DEEP LEARNING

Parameters and Hyperparameters Optimization in Deep Neural Networks

Adrian Horzyk

horzyk@agh.edu.pl

AGH University of Science and Technology Krakow, Poland

Parameters vs. Hyperparameters

Parameters in DNN are:

- weights, biases and other variables of the model that are updated and adjusted during the training process according to the chosen training algorithm.

Hyperparameters in DNN:

- are all variables and parameters of the model that are not adjusted by the training algorithm but by the DNN developer;
- are all parameters that can be changed independently of the way how the training algorithm works;
- can be adjusted by extra supporting algorithms like genetic or evolutional ones;
- number or layers, number of neurons in hidden layers,
- activation functions and types of used layers , and weights initialization
- learning rate, regularization and optimization parameters,
- augmenting and normalizing training and testing (dev) data,
- dropout and other optimization techniques,
- avoiding vanishing and exploding gradients.

Distribution of Training and Testing Data

Training and testing data should be of the same distribution(s):

If we use, e.g. images from different sources to train Convolutional Neural Networks, we must take care about the suitable division of the data from each distribution to the training and testing data. On the other hand, we don't be able to adjust the model and achieve high performance and generalization property.

During the training process, we usually use:

Training examples (training set) for adjusting the model

Verifying examples (def set) for checking the training progress

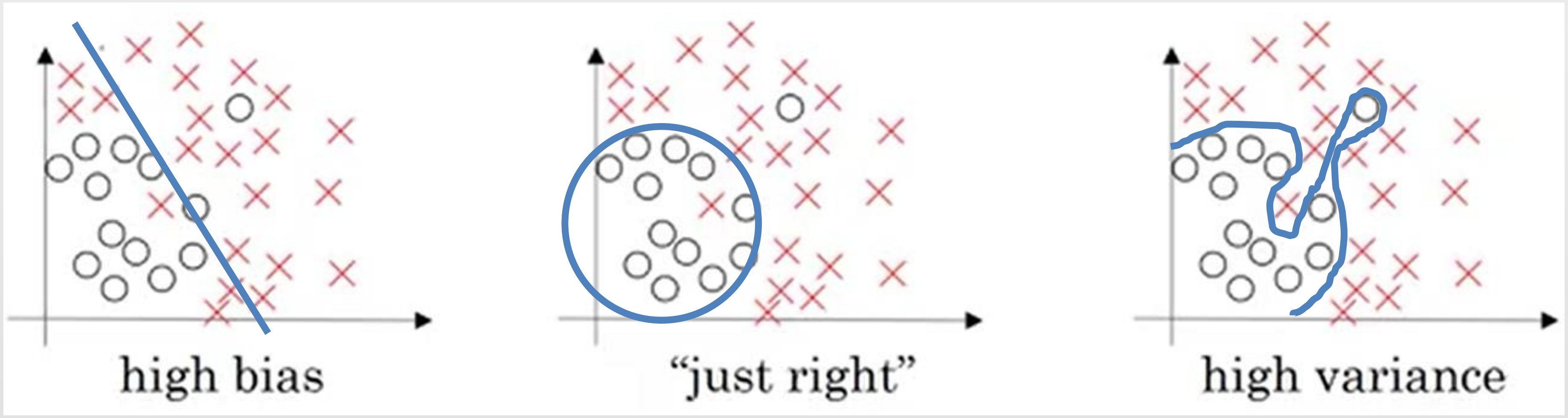
Test examples for checking generalization of the trained model

Sometimes, we don't use test examples, only checking the model during its adaptation and adjustment process.

