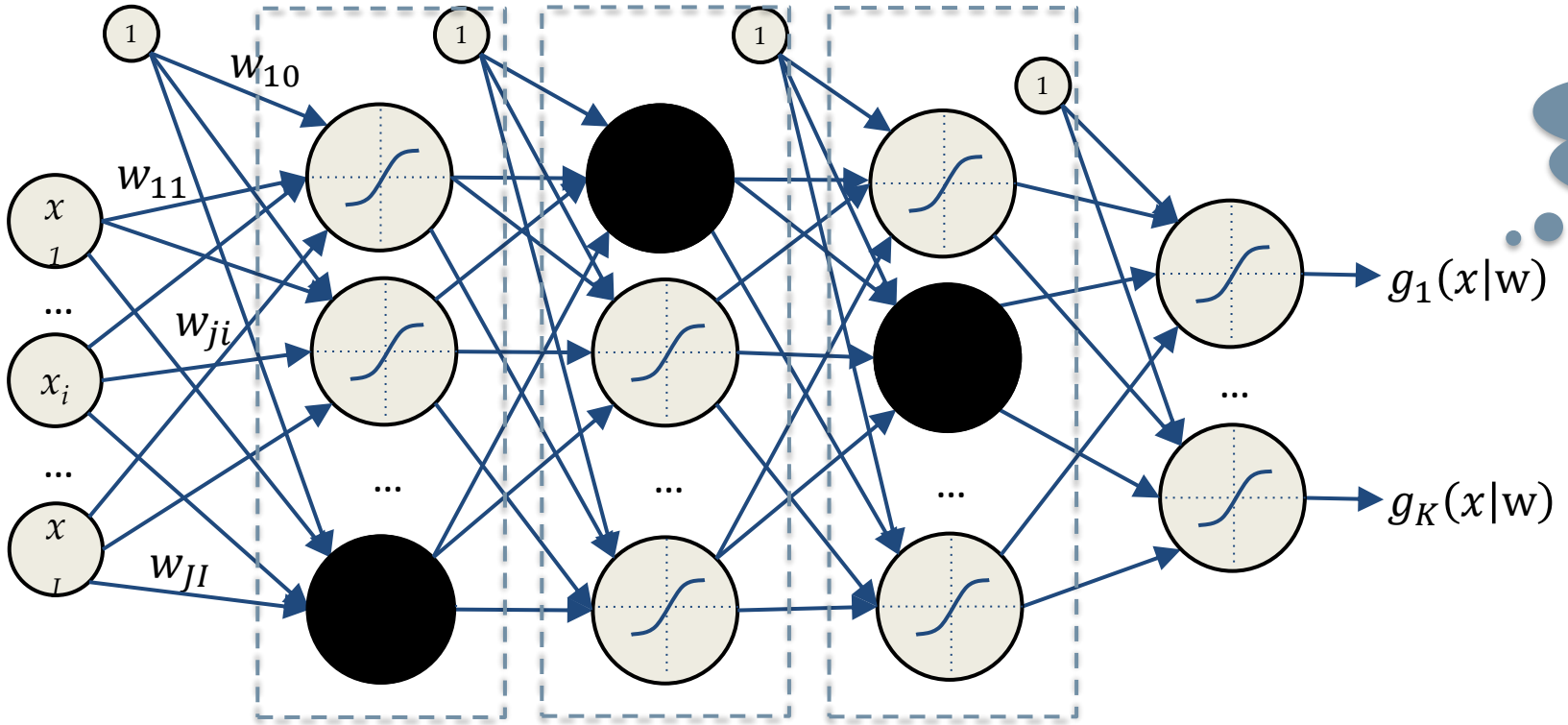
Dropout trains *weaker classifiers*, on different mini- batches and then at test time we implicitly average the responses of all ensemble members.



Dropout: Limiting Overfitting by Stochastic Regularization

Dropout trains *weaker classifiers*, on different mini- batches and then at test time we implicitly average the responses of all ensemble members.

