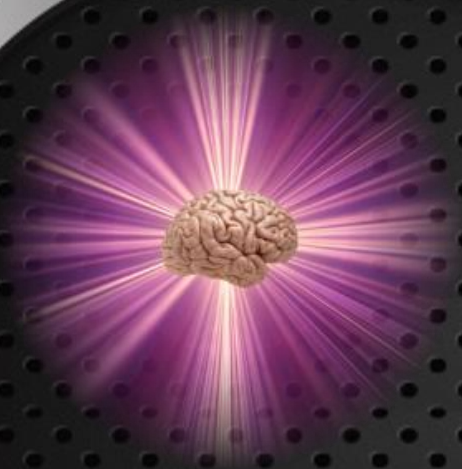
AGH University of Science and Technology
Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering
Department of Biocybernetics and Biomedical Engineering



Computational Intelligence

Data and Model Optimization and Performance Measuring and Improvement

Adrian Horzyk
horzyk@agh.edu.pl

Google: Adrian Horzyk

Orthogonalization Process and Controlling the Training Process

How to define knobs to control the training?

Orthogonalization

Orthogonalization:

- is a clear-eyed process about what to tune and how to achieve a supposed effect.
- is the process that lets us refer to individual hyperparameters in such a way that we can fix a selected training problem by tuning on a limited subset of hyperparameters.

- Why do we prefer to use drones over helicopters?
- Which one is easier to control and why?
- Is it easier to control a single knob changing a single parameter or a compound joystick changing many parameters at the same time?
- Have you tried to fly a helicopter or a drone in the past? What is your experience?

Car Controllers

What about the car controllers like a weal, pedals, knobs, shifts, and buttons?

Is it easier to control it (e.g. speed) when each parameter is controlled separately?

How do you prefer to control the car:

- set of controllers (like weal, pedals, knobs, shifts, and buttons) that control individual parameters of the car (speed, direction, etc.) or
- an integrated controller (like a joystick) that can control a combination of parameters (like speed and direction) by the same move?

Scores for Measuring Performance and Generalization

How to measure the model quality?

Single Number Evaluation Metric

When adapting the model, we usually train it with different hyperparameters and compare achieved precision and recall:

- **Precision** – defines the percentage of correct classifications, e.g. if the achieved precision is 98% after the training is finished, and the network says that the input is a car, there is a 98% that it really is a car.
- **Recall** – is the percentage of correctly classified objects (inputs) for training classes, e.g. how many cars of all the cars from training data were correctly classified?

Classifier	Precision	Recall	F_1 Score
Classifier A	96%	90%	92,90%
Classifier B	98%	88%	92,73%
Classifier C	94%	93%	93,50%

$$F_1 = \frac{2}{Precision^{-1} + Recall^{-1}}$$

$$F_\beta = \frac{(1+\beta^2) \cdot Precision \cdot Recall}{\beta^2 \cdot Precision + Recall} = \frac{(1+\beta^2) \cdot TP}{(1+\beta^2) \cdot TP + \beta^2 \cdot FN + FP}$$

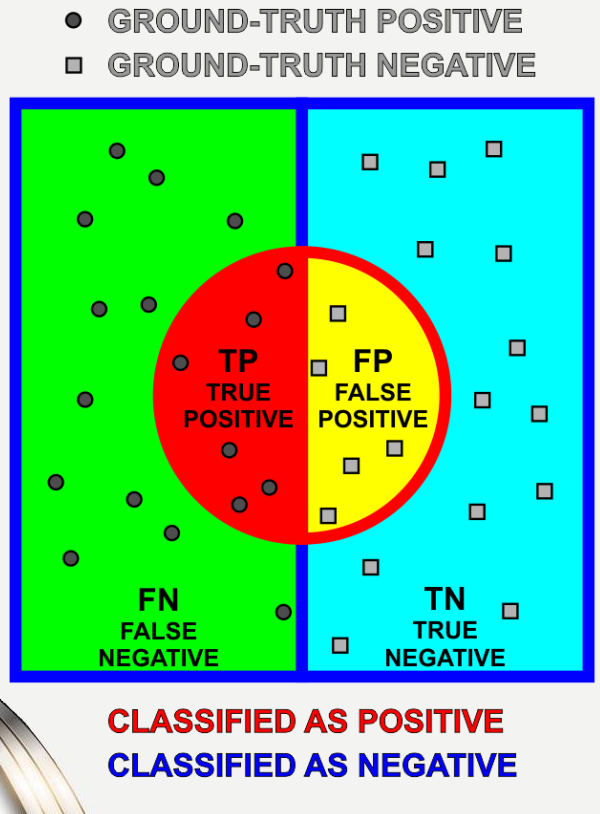
β - how many times recall is more important than precision
TP – true positive, FP – false positive, FN – false negative