Bias and Variance of the Model

When adapting the parameters of the model we can:

- Not enough model the training dataset (underfitting)
- Adjust the model too much, not achieving good generalization (overfitting)
- Fit the dataset adequately (right fitting)



Example Results	Example 1	Example 2	Example 3	Example 4
Train set error	1%	12%	12%	1%
Dev set error	12%	13%	20%	2%
Bias	low	high	high	low
Variance	high	low	high	low
Overfitting	yes	no	yes	no
Underfitting	no	yes	yes	no

Dependently on high bias and/or high variance, we can try to change/adjust different hyperparameters in the model to lower them appropriately and achieve better performance of the final model.

Tackling with high bias and variance

When we achieve high bias (low training data performance), try to:

- Create/use bigger network structure,
- Train the model longer,
- Use different neural network architecture (e.g. CNN, RNN), different layers,
- Change training rate, change activation functions, optimization parameters,
- Use an appropriate loss function not to stuck in local minima,
- …

When we achieve high variance (low dev data performance), try to:

- Use more training data with better distribution over the input and output data space.
- Try to use regularization (like dropout),
- Use different neural network architecture (e.g. CNN, RNN), different layers,
- Check the data distribution between training and dev sets,
- …

Human Level Performance

Human Level Performance:

- Is the classification/prediction error achieved by the committee of highly expertise humans (e.g. surgeons, psychologists, teachers, engineers).
- Is treated as a high bound and goal of training the model.
- Can be sometimes exceeded by machines and retrospectively checked by human experts.

Regularization

Regularization means the addition of the regularization factor and parameter λ to the loss function:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(a^{(i)}, y^{(i)}\right) + \frac{\lambda}{2\cdot m}\cdot\sum_{i=1}^{m}\left\|w^{[l]}\right\|_F^2$$

where we usually use Frobenius norm:

$$\left\|w^{[l]}\right\|_F^2 = \sqrt{\sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}\left(w_{i,j}^{[l]}\right)^2}$$

$$w^{[l]} := w^{[l]} - \alpha\cdot dJw^{[l]} - \frac{\alpha\cdot\lambda}{m}\cdot w^{[l]} = w^{[l]} - \alpha\frac{\partial J\left(w^{[l]},b\right)}{\partial w^{[l]}} - \frac{\alpha\cdot\lambda}{m}\cdot w^{[l]}$$

$$dJw^{[l]} = \frac{\partial J\left(w^{[l]},b\right)}{\partial w^{[l]}} = \frac{1}{m}X\cdot dJZ^T + \frac{\lambda}{m}\cdot w^{[l]}$$

This kind of regularization is often called the “weight decay”.

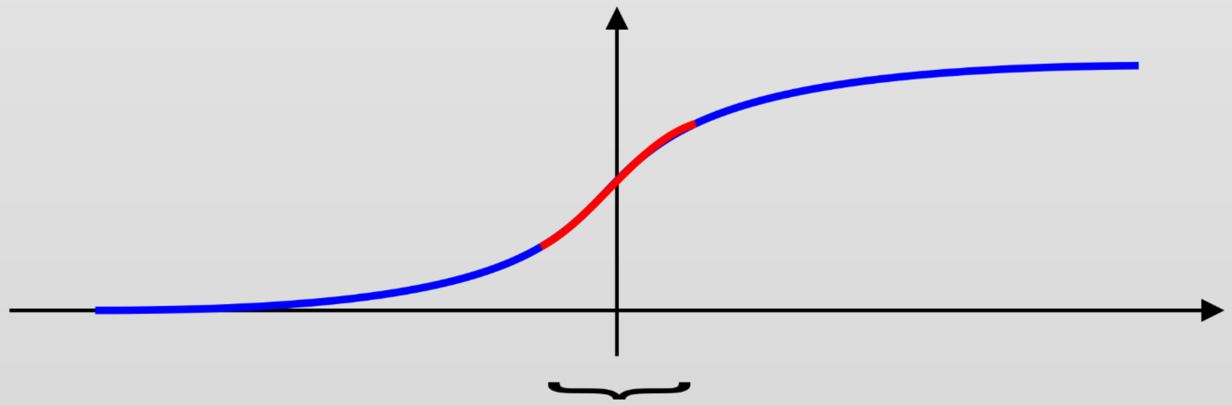
Regularization prevents overfitting

Regularization penalizes the weight matrices to be too large thanks to this extra regularization factor:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(a^{(i)}, y^{(i)}\right) + \frac{\lambda}{2\cdot m}\cdot\sum_{i=1}^{m}\left\|w^{[l]}\right\|_F^2$$