At testing time we remove masks and average output (by weight scaling)



Behaves as an ensemble method

POLITECNICO
MILANO 1863

Artificial Neural Networks and Deep Learning

- Tips and Tricks in Neural Networks Training -

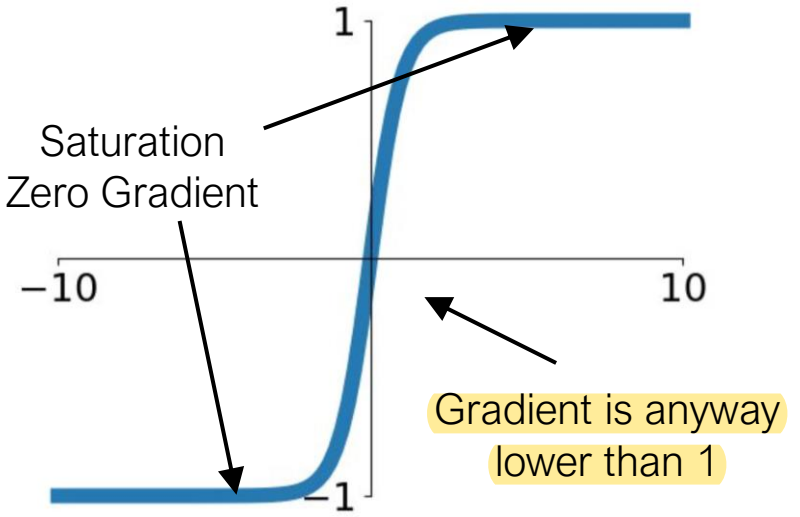
Matteo Matteucci, PhD (matteo.matteucci@polimi.it)
Artificial Intelligence and Robotics Laboratory
Politecnico di Milano

AIRLAB
ARTIFICIAL INTELLIGENCE AND ROBOTICS LAB

Better Activation Functions

Activation functions such as Sigmoid or Tanh satur

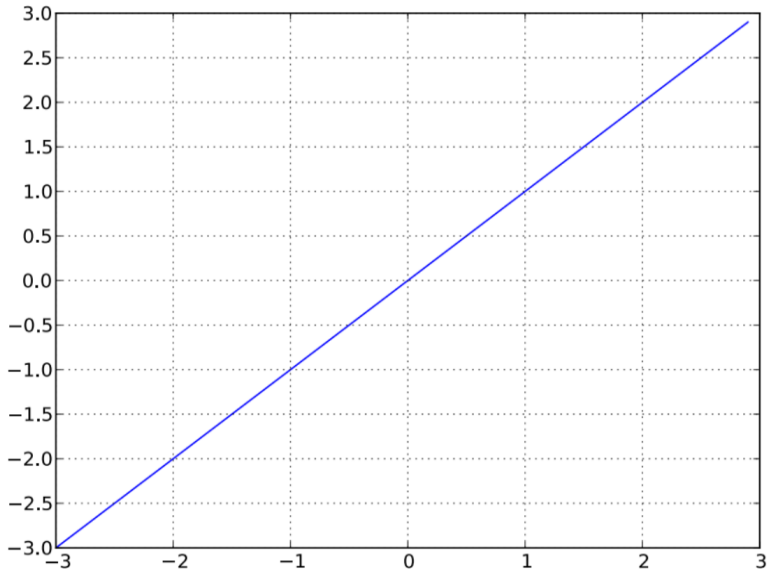
- Gradient is close to zero
- Backprop. requires gradient multiplications
- Gradient faraway from the output vanishes
- Learning in deep networks does not happen



$$\frac{\partial E(w_{ji}^{(1)}))}{\partial w_{ji}^{(1)}} = -2\sum_{n}^{N}\left(t_n - g_1(x_n, w)\right) \cdot g_1'(x_n, w) \cdot w_{1j}^{(2)} \cdot h_j'\left(\sum_{j=0}^{J} w_{ji}^{(1)} \cdot x_{i,n}\right) \cdot x_i$$