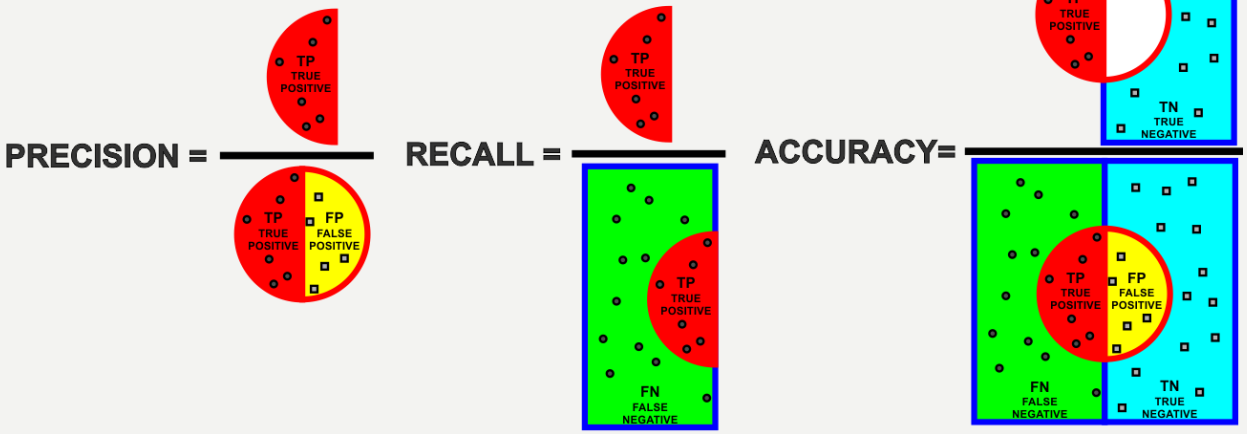
- **Which classifier from the above three is the best one?**
- **It turns out that there is often a trade-off between precision and recall, but we want to care about both of them!**
- **We sometimes use F1 Score that is a harmonic mean of the precision and recall.**

Confusion Matrix & Popular Scores

Confusion Matrix groups the results of binary classification:

- TP (true positive) – is the number of examples correctly classified as positive.
- FP (false positive) – is the number of examples incorrectly classified as positive.
- TN (true negative) – is the number of examples correctly classified as negative.
- FN (false negative) – is the number of examples incorrectly classified as negative.





Precision = TP / (TP + FP)

Precision – a ratio of how many examples were correctly classified as positive (class A) to all examples classified as positive (while not all are really positive, i.e. of class A).

Recall = TP / (TP + FN)

Recall – a ratio of how many examples were correctly classified as positive (class A) to all positive (class A) examples in the training set.

Accuracy = (TP + TN) / ALL

Accuracy – a ratio of how many examples were correctly classified to all examples in the training set.

Metrics and Measures of Results

The most popular measures of results are:

CONFUSION MATRIX			Prevalence = PP / ALL
All Examples (ALL)	Defined as Positive (P) = TP + FN	Defined as Negative (N) = FP + TN	Accuracy (ACC) = (TP + TN) / ALL
Predicted as Positive (PP) = TP + FP	True positive (TP)	False Positive (FP)	Precision = Positive Predictive Value (PPV) = TP / PP
			False Discovery Rate (FDR) = FP / PP
Predicted as Negative (PN) = FN + TN	False Negative (FN)	True Negative (TN)	False Omission Rate (FOR) = FN / PN
			Negative Predictive Value (NPV) = TN / PN
	True Positive Rate (TPR) = Recall = Sensitivity = TP / P	False Positive Rate (FPR) = Fall-out = FP / N	F_1 = 2 · Precision · Recall / (Precision + Recall)
			Positive Likelihood Ratio (LR+) = TPR / FPR
	False Negative Rate (FNR) = Miss Rate = FN / P	True Negative Rate (TNR) = Selectivity = Specificity = TN / N	Negative Likelihood Ratio (LR−) = FNR / TNR
			Diagnostic Odds Ratio (DOR) = LR+ / LR−

Metrics for Comparison of Classifiers

When we have results collected by many classifiers, we need to choose the best one, preferably using a single criterion that takes into account, e.g. various positive or negative classifications for all classes separately:

- **Compute the average error or harmonic mean for all classes and classifiers to compare them:**

