COMPUTATIONAL INTELLIGENCE
DEEP LEARNING

Hyperparameters Initialization Regularization Optimization in Deep Neural Networks

Adrian Horzyk

horzyk@agh.edu.pl

AGH

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 and their parameters,
- 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

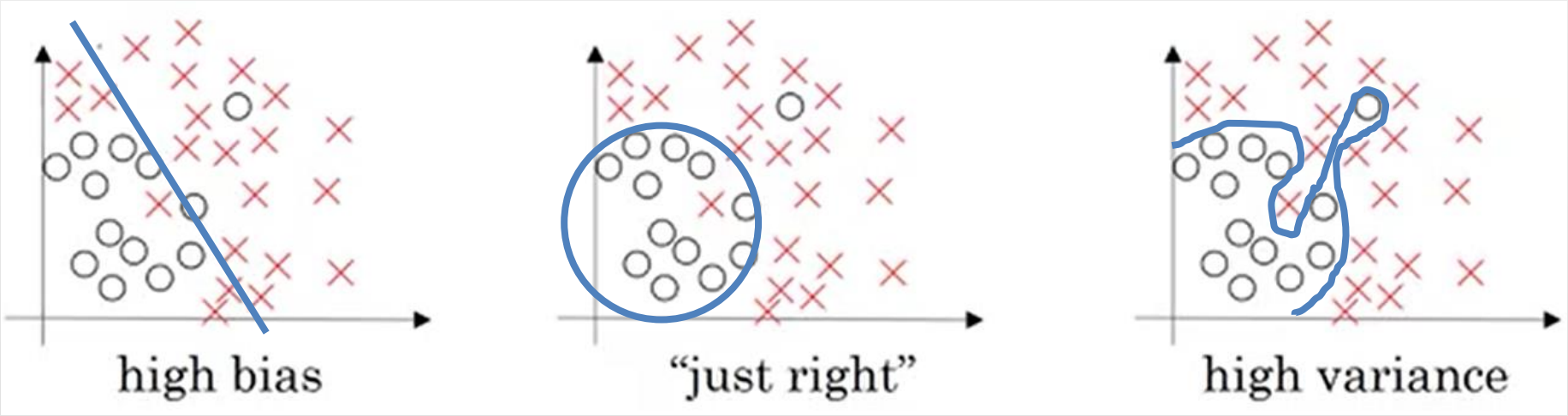
Test examples for checking generalization of the trained model

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 (e.g. use data augmentation),
- 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,
- Early stopping
- …

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.

We will try to achieve human-level performance, and when we do it, we will try to achieve a better performance than the human level one is!

The final performance that exceeds the human level one is unknown, so we generally do not know how much better the final performance might be because human experts cannot do it better.

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”.

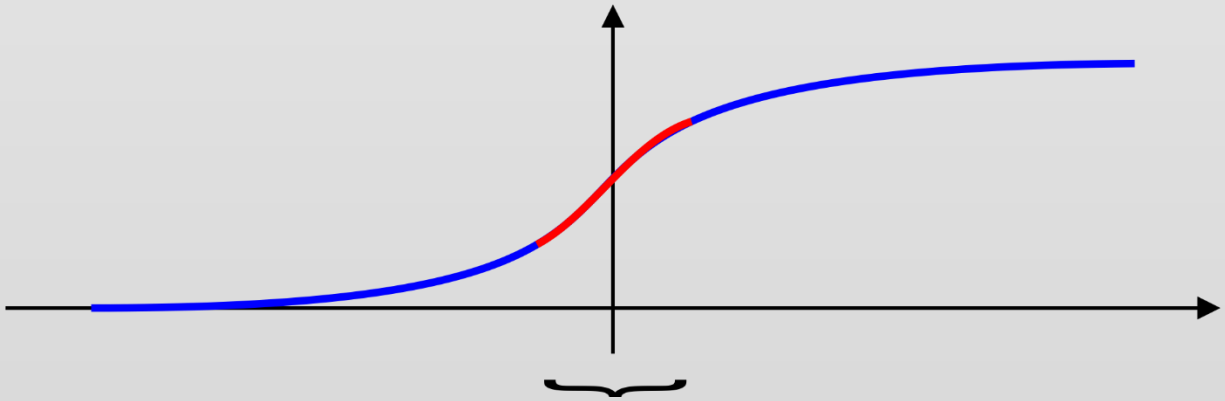
Regularization prevents overfitting

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

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

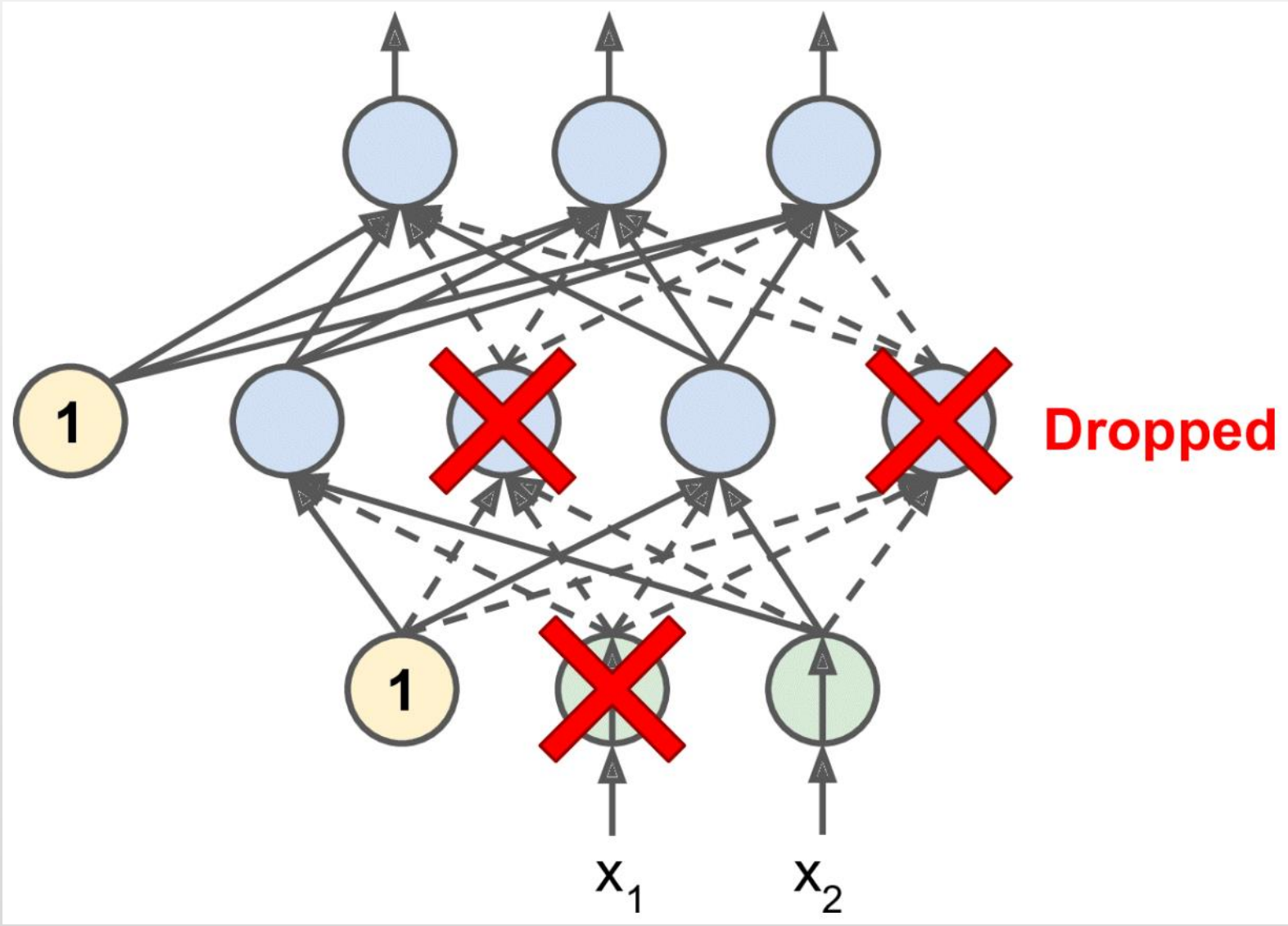
because we want to minimize the above cost function during the training! So the network will compose of nearly linear (not very complex) functions.

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:



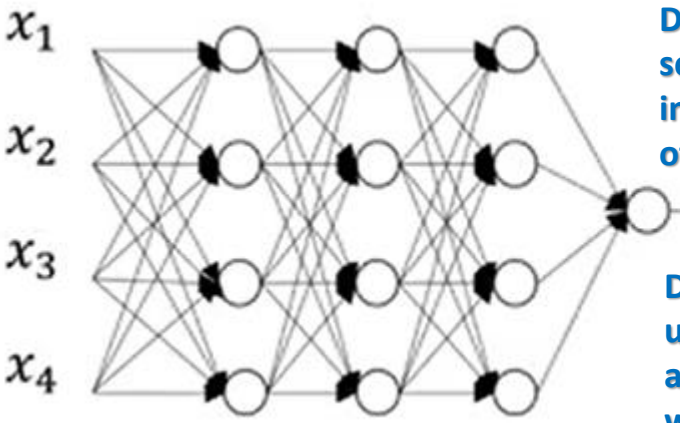
Dropout Regularization

Dropout is one of the most popular regularization techniques for deep neural nets.



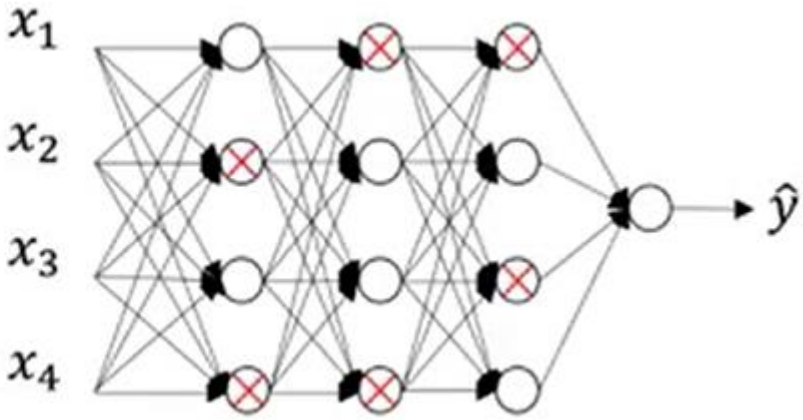
Dropout Regularization

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:

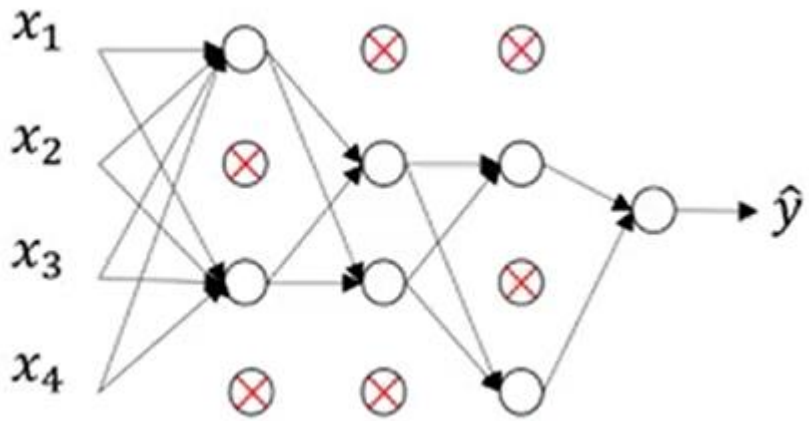


Dropout can be selectively used only in a selected subset of layers.