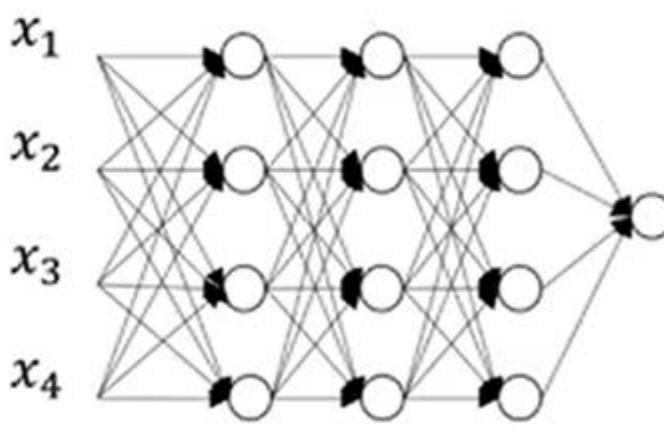
because we want to minimize the above cost function during the training! So the network will compose of nearly linear (not very complex) functions.

If the weights are small the output values of the activation functions of the neurons will also be not exceeding the middle, almost linear part of the activation function, so in case the activation function is nearly linear:



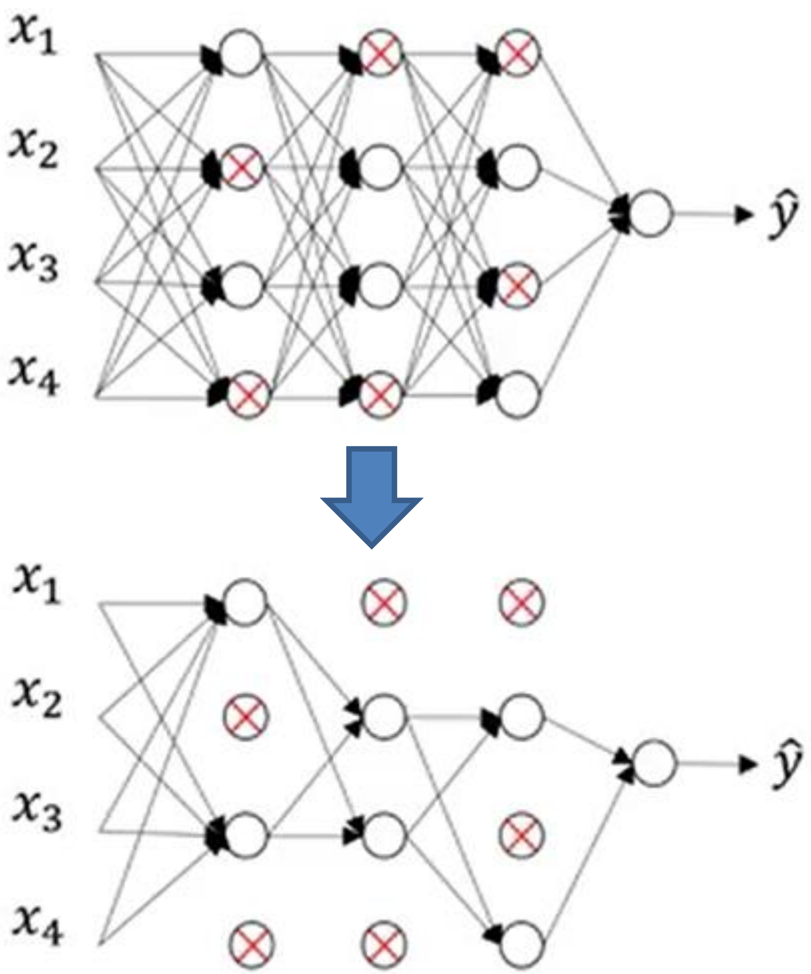
Dropout Regularization

Dropout regularization switches off some neurons with a given probability, not using them temporarily during propagation and backpropagation steps forcing the network to learn the same by various combinations of neurons in the network:



Dropout can be selectively used only in a selected subset of layers.