Classifier	Class A	Class B	Class C	Class D	Average	Harmonic Mean
A	95%	90%	94%	99%	94.5%	94.39%
B	96%	93%	97%	94%	95.0%	94.97%
C	92%	93%	95%	97%	94.3%	94.21%
D	94%	95%	99%	94%	95.5%	95.46%
E	97%	98%	95%	97%	96.8%	96.74%
F	99%	91%	96%	92%	94.5%	94.39%

- **Thanks to such measures, we can more easily point out the best classifier taking into account results collected for all classes.**

Other Criteria for Choosing Classifier

Sometimes an application must run in real-time, so we cannot simply choose the classifier with the best accuracy, precision, or recall, but we must take into account the classification time:

- **The accuracy must be the highest but available at the acceptable time, e.g. < 100 ms**

Classifier	Accuracy	Classification Time
A	94.5%	70 ms
B	95.0%	95 ms
C	94.3%	35 ms
D	95.5%	240 ms
E	96.8%	980 ms
F	94.5%	60 ms

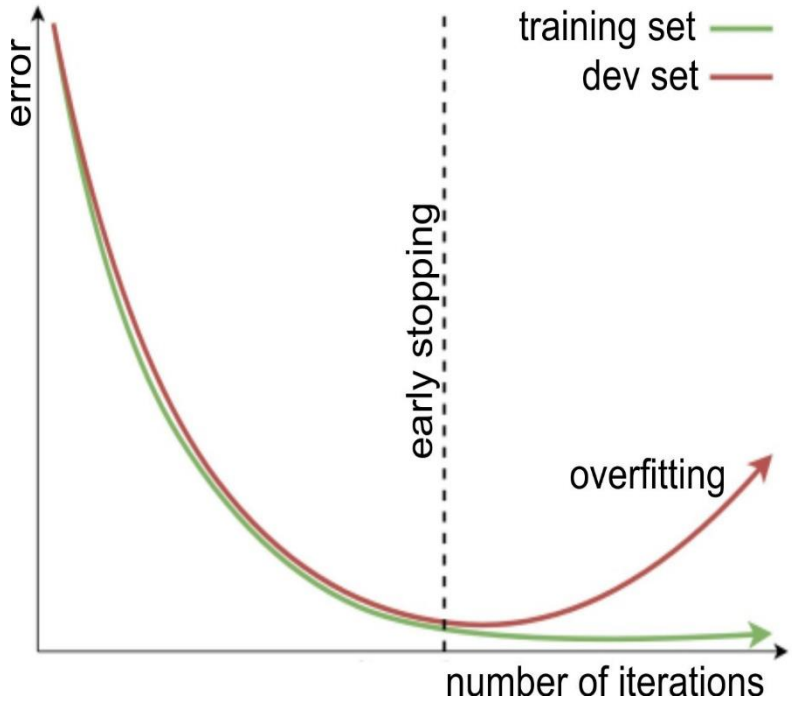
- **Sometimes we must take into account additional criteria to find out the suitable classifier for a given practical problem, e.g., we choose this one with the highest accuracy of those, which have their classification time lower than 100 ms.**
- **The accuracy is optimized, while the classification time must be satisfied.**
- **So we have to do with multi-criteria optimization here.**