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



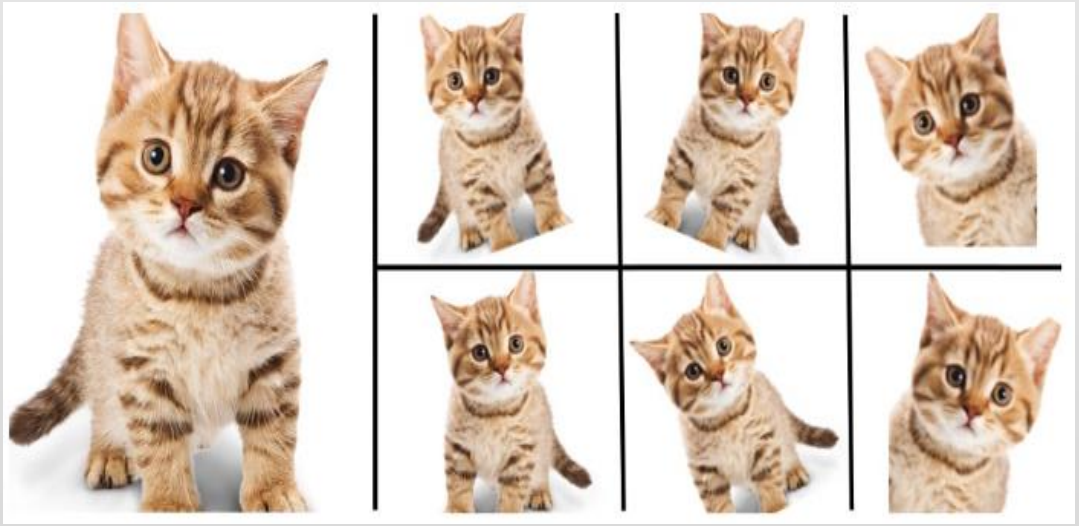
Implementing dropout regularization the input stimuli of neurons are weaken according to the number of the shut off neurons (i.e. the chosen probability of dropout on average, e.g. p = 0.25), so the stimulation must be higher to achieve the right stimulation of the neurons, e.g. the classification neurons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

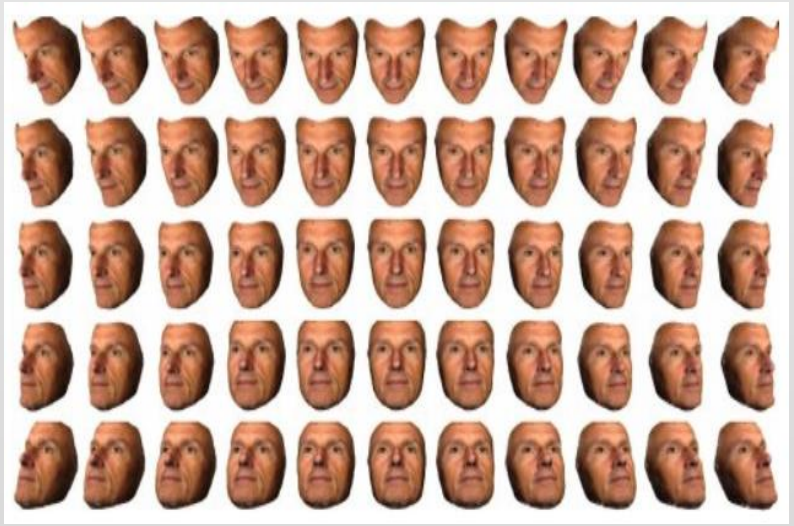


Data Augmentation

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:

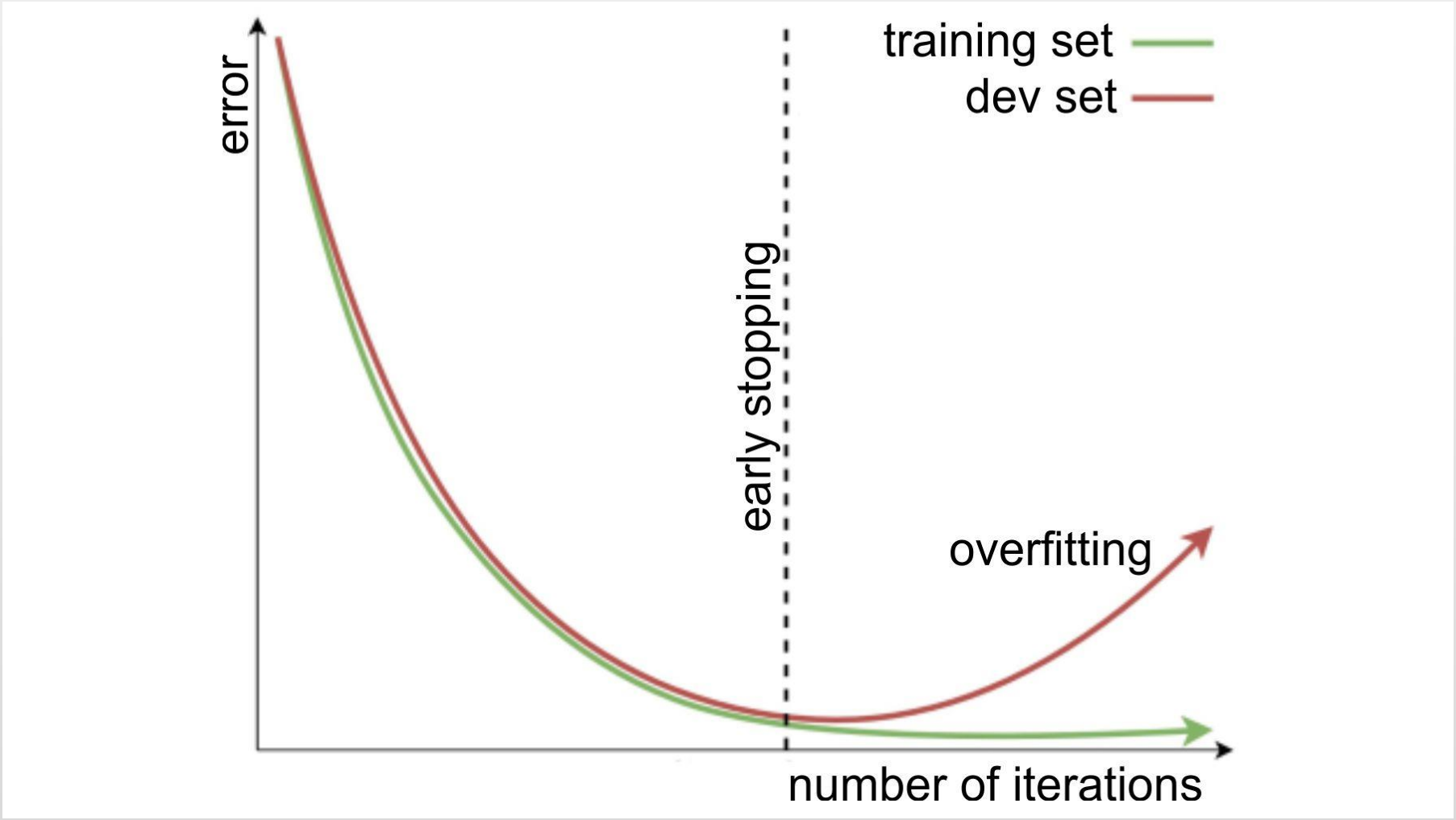
- Rotate or move
- Scale (zoom in or out)
- Cut (different parts of images)
- Flip (horizontally or vertically)
- Inverse or change colors





Early Stopping

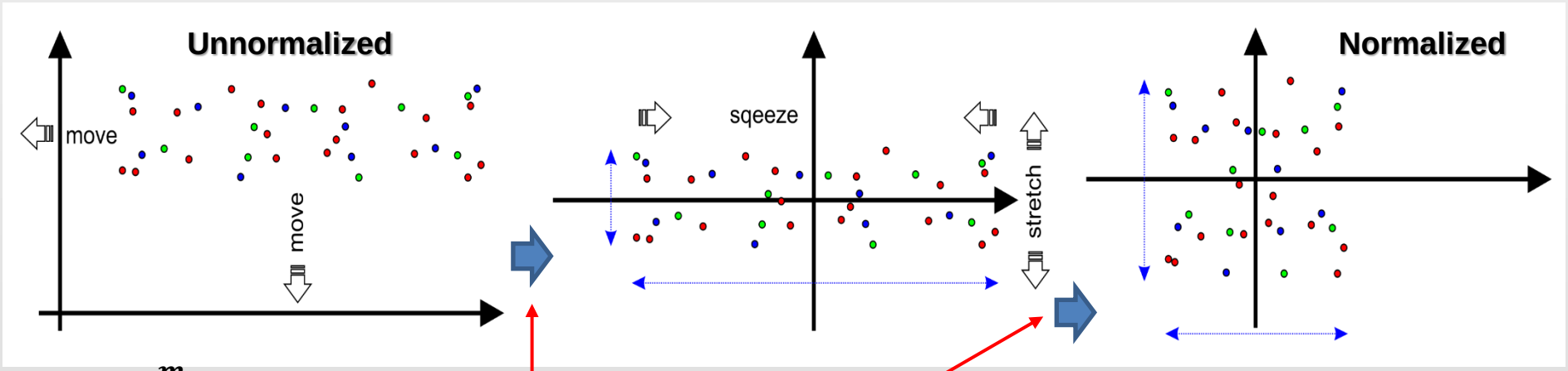
We can also use "early stopping" of the training routine before the error on the dev set starts to grow:



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.