Dropout is usually used to layers with a big amount of weights and neurons.



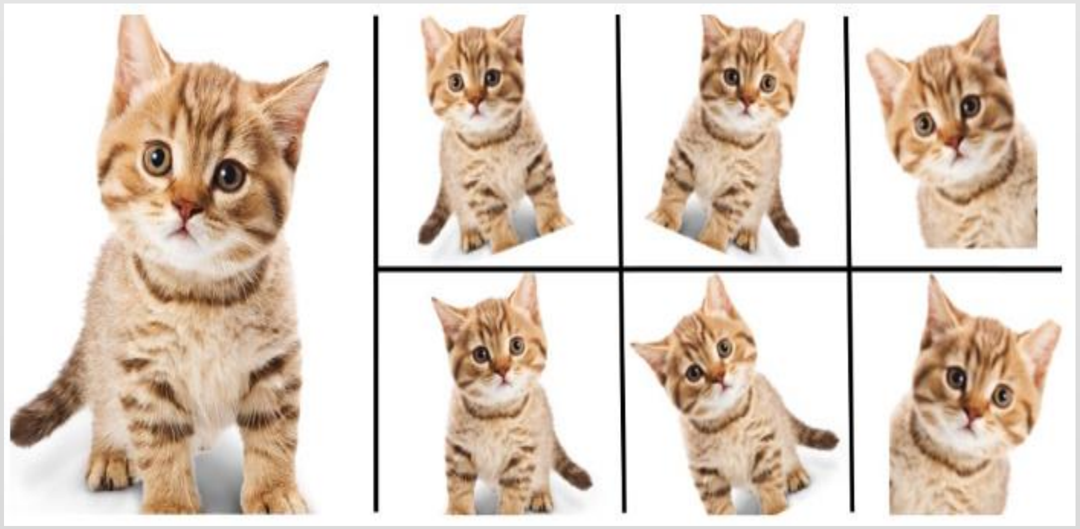
Implementing dropout regularization the input stimuli of neurons are weaken according to the number of the shut off neurons (i.e. the chosen probability of dropout on average, e.g. p = 0.25), so the stimulation must be higher to achieve the right stimulation of the neurons, e.g. the classification neurons in the last layer.