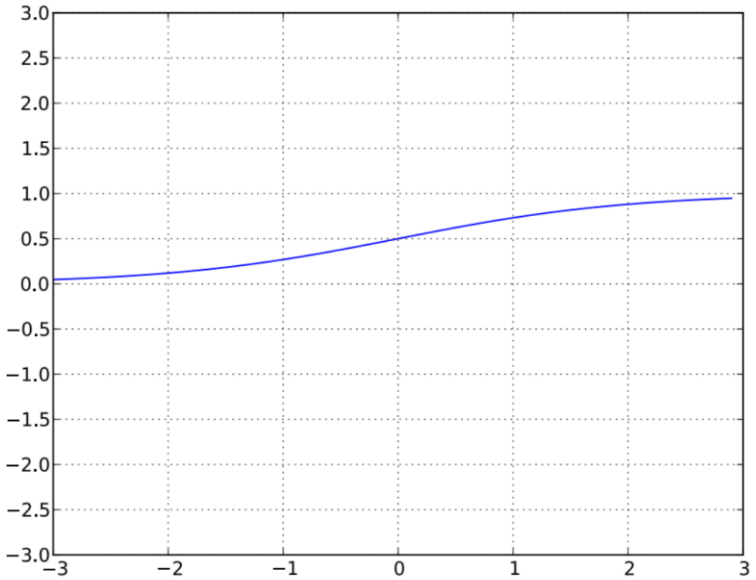
This is a well-known problem in Recurrent Neural Networks, but it affects also deep networks, and it has always hindered neural network training …

Classic Activation Functions and their Derivatives



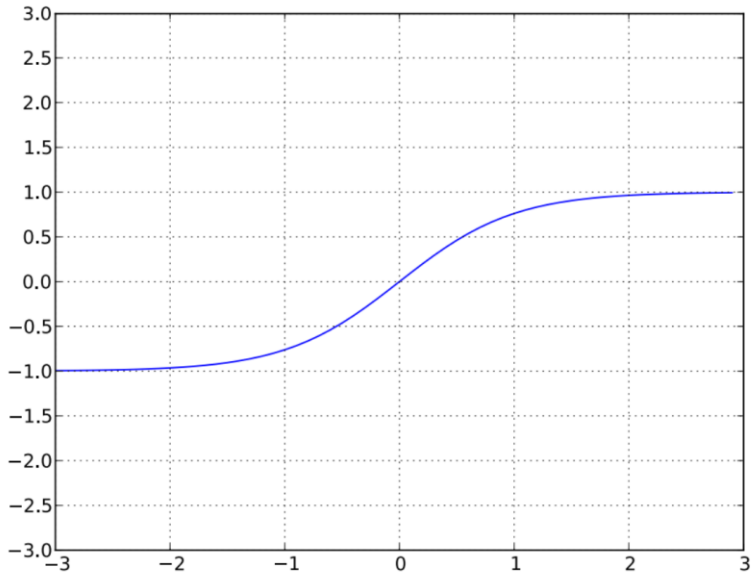
Linear activation function

$$g(a) = a$$
$$g'(a) = 1$$



Sigmoid activation function

$$g(a) = \frac{1}{1 + \exp(-a)}$$
$$g'(a) = g(a)(1 - g(a))$$



Tanh activation function

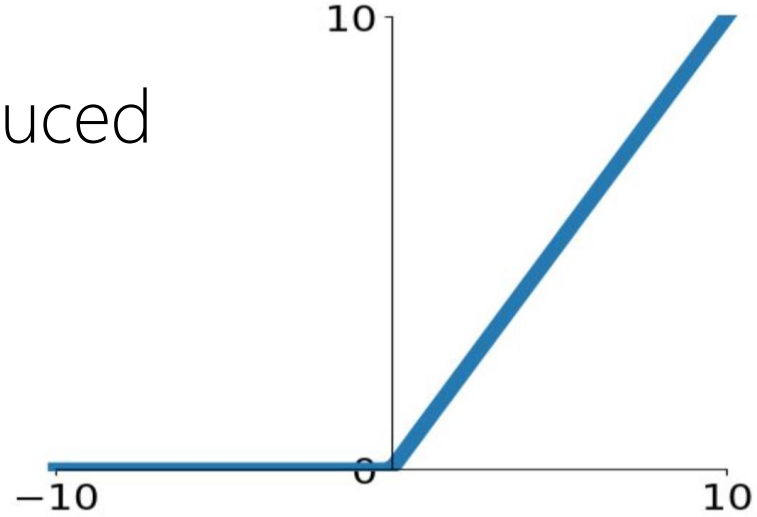
$$g(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)}$$
$$g'(a) = 1 - g(a)^2$$

Rectified Linear Unit

The ReLU activation function has been introduced

$$g(a) = ReLu(a) = \max(0, a)$$

$$g'(a) = 1_{a>0}$$



It has several advantages:

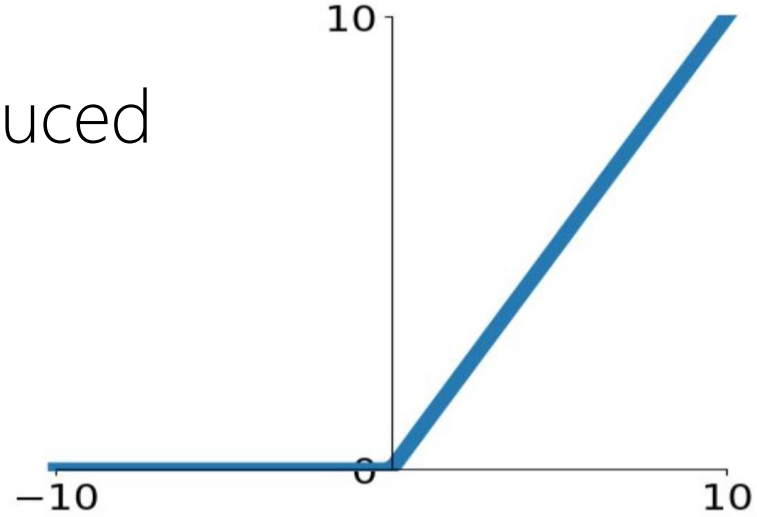
- Faster SGD Convergence (6x w.r.t sigmoid/tanh)
- Sparse activation (only part of hidden units are activated)
- Efficient gradient propagation (no vanishing or exploding gradient problems), and Efficient computation (just thresholding at zero)
- Scale-invariant: $\max(0, ax) = a \max(0, x)$

Rectified Linear Unit

The ReLU activation function has been introduced

$$g(a) = ReLu(a) = \max(0, a)$$

$$g'^{(a)} = 1_{a>0}$$



It has potential disadvantages:

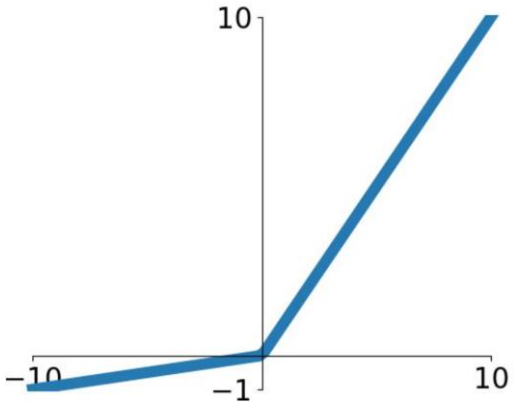
- Non-differentiable at zero: however it is differentiable
- Non-zero centered output
- Unbounded: Could potentially blow up
- Dying Neurons: ReLU neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. No gradients flow backward through the neuron, and so the neuron becomes stuck and "dies".

Decreased model capacity, it happens with high learning rates

Rectified Linear Unit (Variants)

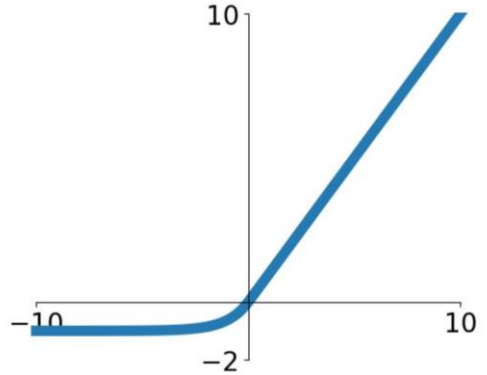
Leaky ReLU: fix for the "dying ReLU" problem

$$f(x) = \begin{cases} x & if\ x \geq 0 \\ 0.01x & otherwise \end{cases}$$



ELU: try to make the mean activations closer to zero which speeds up learning. Alpha is tuned by hand*by hand*

$$f(x) = \begin{cases} x & if\ x \geq 0 \\ \alpha(e^x - 1) & otherwise \end{cases}$$



Weights Initialization

The final result of gradient descent is affected by weight initialization:

- Zeros: it does not work! All gradient would be zero, no learning will happen
- Big Numbers: bad idea, if unlucky might take very long to converge
- $w \sim N(0, \sigma^2 = 0.01)$: good for small networks, but it might be a problem for deeper neural networks

In deep networks:

- If weights start too small, then gradient shrinks as it passes through each layer
- If the weights in a network start too large, then gradient grows as it passes through each layer until it's too massive to be useful