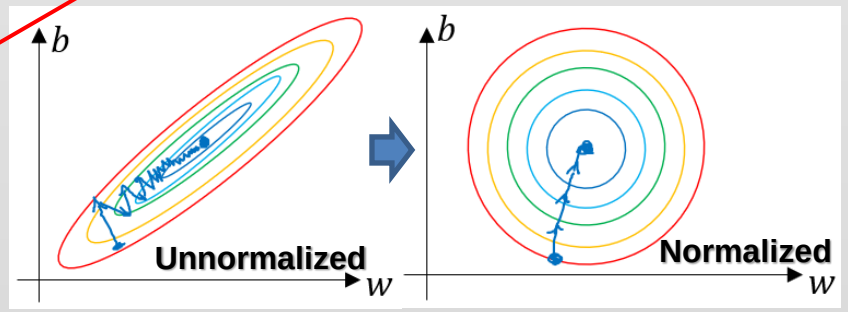


$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$x := x - \mu$$

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

$$x := x/\sigma$$

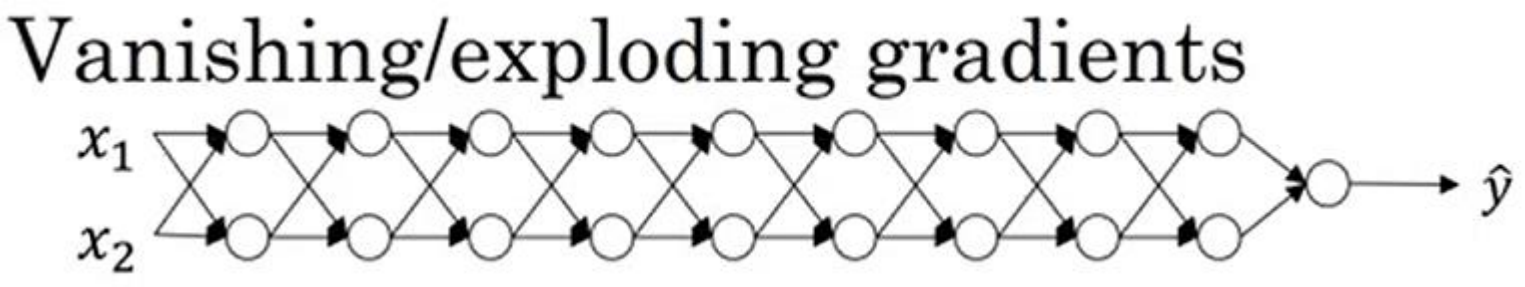


- The training process is faster and better when training data are normalized!

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 greater (explode) when a values bigger than 1 are multiplied /squared**



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

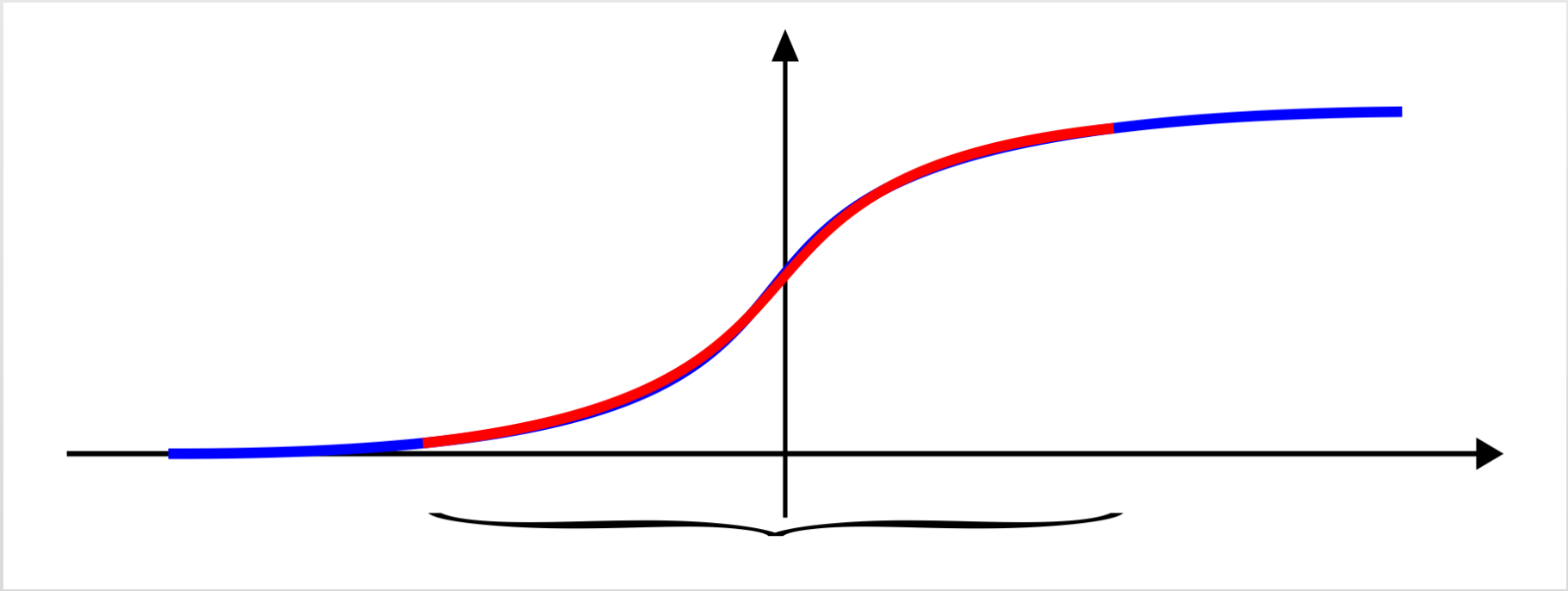
Weights Initialization

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.



On-line and Batch Training

When using a gradient descent algorithm, we have to decide after what number of presented training examples parameters (weights and biases) will be updated, and due to this number we define:

- **Stochastic (on-line) training – when we update parameters immediately after the presentation of each training example.**
  In this case, training process might be unstable.
- **Batch (off-line) training – when we update parameters only after the presentation of all training examples.**
  In this case, training process might take very long time and stuck in local minima or saddle points.
- **Mini-batch training – when we update parameters after the presentation of a subset of training examples consisting of a defined number of these examples.**
  In this case, training process is a compromise between the stability and speed, much better avoiding to stuck in local minima, so this option is recommended.
  If the number of examples is too small, the training process is more unstable.
  If the number of examples is too big, the training process is longer but more stable and robust.
  The mini-batch size is one of the hyperparameters of the model.

Mini-batches used in Deep Learning

Training examples are represented as a set of *m* pairs which are trained and update parameters one after another in on-line training (stochastic gradient descent):

$$(X, Y) = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$$

Hence, we can consider two big matrices storing input data X and output predictions Y, which can be presented and trained as one batch (batch gradient descent):

$$X = [x^{(1)}, x^{(2)}, x^{(3)}, \dots, x^{(1000)}, \dots, x^{(2000)}, \dots, x^{(3000)}, \dots, x^{(m)}]$$

$$Y = [y^{(1)}, y^{(2)}, y^{(3)}, \dots, y^{(1000)}, \dots, y^{(2000)}, \dots, y^{(3000)}, \dots, y^{(m)}]$$

Or we can divide them to mini-batches (mini-batch gradient descent) and update the network parameters after each mini-batch of training examples presentation:

$$X = [x^{(1)}, x^{(2)}, x^{(3)}, \dots, x^{(1000)} \mid x^{(1001)}, \dots, x^{(2000)} \mid x^{(2001)}, \dots, x^{(3000)} \mid x^{(3001)}, \dots, x^{(m)}]$$

$$X^{\{1\}} \qquad X^{\{2\}} \qquad X^{\{3\}} \qquad X^{\{m/batchsize\}}$$

$$Y = [y^{(1)}, y^{(2)}, y^{(3)}, \dots, y^{(1000)} \mid y^{(1001)}, \dots, y^{(2000)} \mid y^{(2001)}, \dots, y^{(3000)} \mid y^{(3001)}, \dots, y^{(m)}]$$

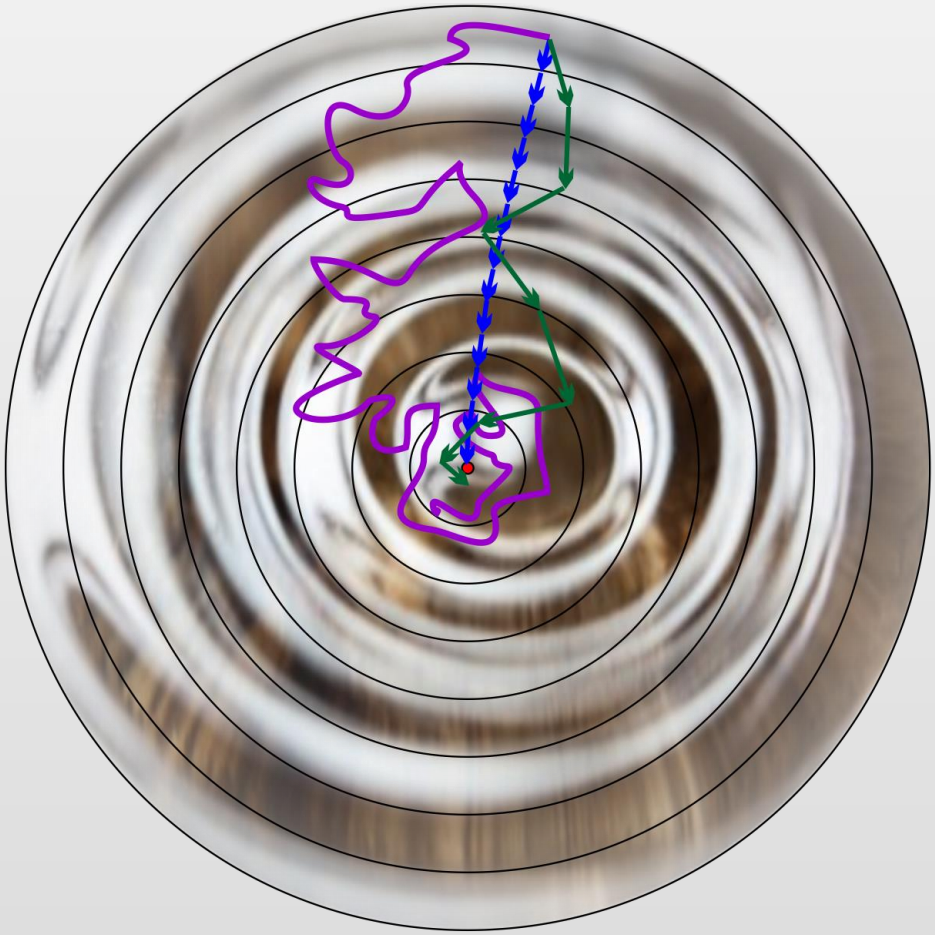
$$Y^{\{1\}} \qquad Y^{\{2\}} \qquad Y^{\{3\}} \qquad Y^{\{m/batchsize\}}$$

$$(X, Y) = \{(X^{\{1\}}, Y^{\{1\}}), (X^{\{2\}}, Y^{\{2\}}), \dots, (X^{\{m/batchsize\}}, Y^{\{m/batchsize\}})\}$$

If m= 20.000.000 training examples and the mini-batch size is 1000, we get 20.000 mini-batches (i.e. training steps for each full training dataset presentation, called training epoch), where T = $m/batchsize$.