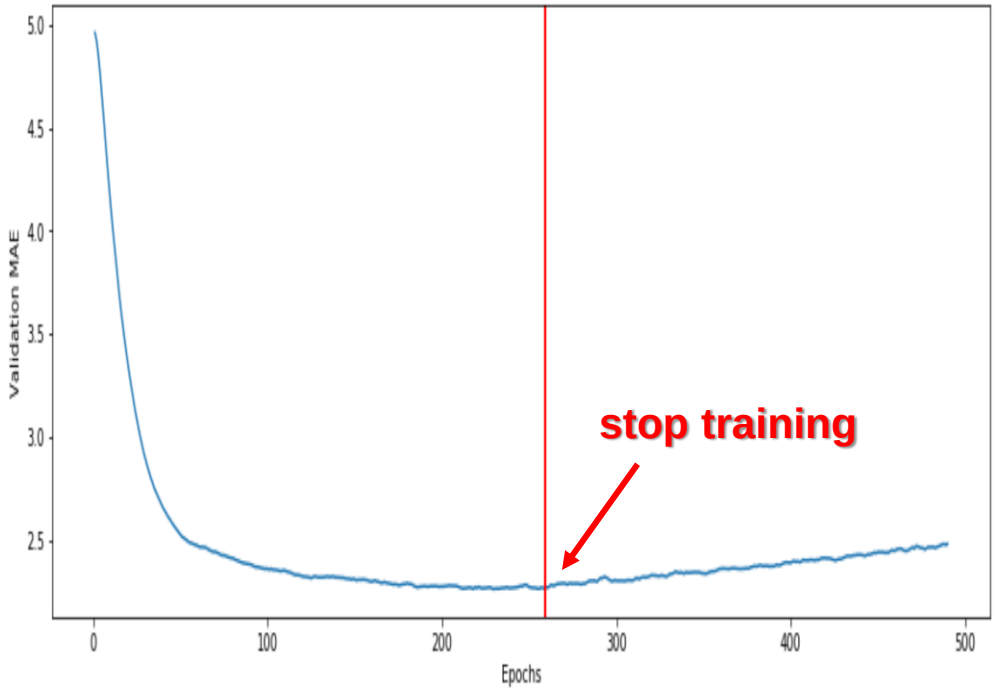
Preventing Overfitting

How to stop before the model starts overfitting?

Early Stopping

One of the easiest method preventing overfitting is to use "early stopping" of the training process, which is stopped when the error on the dev set starts to grow.





We save the model during training and use the last model with the least dev error.

This method does not cure overfitting but only reacts its symptoms and prevents its occurrence when it reveals.

Data Augmentation and Synthetizing Training Data

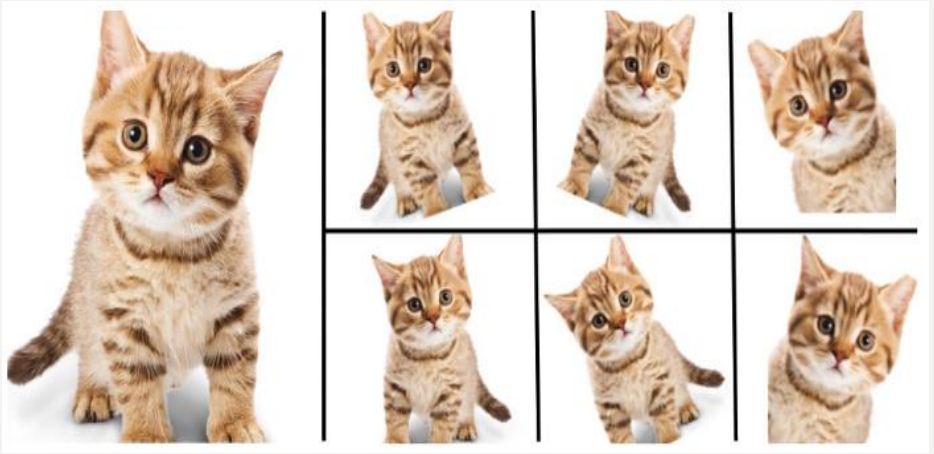
What can we do when having too small number of training data or unequal number of representatives?

Data Augmentation

If we have not enough data to train the model or classes are represented by very different number of representants, we can augment the training data of given less numerous classes appropriately or all training data to avoid training limitations or privileging the most numerous classes.

Augmentation (image data generation) is a standard method implemented to images which can be easy augmented using the following operations:

- Shift and Rotate
- Scale (zoom in or out)
- Shearing (different parts of images)
- Flip (horizontally or vertically)
- Inverse or change colors
- Apply random jitters and perturbations



```
datagen = ImageDataGenerator(
        rotation_range=50,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')
```

Augmentation Prevents Overfitting

Data augmentation takes the approach of generating more training data from existing training samples, by "augmenting" the samples via a number of random transformations (like rotation, shifting, zooming, flipping etc.) that yield believable-looking images.

The goal is that at training time, the model would never see the exact same picture twice. This helps the model get exposed to more aspects of the data and generalize better.

Thanks to it, it also prevents overfitting that is caused by having too few samples to learn from and to cover input data space enough representatively, rendering us unable to train a model able to generalize to new data.

In Keras, this can be done by configuring a number of random transformations to be performed on the images read by the ImageDataGenerator instance.