Some proposal to solve this Xavier initialization or He initialization …

Xavier Initialization

Suppose we have an input $\boldsymbol{x}$ with $\boldsymbol{I}$ components and a linear neuron with random weights $\boldsymbol{w}$. Its output is

$$h_j = w_{j1}x_1 + \cdots + w_{ji}x_I + \cdots + w_{jI}\,x_I$$

We can derive that $\boldsymbol{w_{ji}x_i}$ is going to have variance

$$Var(w_{ji}x_i) = E[x_i]^2 Var(w_{ji}) + E[w_{ji}]^2 Var(x_i) + Var(w_{ji})Var(x_i)$$

Now if our inputs and weights both have mean 0, that simplifies to

$$Var(w_{ji}x_i) = Var(w_{ji})Var(x_i)$$

If we assume all $\boldsymbol{w_i}$ and $\boldsymbol{x_i}$ are i.i.d. we obtain

$$Var(h_j) = Var(w_{j1}x_1 + \cdots + w_{ji}x_I + \cdots + w_{jI}\,x_I) = I * Var(w_i)Var(x_i)$$

Variance of output is the variance of the input but scaled by $\mathbf{I} * \boldsymbol{Var(w_i)}$.

Xavier Initialization

If we want the variance of the input and the output to

$$I * Var(w_j) = 1$$

Linear assumption seem too much, but in practice it works!

For this reason Xavier proposes to initialize $\boldsymbol{w} \sim \boldsymbol{N}\left(\mathbf{0}, \frac{\mathbf{1}}{\boldsymbol{n_{in}}}\right)$

Performing similar reasoning for the gradient Glorot & Bengio found

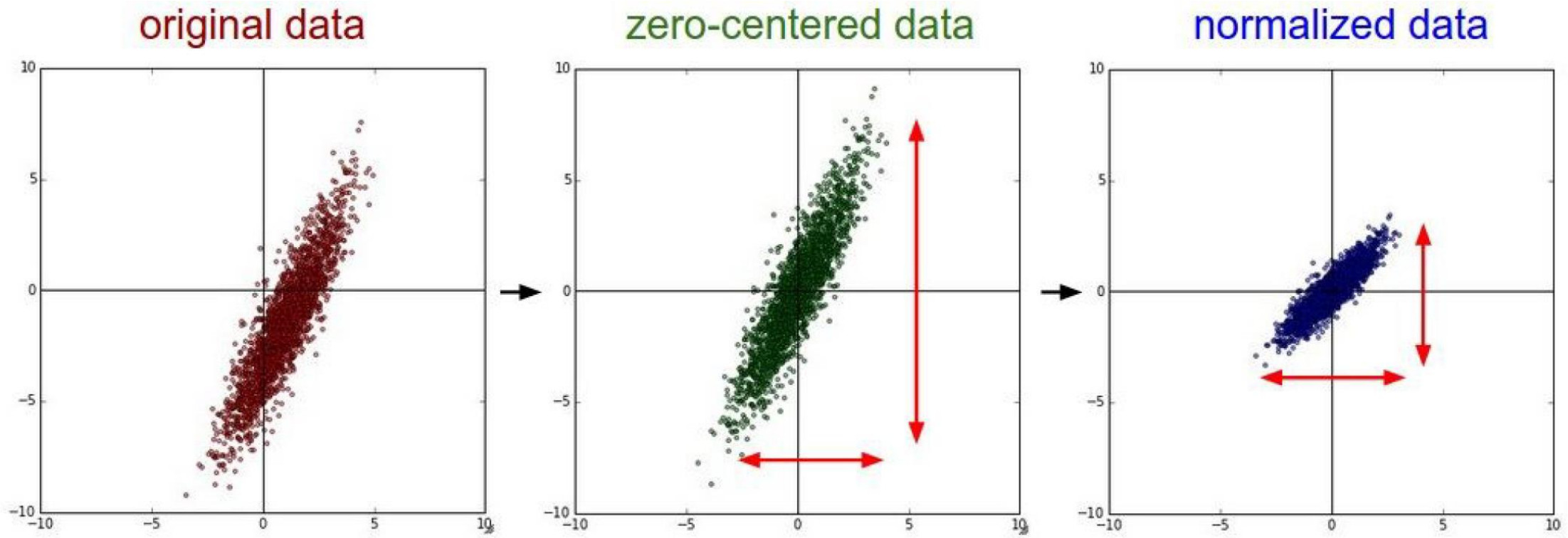
$$n_{out} Var(w_j) = 1$$

To accommodate for this and Xavier propose $\boldsymbol{w} \sim \boldsymbol{N}\left(\mathbf{0}, \frac{\mathbf{2}}{\boldsymbol{n_{in}+n_{out}}}\right)$