In deep learning, we use mini-batches to speed up training and avoid stacking in saddle points.

Graphical Interpretation of Mini-batches



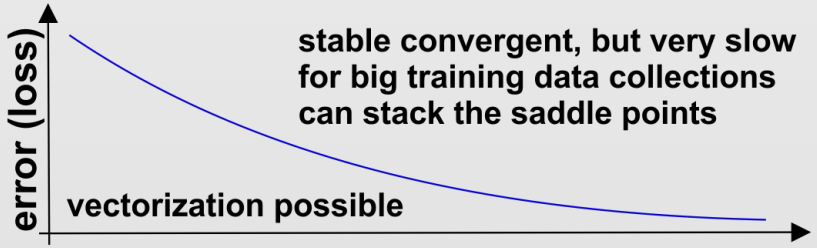
usually very large oscillations close to the minimum

almost no oscillations close to the minimum

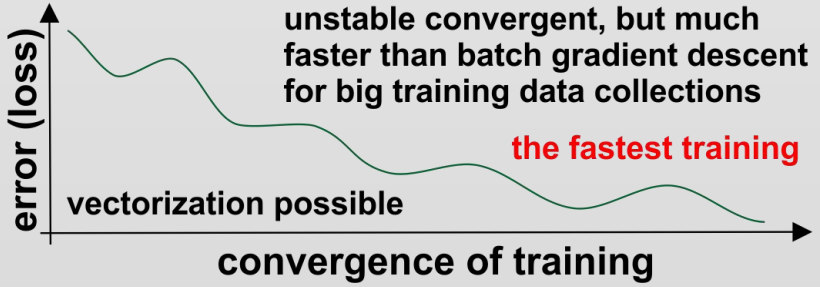
possible small oscillations close to the minimum

Stochastic Gradient Descent

Batch Gradient Descent



Mini-Batch Gradient Descent



Mini-batch Gradient Descent

To optimize computation speed, the mini-batch size (mbs) is usually set according to the number of parallel cores in the GPU unit, so it is typically any power of two:

- mbs = 32, 64, 128, 256, 512, 1024, or 2048

because then such mini-batches can be processed in one parallel step time-efficiently.

If mbs = m, we get Batch Gradient Descent typically used for small training dataset (a few thousands of training examples).

If mbs = 1, we get Stochastic Gradient Descent.

Therefore, instead of looping over every training example (like in stochastic training) or stacking all training examples into two big matrices X and Y,
we loop over the number of mini-batches, computing outputs, errors, gradients and updates of parameters (weights and biases):

One training epoch consists of T training steps over the mini-batches.

Mini-batches are used for big training dataset (ten or hundred thousands and millions of training examples) to accelerate computation speed.

repeat

$J = 1$

for $j = 1$ to n_x

$dJW_j = 0$

$dLB = 0$

for $t = 1$ to T

$Z^{\{t\}} = W^T X^{\{t\}} + B$

$A^{\{t\}} = \sigma(Z^{\{t\}})$

$J += -\left(Y^{\{t\}} \, log \, A^{\{t\}} + (1 - Y^{\{t\}}) log(1 - A^{\{t\}})\right)$

$dJZ^{\{t\}} = A^{\{t\}} - Y^{\{t\}}$

for $j = 1$ to n_x

$dJW_j += X_j^{\{t\}} \cdot dJZ^{\{t\}}$

$dJB += dJZ^{\{t\}}$

$J /= m$

for $j = 1$ to n_x

$dJW_j /= m$

$W_j -= \alpha \cdot dJW_j$

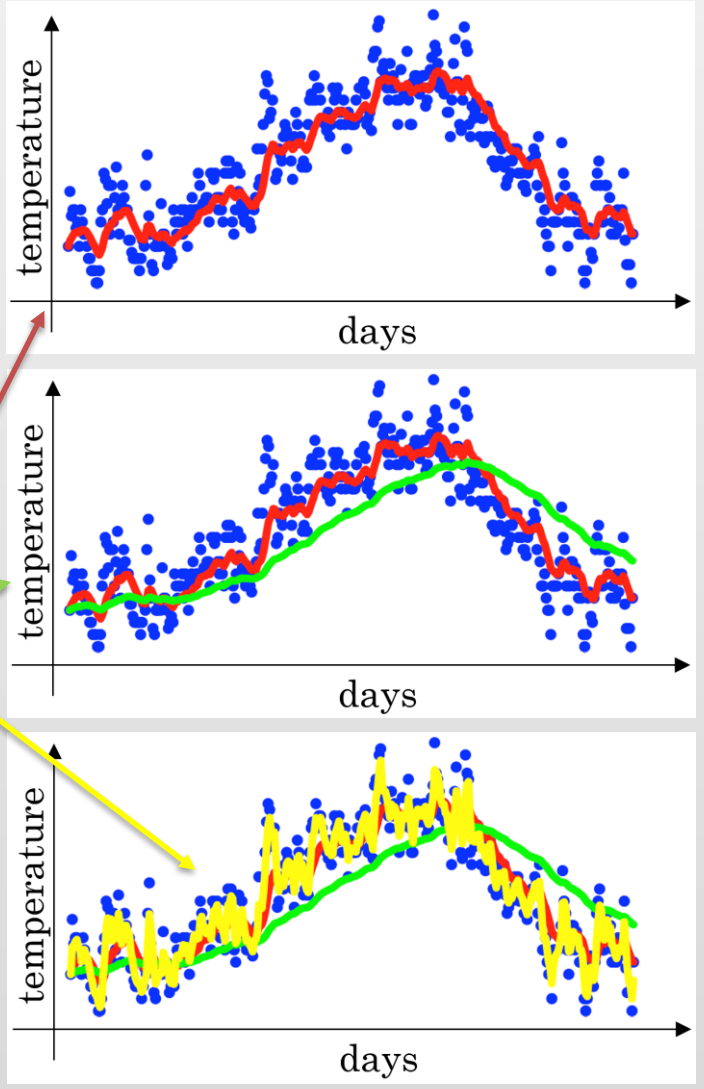
$dJB /= m$

$B -= \alpha \cdot dJB$

Exponentially Weighted Averages

Exponentially Weighted (Moving) Averages is another much faster optimization algorithm than Gradient Descent:

- **We compute weighted averages after the following formula:**
- $v_0 = 0$
- $v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot \theta_t$
- **where β controls the number or previous steps that control the current value v_t :**
- $\beta_{red} = 0.9$ (adapts taking into account 10 days)
- $\beta_{green} = 0.98$ (adapts slowly in view of 50 days)
- $\beta_{yellow} = 0.5$ (adapts quickly averaging 2 days)
- θ_t - is a currently measured value (temperature)



We can use this approach for optimization in deep neural networks.

Exponentially Weighted Averages

Why we call this algorithm Exponentially Weighted Averages:

When we substitute and develop the formula:

$$v_0 = 0$$
$$v_t = \beta \cdot v_{t-1} + (1-\beta) \cdot \theta_t$$

we get the following:

$$\begin{aligned} v_t &= \beta \cdot v_{t-1} + (1-\beta) \cdot \theta_t = \beta \cdot (\beta \cdot v_{t-2} + (1-\beta) \cdot \theta_{t-1}) + (1-\beta) \cdot \theta_t = \\ &= \beta \cdot (\beta \cdot (\beta \cdot v_{t-3} + (1-\beta) \cdot \theta_{t-2}) + (1-\beta) \cdot \theta_{t-1}) + (1-\beta) \cdot \theta_t = \\ &= (1-\beta)\left[\beta^0 \cdot \theta_t + \beta^1 \cdot \theta_{t-1} + \beta^2 \cdot \theta_{t-2} + \beta^3 \cdot \theta_{t-3} + \beta^4 \cdot \theta_{t-4} + \cdots\right] \end{aligned}$$

and when we now substitute $\beta = 0.9$ we get the weighted average by the exponents of the β value:

$$\begin{aligned} v_t &= (1-0.9)\left[\theta_t + 0.9 \cdot \theta_{t-1} + 0.9^2 \cdot \theta_{t-2} + 0.9^3 \cdot \theta_{t-3} + 0.9^4 \cdot \theta_{t-4} + \cdots\right] = \\ &= \frac{\theta_t + 0.9 \cdot \theta_{t-1} + 0.9^2 \cdot \theta_{t-2} + 0.9^3 \cdot \theta_{t-3} + 0.9^4 \cdot \theta_{t-4} + \cdots}{10} \end{aligned}$$

Bias Correction for Exp Weighted Averages

When we start with the Exponential Weighted Averages, we are too much influenced by the $v_0 = 0$ value (violet curve):

$$v_0 = 0 \quad \& \quad \beta = 0.98$$

$$v_1 = 0.98 \cdot v_0 + 0.02 \cdot \theta_1 = 0 + 0.02 \cdot \theta_1 \ll \theta_1$$

$$v_2 = 0.98 \cdot (0.98 \cdot v_0 + 0.02 \cdot \theta_1) + 0.02 \cdot \theta_2 = 0.0196 \cdot \theta_1 + 0.02 \cdot \theta_2 \ll \frac{\theta_1 + \theta_2}{2}$$