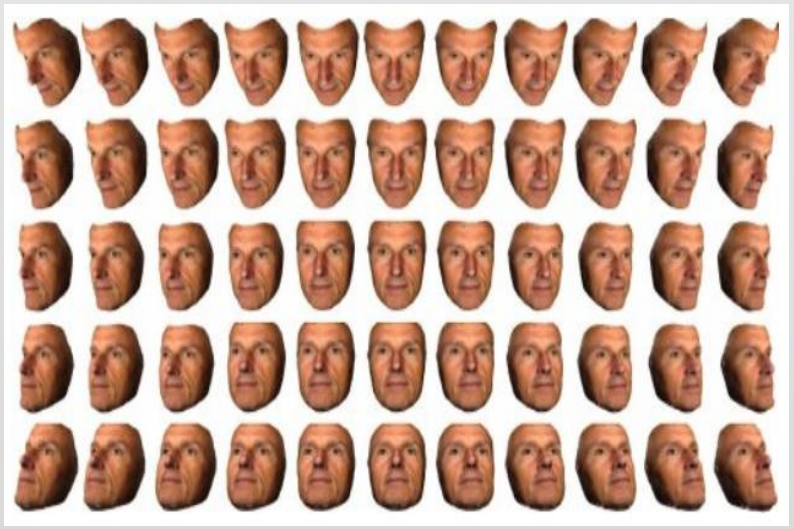
Data Augmentation

We can also augment training dataset to avoid the known limitations of the neural structures and learning algorithms to deal with rotated, scaled and moved patterns in the input data space. Therefore, we rotate, scale, and move pattern and thus augment the training data space by these variations of training data. This techniques usually allows to achieve better training results:

- Rotate
- Scale
- Cut (different parts of images)
- Move

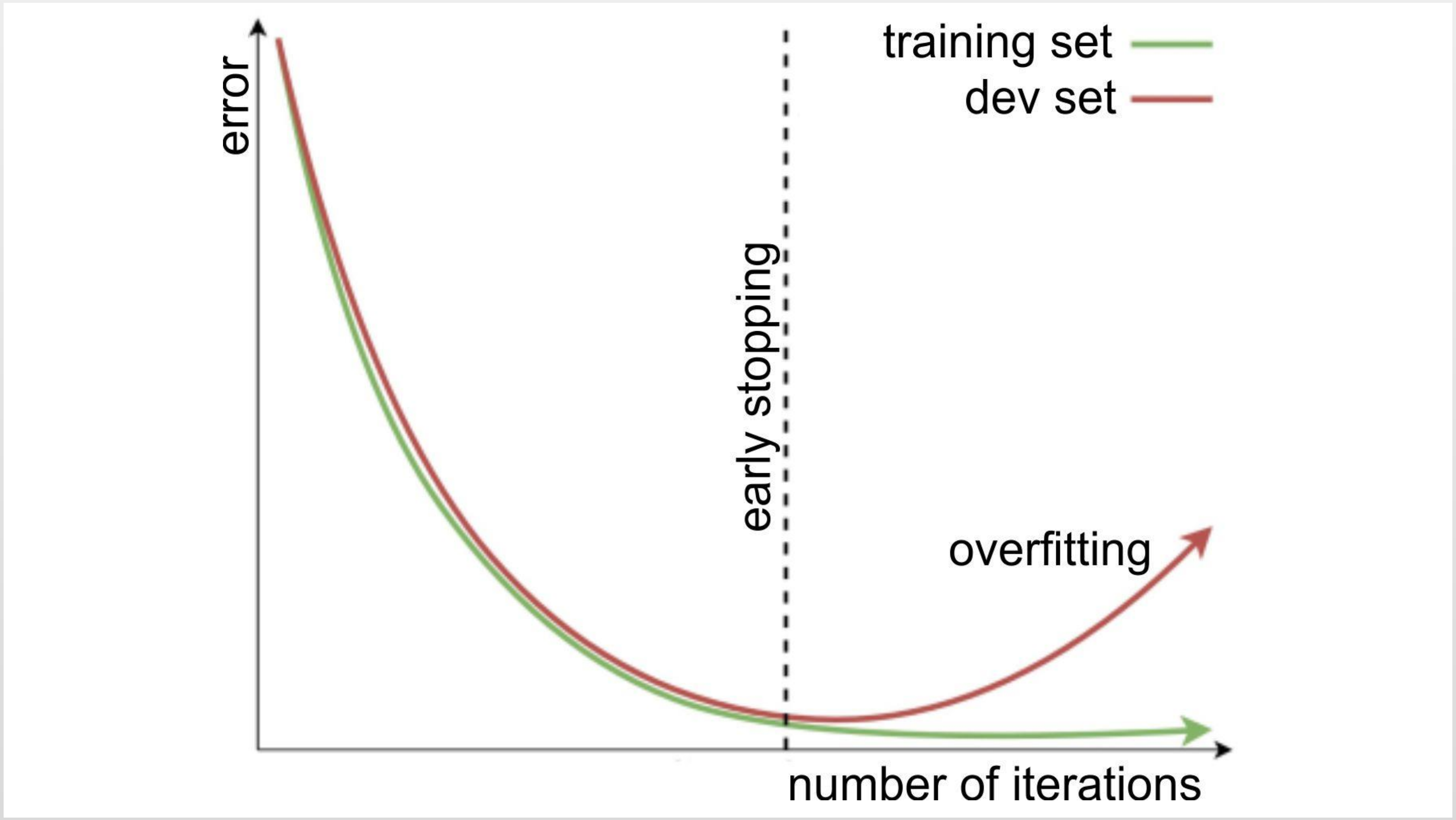
EXPLORE OKLAHOMA
2018
ABC - 123
TRAVELOK.COM