More recently He proposed, for rectified linear units, $\boldsymbol{w} \sim \boldsymbol{N}\left(\mathbf{0}, \frac{\mathbf{2}}{\boldsymbol{n_{in}}}\right)$

Batch Normalization

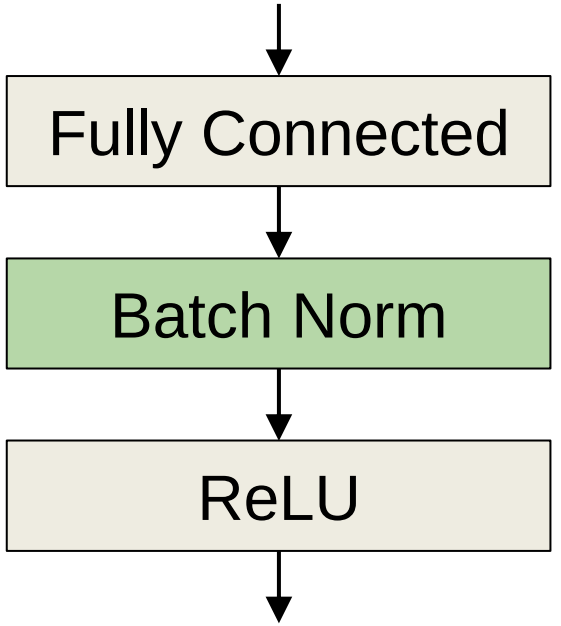
Networks converge faster if inputs have been whitened (zero mean, unit variances) and are uncorrelated to account for *covariate shift*.



Batch Normalization

Networks converge faster if inputs have been whitened (zero mean, unit variances) and are uncorrelated to account for *covariate shift*.

We can have internal covariate shift; normalization could be useful also at the level of hidden layers.



Batch normalization is a technique to cope with this:

- Forces activations to take values on a unit Gaussian at the beginning of the training
- Adds a BatchNorm layer after fully connected layers (or convolutional layers), and before nonlinearities.
- Can be interpreted as doing preprocessing at every layer of the network, but integrated into the network itself in a differentiable way.

Batch Normalization

In practice

- Each unit's pre-activation is normalized (mean subtraction, stddev division)
- During training, mean and stddev are computed for each minibatch
- Backpropagation takes into account normalization
- At test time, the global mean / stddev are used (global statistics are estimated using training running averages)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Simple Linear operation!
So it can be back-propagated

Apply a linear transformation, to squash the range, so that the network can decide (learn) how much normalization needs.

Can also learn $\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$
to recover the $\beta^{(k)} = \mathrm{E}[x^{(k)}]$
Identity mapping

Batch Normalization

Has shown to

- Improve gradient flow through the network
- Allow using higher learning rates (faster learning)
- Reduce the strong dependence on weights initialization
- Act as a form of regularization slightly reducing the need for dropout