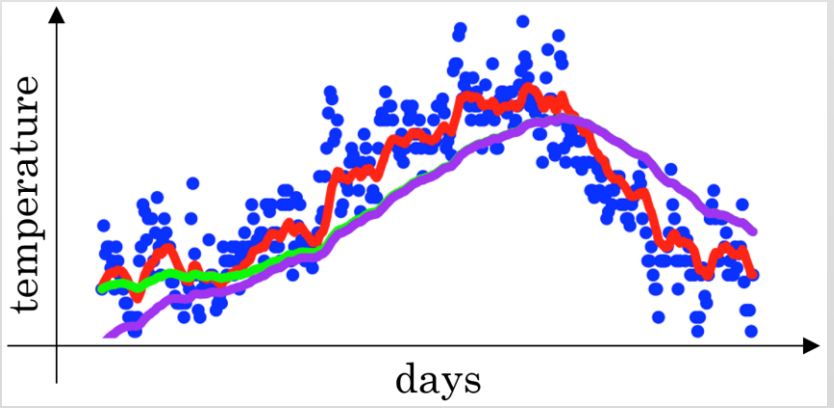
To avoid this, we use the correction factor (green curve) $1 - \beta^t$:

$$v_t = \frac{\beta \cdot v_{t-1} + (1 - \beta) \cdot \theta_t}{1 - \beta^t}$$

$$v_1 = \frac{0.98 \cdot v_0 + 0.02 \cdot \theta_1}{1 - 0.98} = \frac{0 + 0.02 \cdot \theta_1}{0.02} = \theta_1$$

$$v_2 = \frac{0.98 \cdot (0.98 \cdot v_0 + 0.02 \cdot \theta_1) + 0.02 \cdot \theta_2}{1 - 0.98^2} = \frac{0.0196 \cdot \theta_1 + 0.02 \cdot \theta_2}{0.0396} \approx \frac{\theta_1 + \theta_2}{2}$$

Thanks to this bias correction, we do not follow the violet curve but the green (corrected) one:



Gradient Descent with Momentum

Gradient Descent with Momentum:

- Uses exponentially weighted averages of the gradients
- Slows down oscillations that cancel each other out when the gradients differ in the consecutive steps.
- Accelerates the convergence steps like a ball rolling in a bowl if the gradients are similar in the consecutive steps.
- $W := W - \alpha \cdot v_{dw}$
- $b := b - \alpha \cdot v_{db}$
- $v_{dW} := \underbrace{\beta}_{\text{friction}} \cdot \underbrace{v_{dW}}_{\text{velocity}} + \underbrace{(1-\beta) \cdot dW}_{\text{acceleration}}$
- $v_{db} := \beta \cdot v_{db} + (1-\beta) \cdot db$
- The quotient $(1-\beta)$ is often omitted:
- $v_{dW} := \beta \cdot v_{dW} + (1-\beta) \cdot dW$
- $v_{db} := \beta \cdot v_{db} + (1-\beta) \cdot db$
- Hyperparameters: α, β, Typical values: $\alpha = 0.1, \beta = 0.9$
- Bias correction is rarely used with momentum, however might be used.

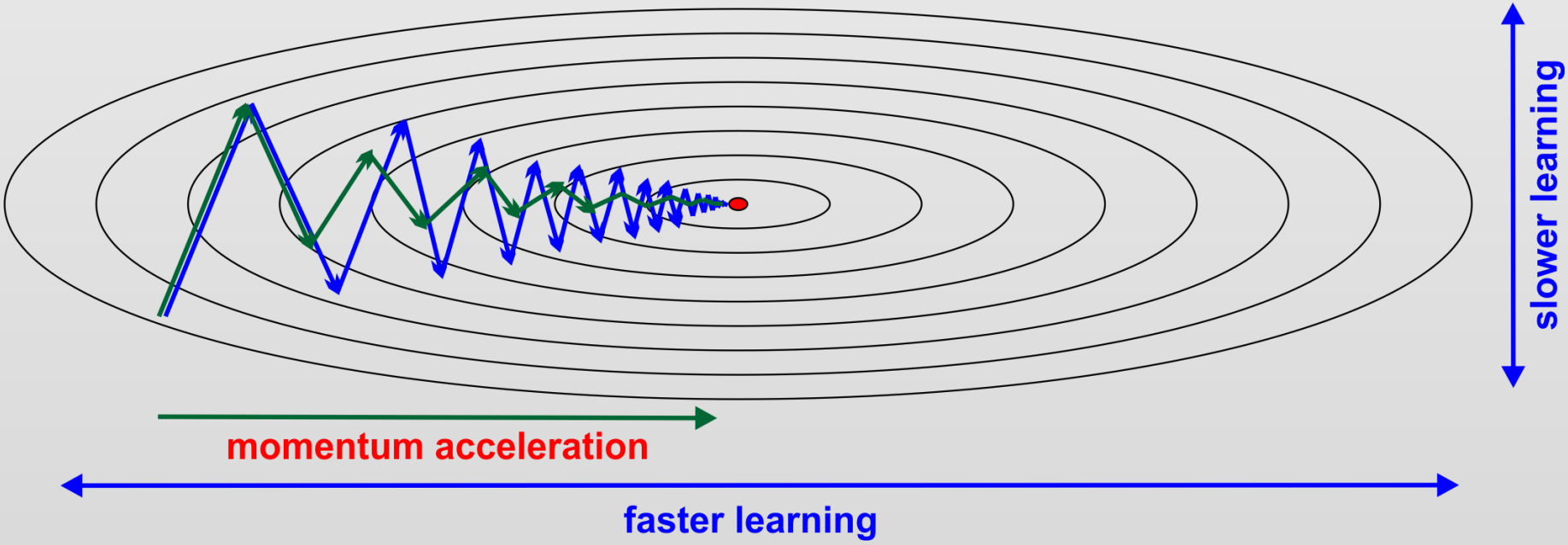
Gradient Descent with Momentum

Gradient Descent with Momentum:

- Uses exponentially weighted averages of the gradients
- Slows down oscillations that cancel each other out when the gradients differ in the consecutive steps.
- Accelerates the convergence steps like a ball rolling in a bowl if the gradients are similar in the consecutive steps.

Oscillations of gradient descent prevent convergence and slows down training

Momentum with gradient descent prevents oscillations and speed up training



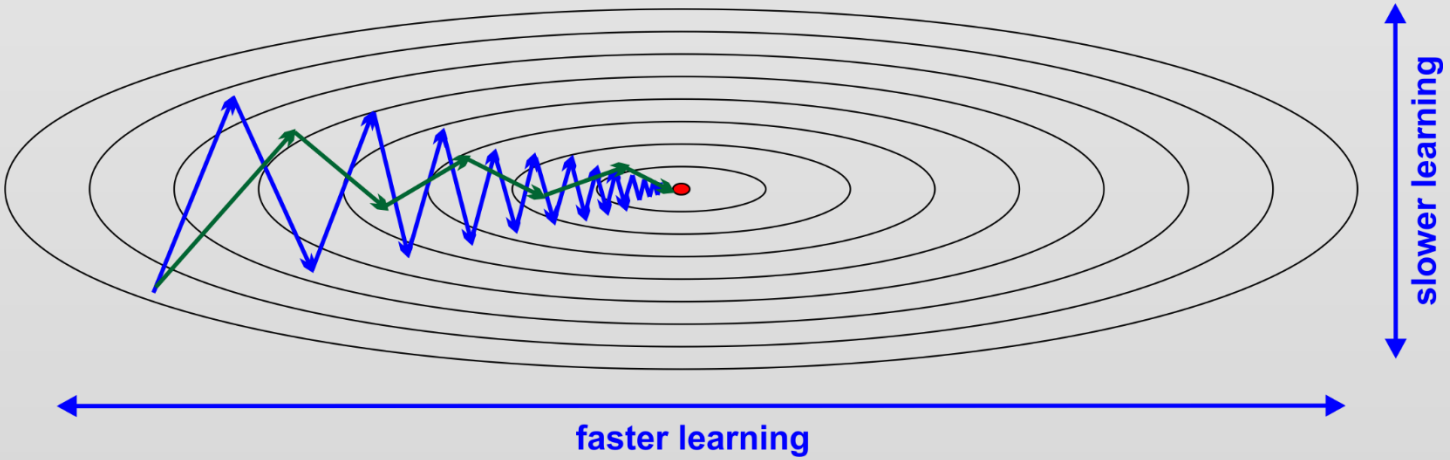
Root Mean Square Propagation

Root Mean Square Propagation (RMSprop):

- Computes exponentially weighted average of the squares of the derivatives
- $s_{dW} \coloneqq \beta \cdot s_{dW} + (1-\beta) \cdot dW^2$ *where dW^2 is element-wise*
- $s_{db} \coloneqq \beta \cdot s_{db} + (1-\beta) \cdot db^2$ *where db^2 is element-wise*
- Parameters are updated in the following way:
- $W \coloneqq W - \alpha \cdot \frac{dW}{\sqrt{s_{dW}}}$ $\quad b \coloneqq b - \alpha \cdot \frac{db}{\sqrt{s_{db}}}$
- *Where $\sqrt{s_{dW}}$ and $\sqrt{s_{db}}$ balance the convergence process independently of how big or how small are dW, db, s_{dW}, and s_{db}.*

Oscillations of gradient descent prevent convergence and slows down training

RMSprop with gradient descent prevents oscillations, balance and speed up training



Adam Optimization Algorithm

Adam optimizer puts momentum and RMSprop together:

- Initialize Hyperparameters:

α – needs to be tuned

β_1 = 0.9 (typical, default)

β_2 = 0.999 (typical , default)

$\varepsilon = 10^{-8}$ (typical , default)

- Initialize: $v_{dW} := 0; v_{db} := 0; s_{dW} := 0; s_{db} := 0$
- Loop for t iterations over the mini-batches of the training epoch:
- Compute gradients dW and db for current mini-batches.
- Compute correction parameters with corrections and final parameter updates:

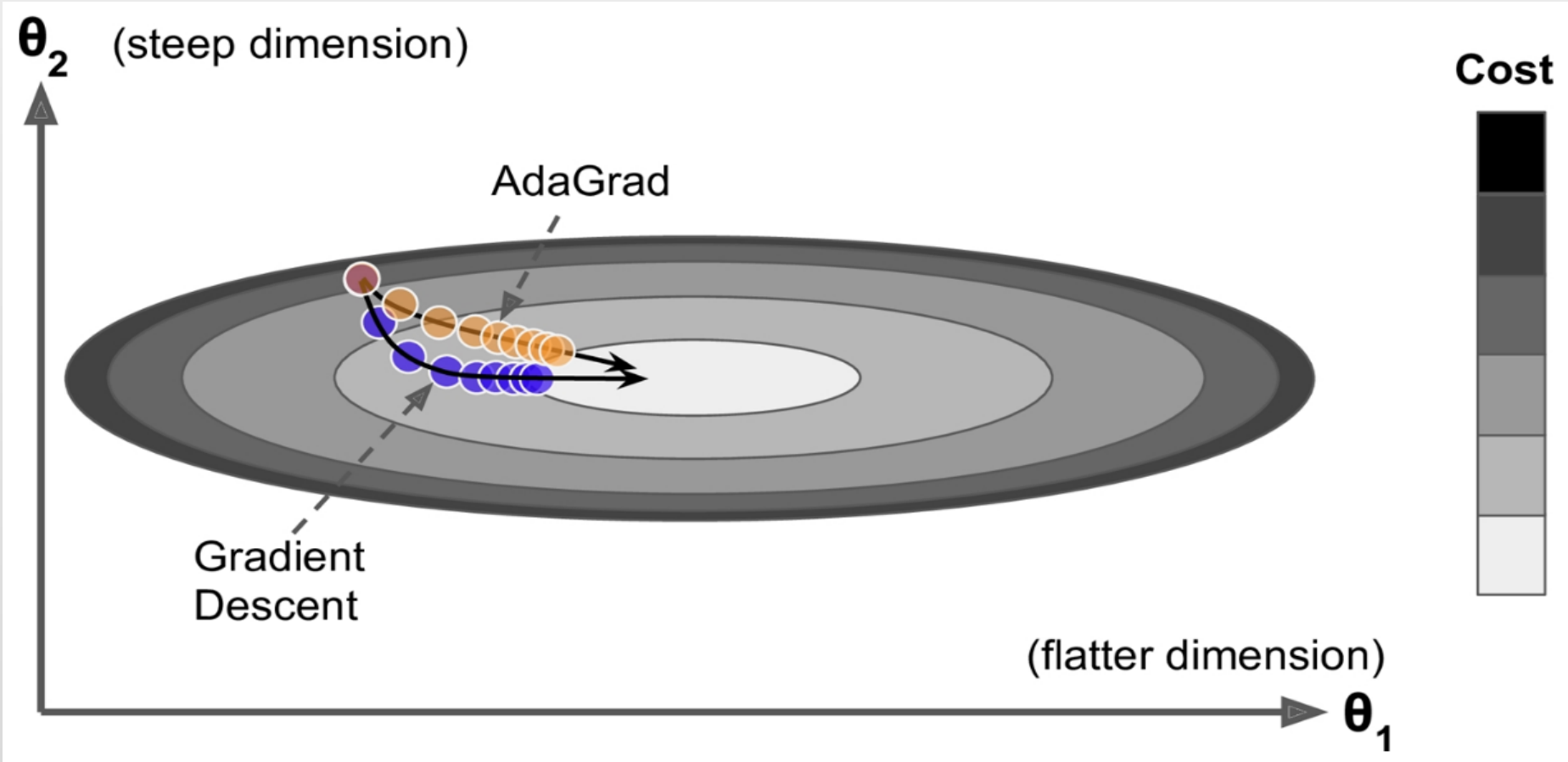
$$v_{dW}^{corr} := \frac{\beta_1 \cdot v_{dW} + (1-\beta_1) \cdot dW}{1-\beta_1^t} \qquad v_{db}^{corr} := \frac{\beta_1 \cdot v_{db} + (1-\beta_1) \cdot db}{1-\beta_1^t}$$

$$s_{dW}^{corr} := \frac{\beta_2 \cdot s_{dW} + (1-\beta_2) \cdot dW^2}{1-\beta_2^t} \qquad s_{db}^{corr} := \frac{\beta_2 \cdot s_{db} + (1-\beta_2) \cdot db^2}{1-\beta_2^t}$$

$$W := W - \alpha \cdot \frac{v_{dW}^{corr}}{\sqrt{s_{dW}^{corr}} + \varepsilon} \qquad b := b - \alpha \cdot \frac{v_{db}^{corr}}{\sqrt{s_{db}^{corr}} + \varepsilon}$$

AdaGrad Optimization

Adaptive Gradient descent (AdaGrad) decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. AdaGrad frequently performs well for simple quadratic problems, but it often stops too early when training neural networks.



Learning Rate Decay

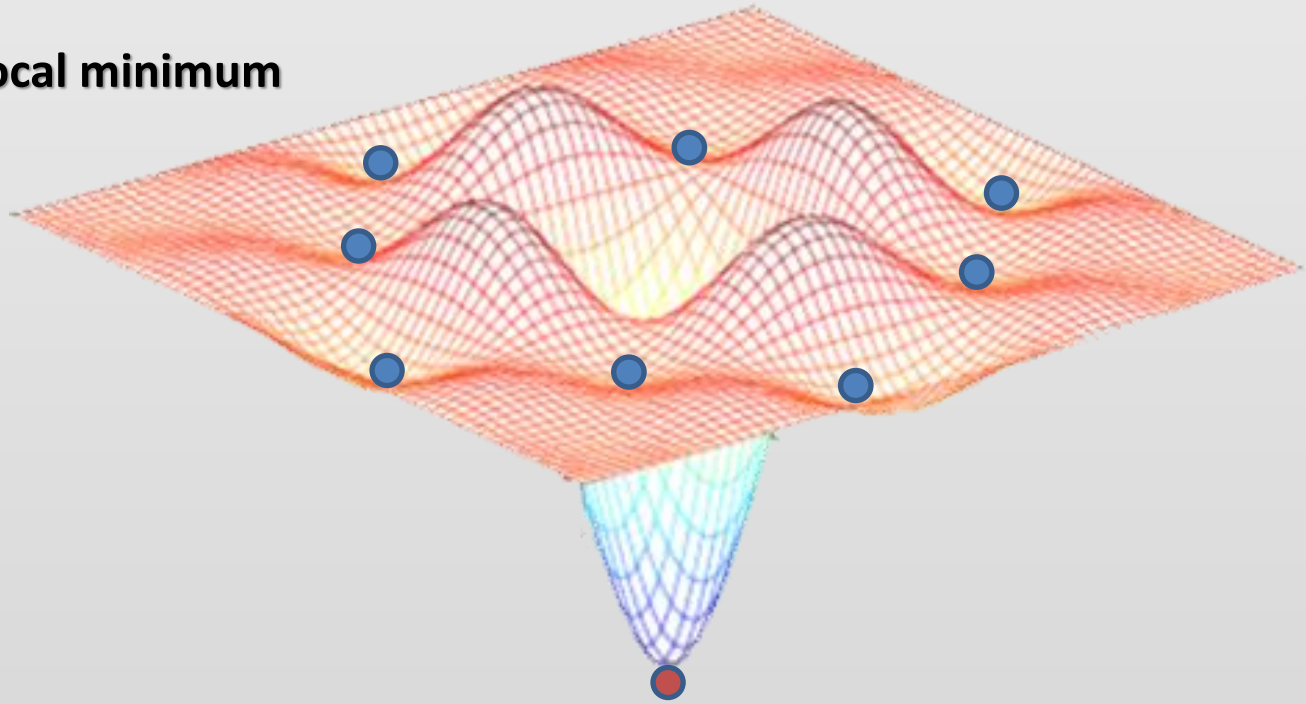
To avoid oscillation close to the minimum of the loss function, we should use non-constant learning rate but its decay, e.g.:

- We can decay the learning rate along with the training epochs:
  - $\alpha = \frac{\alpha_0}{1 + decayrate \cdot noepoch}$
- We can use an exponential learning rate decay:
  - $\alpha = \alpha_0 \cdot e^{-decayrate \cdot noepoch}$
- Another way to decay a learning rate:
  - $\alpha = \frac{k \cdot \alpha_0}{\sqrt{noepoch}}$
- We can also use a staircase decay, decreasing a learning rate after a given number of epochs by half or in another way.

Local Optima (Minima)

A loss function can have many local minima, but we are interested in finding the global minimum do reduce training error as much as possible:

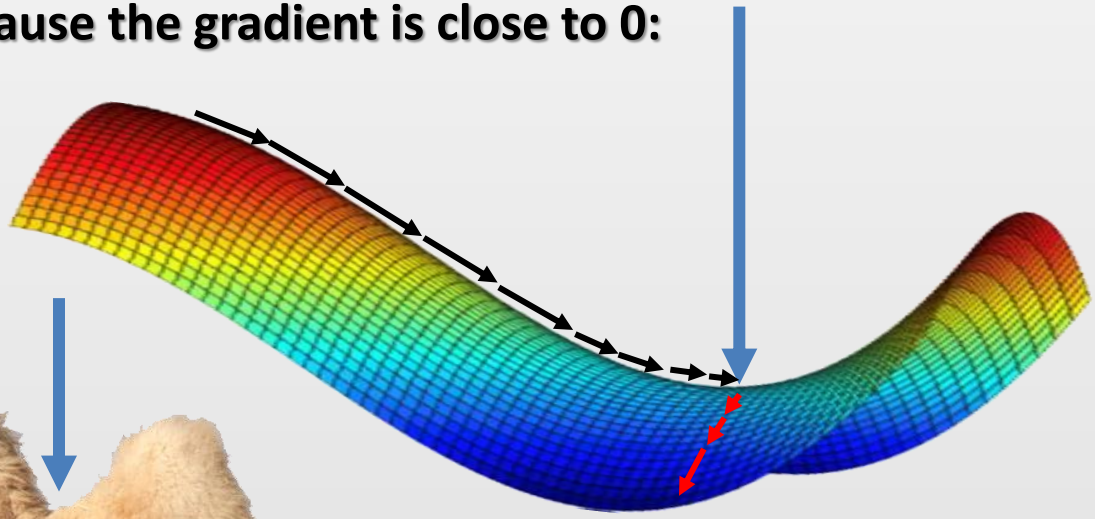
- **We must avoid stacking in local minima of the loss function.**
- **We can try to define such a loss function to have no local minima.**
- **We can try to escape from local minima using mini-batches, momentum, RMSprop, Adam optimizer.**
- **The gradient in any local minimum is always equal to 0!**



Saddle Points and Plateaus

Even if the loss function has no local minima, it can have saddle points where the gradient algorithm can stack because the gradient is close to 0:

- **The loss function surface can be locally flat.**
- **We want to escape from such local plateaus (flat areas) where the gradients are very small.**