Early Stopping

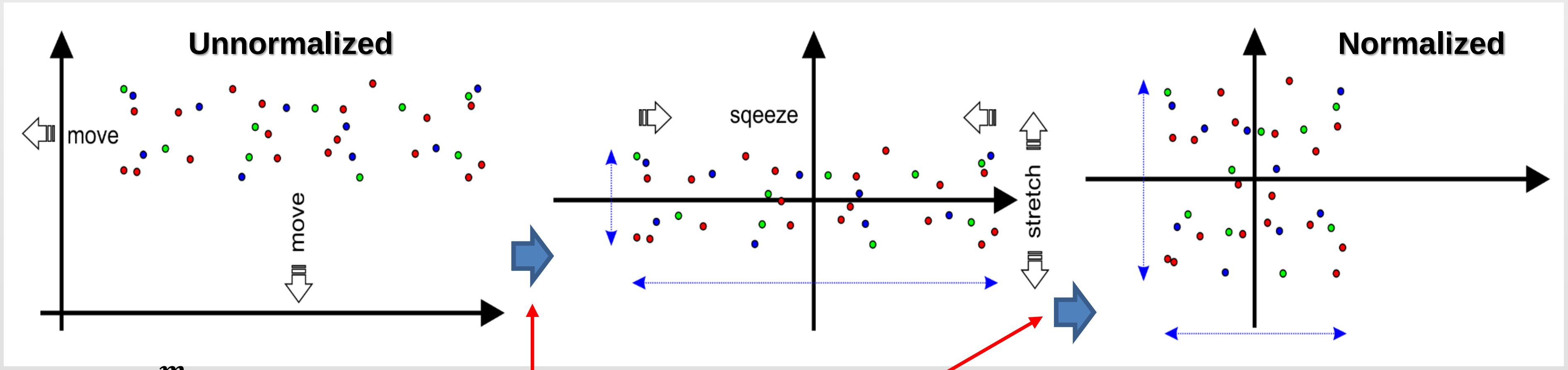
We can also use "early stopping" of the training routine before the error on the dev set starts to grow:



Normalizing Training Sets

Normalization:

- Makes data of different attributes (different ranges) comparable and not favourited or neglected during the training process. Therefore, we scale all training and testing (dev) data inside the same normalized ranges.
- We also must not forget to scale testing (dev) data using the same μ and σ^2.

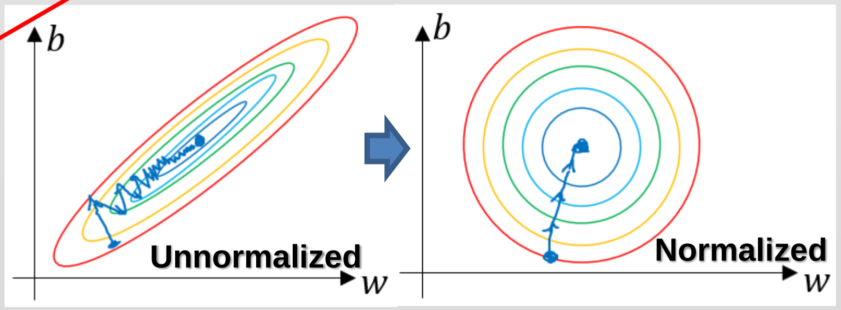


$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$x := x - \mu$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} *_{elementwise}\, x^{(i)}$$