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

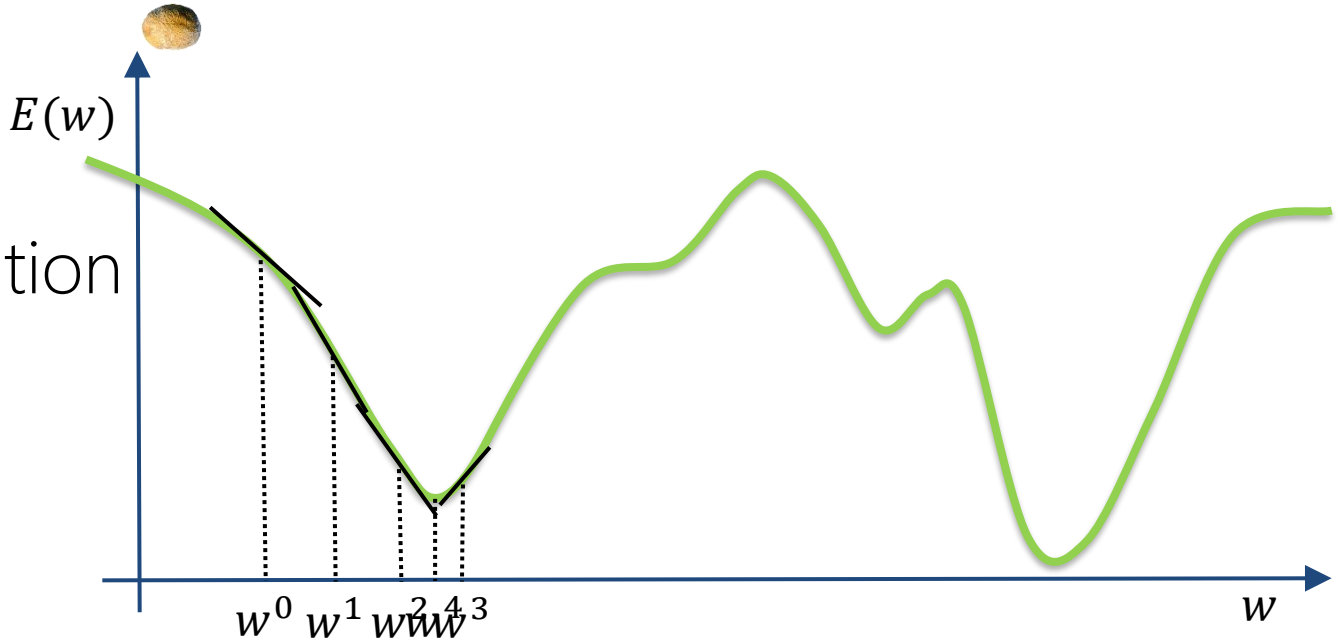
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Recall about Backpropagation

Finding weighs of a Neural Network is a non linear minimization process

$$argmin_w \; E(w) = \sum_{n=1}^{N} (t_n - g(x_n, w))^2$$



We iterate from a initial configuration

$$w^{k+1} = w^k - \eta \left. \frac{\partial E(w)}{\partial w} \right|_{w^k}$$

To avoid local minima can use momentum

$$w^{k+1} = w^k - \eta \left. \frac{\partial E(w)}{\partial w} \right|_{w^k} - \alpha \left. \frac{\partial E(w)}{\partial w} \right|_{w^{k-1}}$$

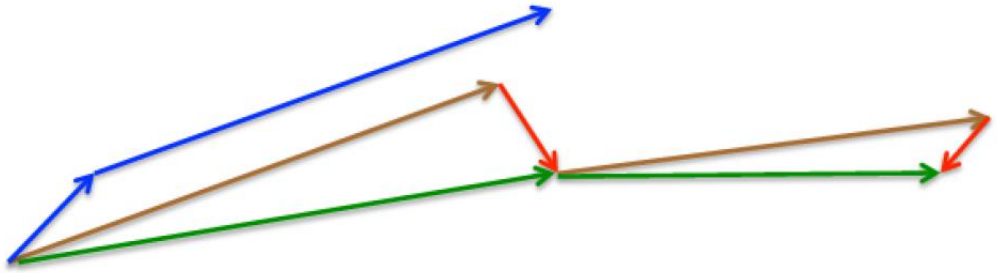
Several variations exists beside these two …

More about Gradient Descent

Nesterov Accelerated gradient: make a jump as momentum, then adjust

$$w^{k+\frac{1}{2}} = w^k - \alpha \left.\frac{\partial E(w)}{\partial w}\right|_{w^{k-1}}$$

$$w^{k+1} = w^k - \eta \left.\frac{\partial E(w)}{\partial w}\right|_{w^{k+\frac{1}{2}}}$$



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Adaptive Learning Rates

Neurons in each layer learn differently

- Gradient magnitudes vary across layers
- Early layers get "vanishing gradients"
- Should ideally use separate adaptive learning rates

Several algorithm proposed:

- Resilient Propagation (Rprop – Riedmiller and Braun 1993)
- Adaptive Gradient (AdaGrad – Duchi et al. 2010)
- RMSprop (SGD + Rprop – Teileman and Hinton 2012)
- AdaDelta (Zeiler et at. 2012)
- Adam (Kingma and Ba, 2012)
- …

Learning Rate Matters