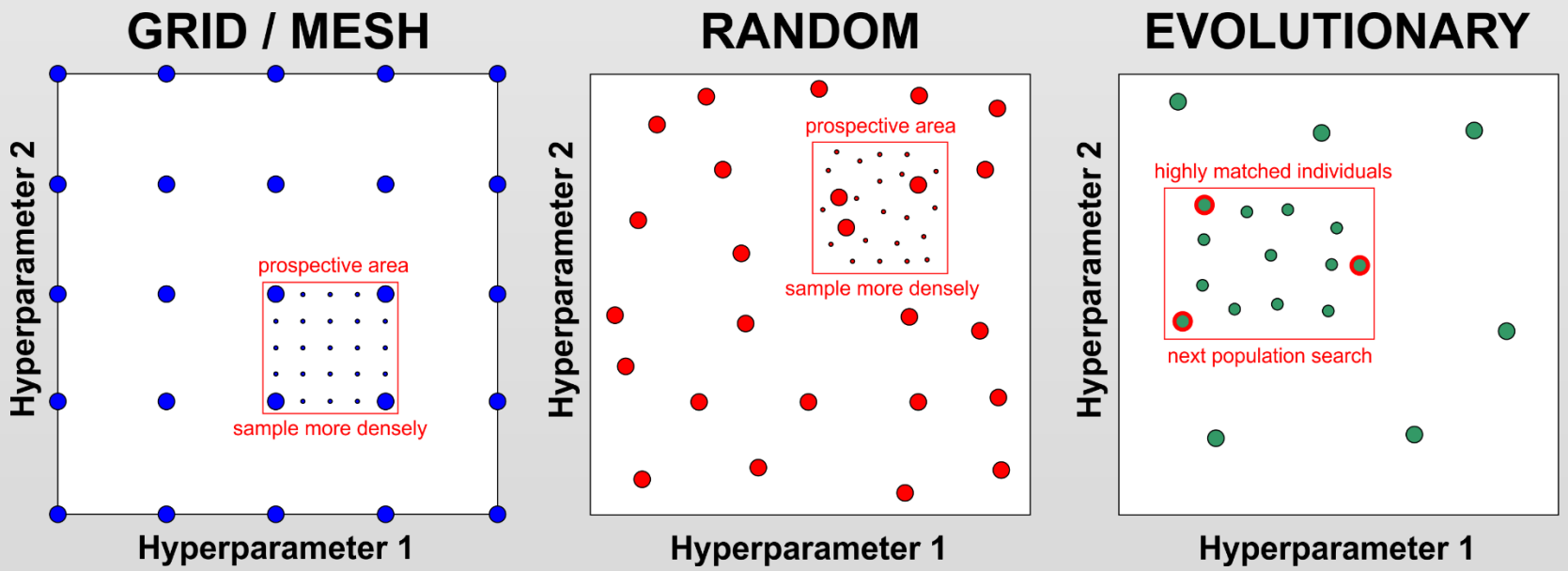


Optimization of Hyperparameters

In deep learning, we have a huge number of hyperparameters that must be tuned to get a good enough computational model.

We have various techniques that help us to deal with this problem:

1. **Systematically chose hyperparameters over the grid (mesh) tightening the prospective areas (sampling more densely prospective areas) (computationally very expensive due to the huge number of combinations to check).**
2. **Chose hyperparameters randomly many times sampling more densely prospective areas (uncertain but may be faster if you are lucky).**
3. **Use evolutional and genetic approaches (smart choice based on previous populations).**

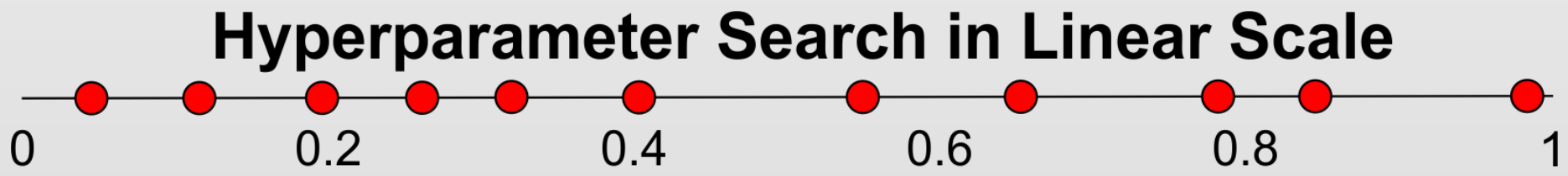


Scale of Hyperparameter Optimization

Sampling hyperparameters we cannot simply scale them in a linear scale. Sometimes we need to use a different scale, e.g. logarithmic or exponential. Otherwise, we will sample not useful hyperparameters, not improving the developed computational model.

For example, when we want to sample learning rate, we should use a logarithmic scale, e.g.:

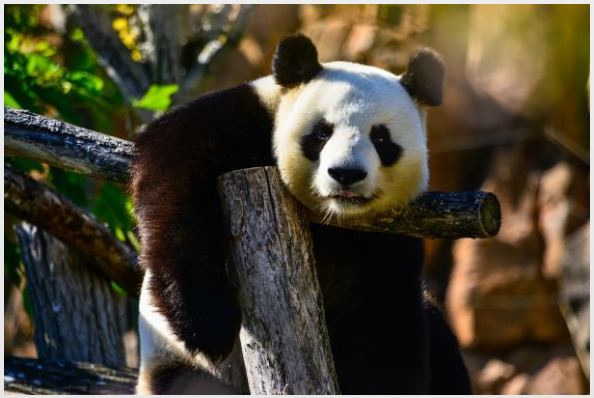
$$\alpha = 10^{r} \quad where \quad r = -4 * np.random.rand()$$



Approaches to Choose Hyperparameters

There at two main approaches to search for suitable hyperparameters:

- **A babysitting model (Panda strategy) – in which we try to look at the performance of a model and improve patiently its hyperparameters.**

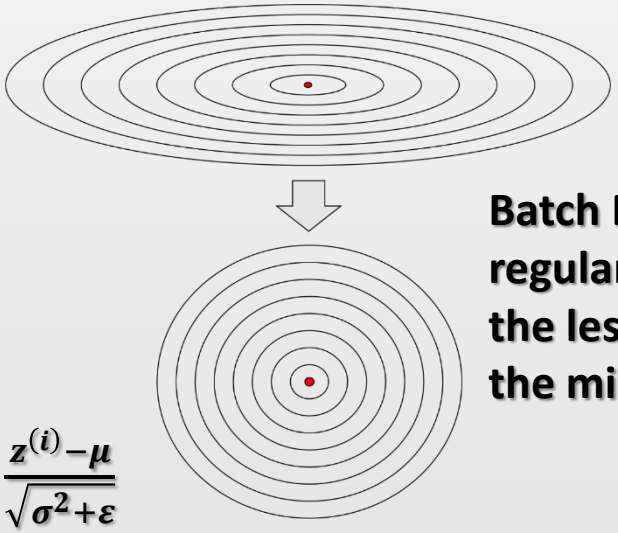


- **Many models train in parallel (Caviar strategy) – check many models using various combinations of the hyperparameters and choose the best one automatically.**
  If you have enough computational resources, you can afford this model.

Batch Normalization (Batch Norm)

We normalize data to make their gradients comparable and to speed up the training process:

- We compute mean:
- $\mu = \frac{1}{m}\sum_i z^{(i)}$
- and variance:
- $\sigma^2 = \frac{1}{m}\sum_i \left(z^{(i)} - \mu\right)^2$
- to normalize:
- $\tilde{z}^{(i)} = \gamma \cdot z^{(i)}_{norm} + \beta$ where $z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$



Batch Norm has a slight regularization effect, the less the bigger are the mini-batches.

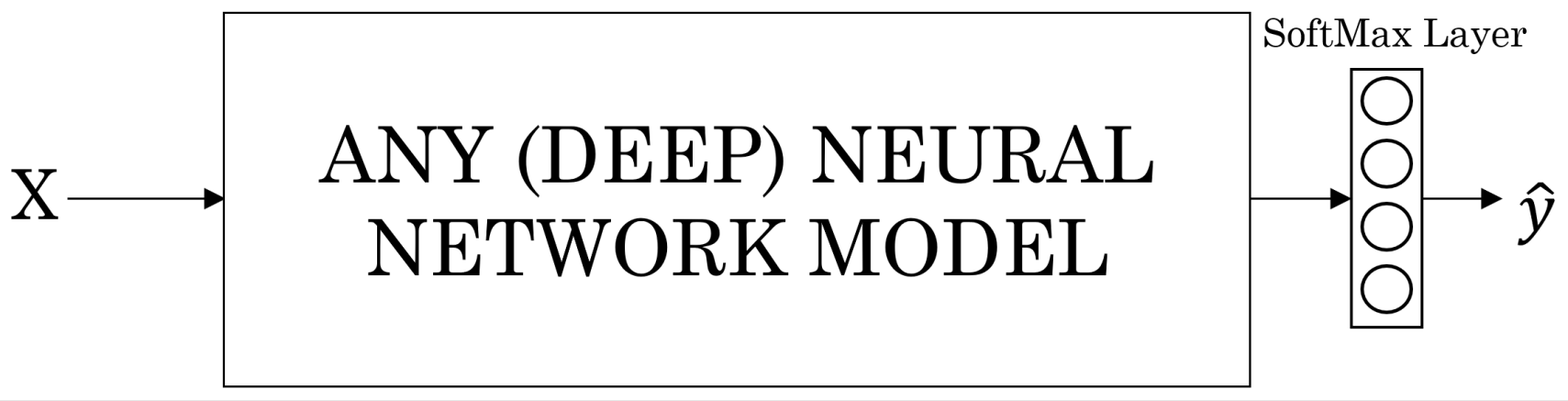
where β, γ are trainable parameters ($\beta^{[l]} := \beta^{[l]} - \alpha \cdot d\beta^{[l]}$, $\gamma^{[l]} := \gamma^{[l]} - \alpha \cdot d\gamma^{[l]}$) of the model, so we use gradients to update them in the same way as weights and biases.

- If $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$, then $\tilde{z}^{(i)} = z^{(i)}$
- so the sequence of input data processing with normalization is as follows:
- $x^{\{t\}} \rightarrow z^{[1]} \rightarrow \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}\left(\tilde{z}^{[1]}\right) \rightarrow z^{[2]} \rightarrow \tilde{z}^{[2]} \rightarrow a^{[2]} = g^{[2]}\left(\tilde{z}^{[2]}\right) \ldots$
- and we apply it usually for $t \in \{1, \ldots, T\}$ minibatches subsequently.
- Thus, we have $W^{[l]}$, $b^{[l]}$, $\gamma^{[l]}$, and $\beta^{[l]}$ parameters for each layer, but we do not need to use $b^{[l]}$, because the shifting function is supplied by $\beta^{[l]}$.