$$x := x/\sigma$$

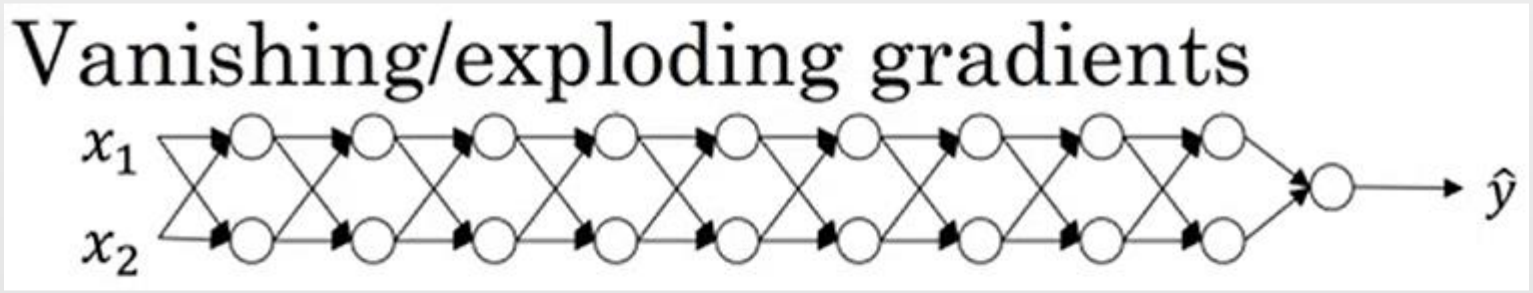


- The training process is faster and better when training data are normalized!

Vanishing and Exploding Gradients

In deep structures, computed gradients in previous layers are:

- smaller and smaller (vanish) when a values lower than 1 are multiplied/squared
- greater and grater (explode) when a values bigger than 1 are multiplied /squared