SoftMax Regression

SoftMax regression is a generalization of **logistic regression** for multi-class classification:

- It can be use together with different neural network architectures.
- It is used in the last network layer (L-layer) to proceed multi-class classification.



- **Multi-class classification** is when our dataset defines more than 2 classes, and the network answer should be not only between the answers yes or no.
- For each trained class (because there might be more classes in the dataset than the trained number of classes, but they are not labelled for supervised training), we create a single output neuron that should give us the probability of the recognized class of the input data. So for all trained classes we get the output vector $\hat{y}$ that defines the probabilities of classification of the input X to one of the trained classes.
- **SoftMax layer** normalizes the final outputs $a^{[L]}$ of all neurons of this layer by the sum of the computed outputs $\hat{a}^{[L]}$ of the activation function used in this layer.

SoftMax Evaluation

In the SoftMax layer, the activation function $g^{[L]}$ is defined as: $\hat{a}^{[L]} = g^{[L]}(z^{[L]}) = e^{z^{[L]}}$

Specifically for each output neuron: $\hat{a}_j^{[L]} = g^{[L]}\left(z_j^{[L]}\right) = e^{z_j^{[L]}}$

We use the sum of all output values of the activation functions $\hat{a}_j^{[L]}$

$$e_{sum} = a_j^{[L]} = \frac{\hat{a}_j^{[L]}}{\sum_{j=1}^{n^{[L]}} \hat{a}_j^{[L]}}$$

to compute the final output values of output SoftMax nodes as normalized by this sum:

$$a_j^{[L]} = \frac{\hat{a}_j^{[L]}}{e_{sum}} = a_j^{[L]} = \frac{\hat{a}_j^{[L]}}{\sum_{j=1}^{n^{[L]}} \hat{a}_j^{[L]}}$$

Thanks to this approach, the sum of all output values always sums up to 1, and the output values can be used to emphasise the probabilities of classifications to all trained classes and point the winner, e.g.:

$$if \quad z^{[L]} = \begin{bmatrix} 2 \\ 5 \\ -1 \\ 3 \end{bmatrix} \quad then \quad \hat{a}^{[L]} = \begin{bmatrix} e^2 \\ e^5 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 7.39 \\ 148.41 \\ 0.37 \\ 20.09 \end{bmatrix}$$

$$e_{sum} = 7.39 + 148.41 + 0.37 + 20.09 = 176.26 \quad then \quad a^{[L]} = \begin{bmatrix} 7.39/e_{sum} \\ 148.41/e_{sum} \\ 0.37/e_{sum} \\ 20.09/e_{sum} \end{bmatrix} = \begin{bmatrix} 0.042 \\ 0.842 \\ 0.002 \\ 0.114 \end{bmatrix}$$

As we can notice $\sum_{j=1}^{n^{[l]}} \hat{a}_j^{[L]} = 1$, in our case $0.042 + 0.842 + 0.002 + 0.114 = 1.0$

SoftMax Loss Function

When using SoftMax, the loss function is defined as:

$$L(\hat{y}_j, y_j) = -\sum_{j=1}^{n^{[L]}} y_j \, log \, \hat{y}_j = -y_c \, log \, \hat{y}_c = -log \, \hat{y}_c$$

because only for $j = c$ it is true that $y_c \neq 0$, i.e. for the class it defines, moreover, $y_c = 1$:

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Therefore, the loss function can be minimized, when the $\hat{y}_c$ is maximised, i.e. tends to be close 1:

$$\hat{y} = a^{[L]} = \begin{bmatrix} 0.2 \\ 0.4 \\ 0.3 \\ 0.1 \end{bmatrix}$$

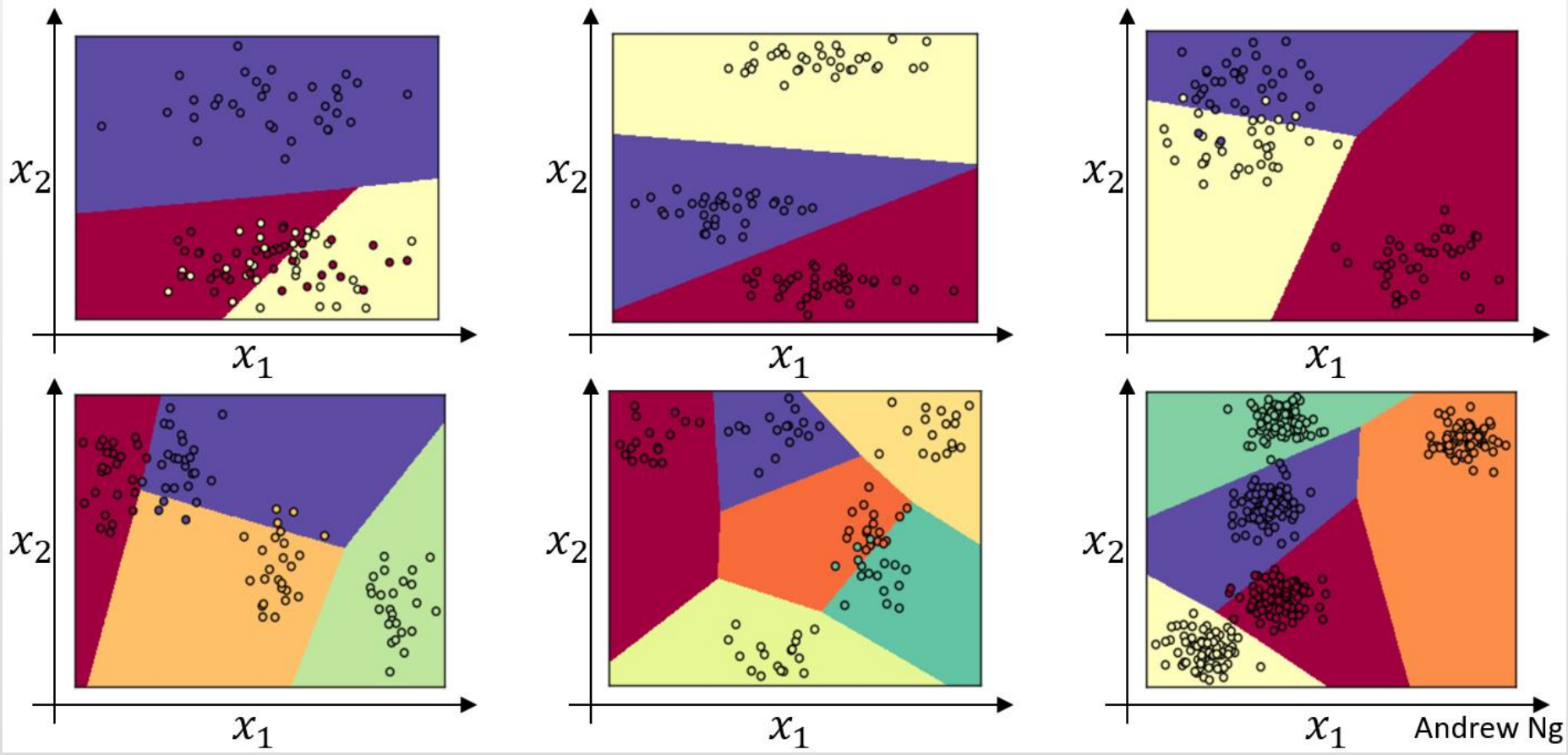
So the goal of the training is intuitively fulfilled.

So the backpropagation step is started from:

$$dz^{[L]} = \hat{y} - y$$

Possible results get by SoftMax

Consider the trustworthy of the following example results got by the flat SoftMax neural network using various numbers of trained classes:



Can we trust such results or should we use deeper architecture to classify inputs with higher confidence?

SoftMax Modifications

In the SoftMax layer, we can also use another activation function $g^{[L]}$ to compute outputs values $\hat{a}^{[L]}$, e.g. if the activation function $g^{[L]}$ would be a logistic function, then we got $\hat{a}_j^{[L]} \in (0,1)$, e.g. for the four trained classes, we get the output $\hat{a}^{[L]}$ that is normalized to $a^{[L]}$:

(a) We have two initial high estimations of the logistic functions 0.98 and 0.92:

$$\hat{a}^{[L]} = \begin{bmatrix} 0.06 \\ 0.98 \\ 0.04 \\ 0.92 \end{bmatrix} \quad sum = 0.06 + 0.98 + 0.04 + 0.92 = 2.0 \quad a^{[L]} = \begin{bmatrix} 0.06/sum \\ 0.98/sum \\ 0.04/sum \\ 0.92/sum \end{bmatrix} = \begin{bmatrix} 0.03 \\ 0.49 \\ 0.02 \\ 0.41 \end{bmatrix}$$

In this case, we got two quite high estimations of the logistic functions 0.98 and 0.92, but the final multi-class classification is not so high because the network is not sure which of these two highly approximated classes should the input belong to?! The result show this hesitation: 0.49 and 0.41.

The highest output value of the soft-max layer neurons is treated as the winning one and the most probable classification over the trained classes, but we also should take into account the final highest values that reduce the confidence of the answer given by the network!

Consider another classification result that gives only one initial high estimation 0.88 for class 2, but it is lower than 0.98. Which of these two classifications should we trust more (a) or (b) and why?

(b) We have only one initial high estimation 0.88 but it is lower than 0.98:

$$\hat{a}^{[L]} = \begin{bmatrix} 0.14 \\ 0.88 \\ 0.12 \\ 0.06 \end{bmatrix} \quad sum = 0.14 + 0.88 + 0.12 + 0.06 = 1.2 \quad a^{[L]} = \begin{bmatrix} 0.14/sum \\ 0.88/sum \\ 0.12/sum \\ 0.06/sum \end{bmatrix} = \begin{bmatrix} 0.12 \\ 0.73 \\ 0.10 \\ 0.05 \end{bmatrix}$$

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

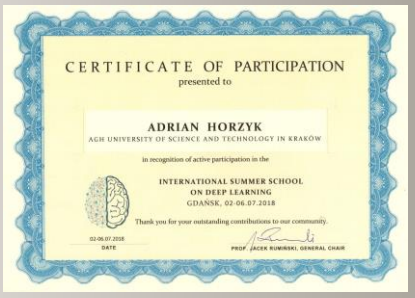
Bibliography and Literature

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



AGH

University of Science and Technology in Krakow, Poland

xxx

XXXXX:

- xxxxx

xxx

XXXXX:

- xxxxx

xxx

XXXXX:

- xxxxx

xxx

XXXXX:

- xxxxx

xxx

XXXXX:

- xxxxx

xxx

XXXXX:

- xxxxx

xxx

XXXXX:

- xxxxx