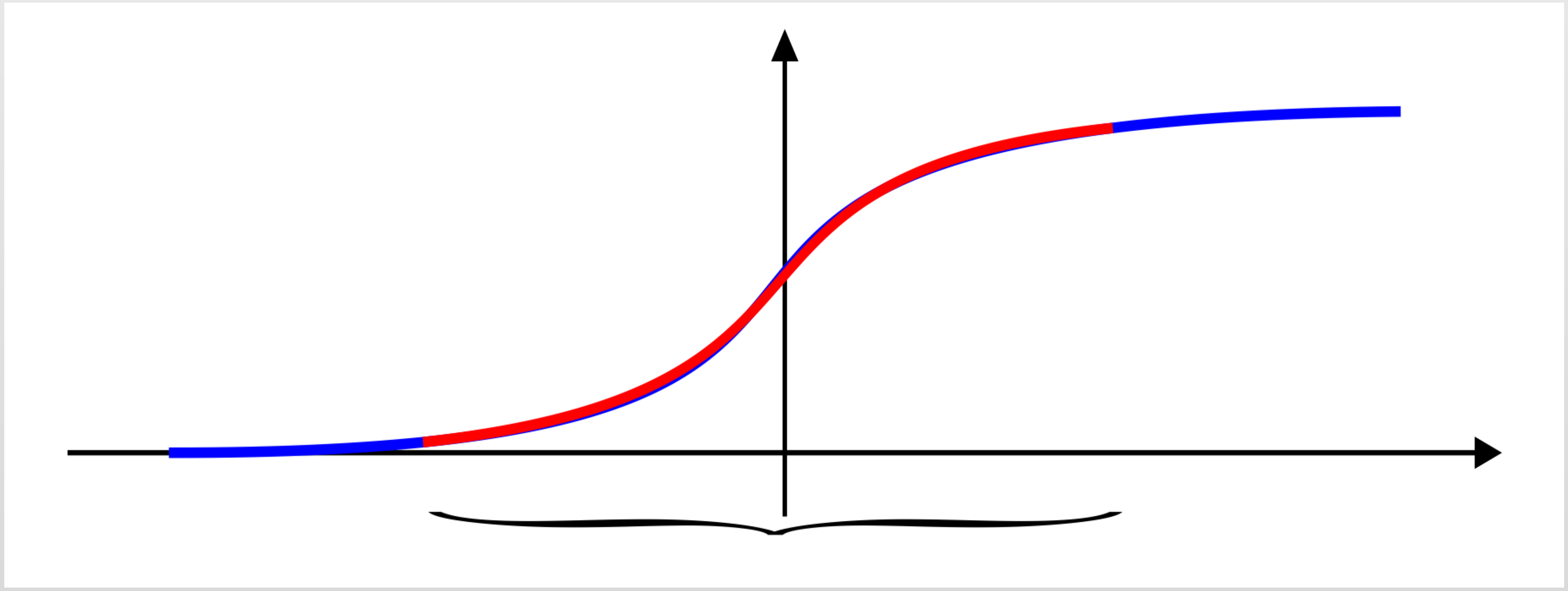


because today we use deep neural networks that consist of tens of layers!

Weights Initialization

We initialize weights with small values:

- to put the values of activation functions in the range of the largest variance, which speed up the training process.
- taking into account the number neurons $n^{[l-1]}$ of the previous layer, e.g. for tanh: $\sqrt{\frac{1}{n^{[l-1]}}}$ (popular Xavier initialization) or $\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$, multiplying the random numbers from the range of 0 and 1 by such a factor.



Let’s start to change hyperparameters!

- ✓ Improving performance of the training
- ✓ Speeding up the training process
- ✓ Not stacking in local minima
- ✓ Using less computational resources to get the model

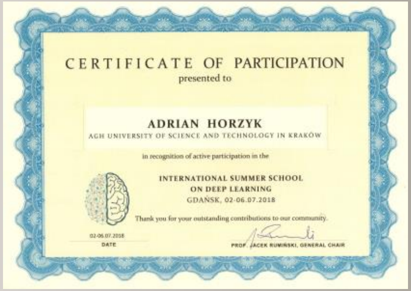
Bibliography and Literature

1. Nikola K. Kasabov, *Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence*, In Springer Series on Bio- and Neurosystems, Vol 7., Springer, 2019.
2. Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, MIT Press, 2016, ISBN 978-1-59327-741-3 or PWN 2018.
3. Holk Cruse, *Neural Networks as Cybernetic Systems*, 2nd and revised edition
4. R. Rojas, *Neural Networks*, Springer-Verlag, Berlin, 1996.
5. *Convolutional Neural Network* (Stanford)
6. *Visualizing and Understanding Convolutional Networks*, Zeiler, Fergus, ECCV 2014
7. IBM: https://www.ibm.com/developerworks/library/ba-data-becomes-knowledge-1/index.html
8. NVIDIA: https://developer.nvidia.com/discover/convolutional-neural-network
9. JUPYTER: https://jupyter.org/

Adrian Horzyk

horzyk@agh.edu.pl

Google: Horzyk



CERTIFICATE OF PARTICIPATION
presented to

ADRIAN HORZYK
AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY IN KRAKÓW

in recognition of active participation in the

INTERNATIONAL SUMMER SCHOOL
ON DEEP LEARNING
GDAŃSK, 02-06.07.2018

Thank you for your outstanding contributions to our community.

02-06.07.2018
DATE

[illegible]
PROF. JACEK RUMIŃSKI, GENERAL CHAIR

University of Science and Technology in Krakow, Poland