Augmentation by ImageDataGenerator

The most popular parameters of ImageDataGenerator are:

rotation_range is a value in degrees (0-180), a range within which to randomly rotate pictures.

width_shift and **height_shift** are ranges (as a fraction of total width or height) within which to randomly shift pictures vertically or horizontally.

shear_range is for randomly applying shearing transformations.

zoom_range is for randomly zooming inside pictures.

horizontal_flip is for randomly flipping half of the images horizontally - relevant when there are no assumptions of horizontal asymmetry (e.g. real-world pictures).

fill_mode is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

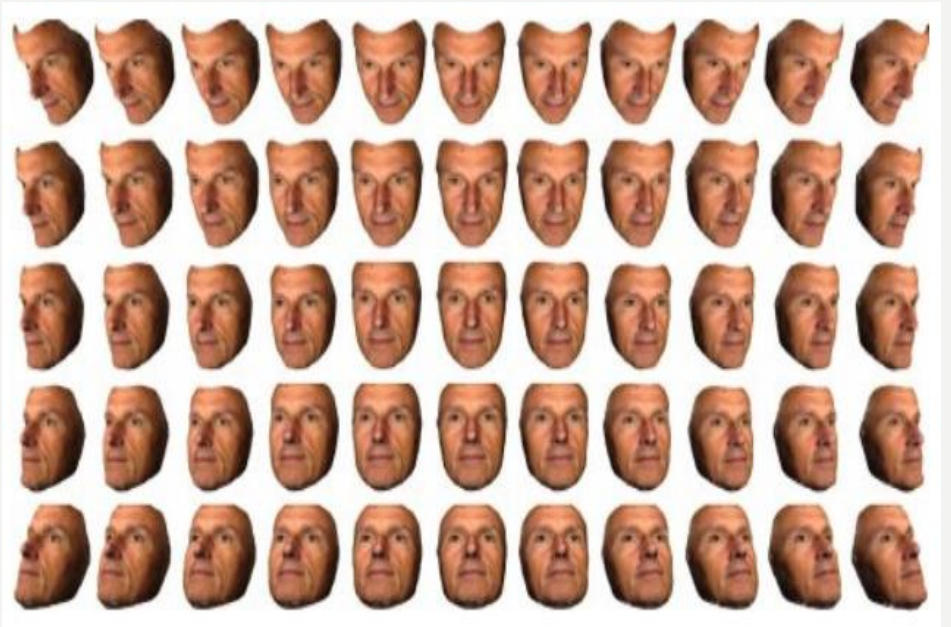
```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
      rotation_range=50,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

Image Data Generator

In Keras, the ImageDataGenerator supplies us with a rich set of transformations:

```
tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    zca_epsilon=1e-06,
    rotation_range=0,
    width_shift_range=0.0,
    height_shift_range=0.0,
    brightness_range=None,
    shear_range=0.0,
    zoom_range=0.0,
    channel_shift_range=0.0,
    fill_mode="nearest",
    cval=0.0,
    horizontal_flip=False,
    vertical_flip=False,
    rescale=None,
    preprocessing_function=None,
    data_format=None,
    validation_split=0.0,
    dtype=None,
)
```



Synthetic Training Data

When dev data, testing data or real-world data differ from training data (e.g. are noisy), we can try to artificially synthetize new training data that will be more similar to real-world data (noise data augmentation), e.g.:

- add typical noise to training data.
- blur training data.
- add some distortions to training data.

Such techniques allow us to overcome the Data Mismatch Problem between training data and real-world noisy data.

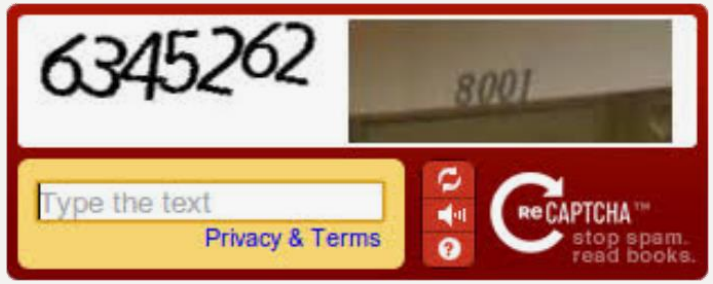
When dealing with texts, we can use various text generators or transformers like SynthText or TextRenderer used e.g. by CAPTCHA or HIP.

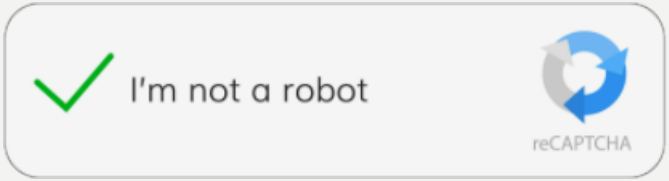
CAPTCHA & HIP

Web services are often protected with a challenge that's supposed to be easy for people to solve, but difficult for computers. Such a challenge is often called:

- **CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) or**
- **HIP (Human Interactive Proof).**

HIPs are used for many purposes, such as to reduce email and blog spam and prevent brute-force attacks on web site passwords.





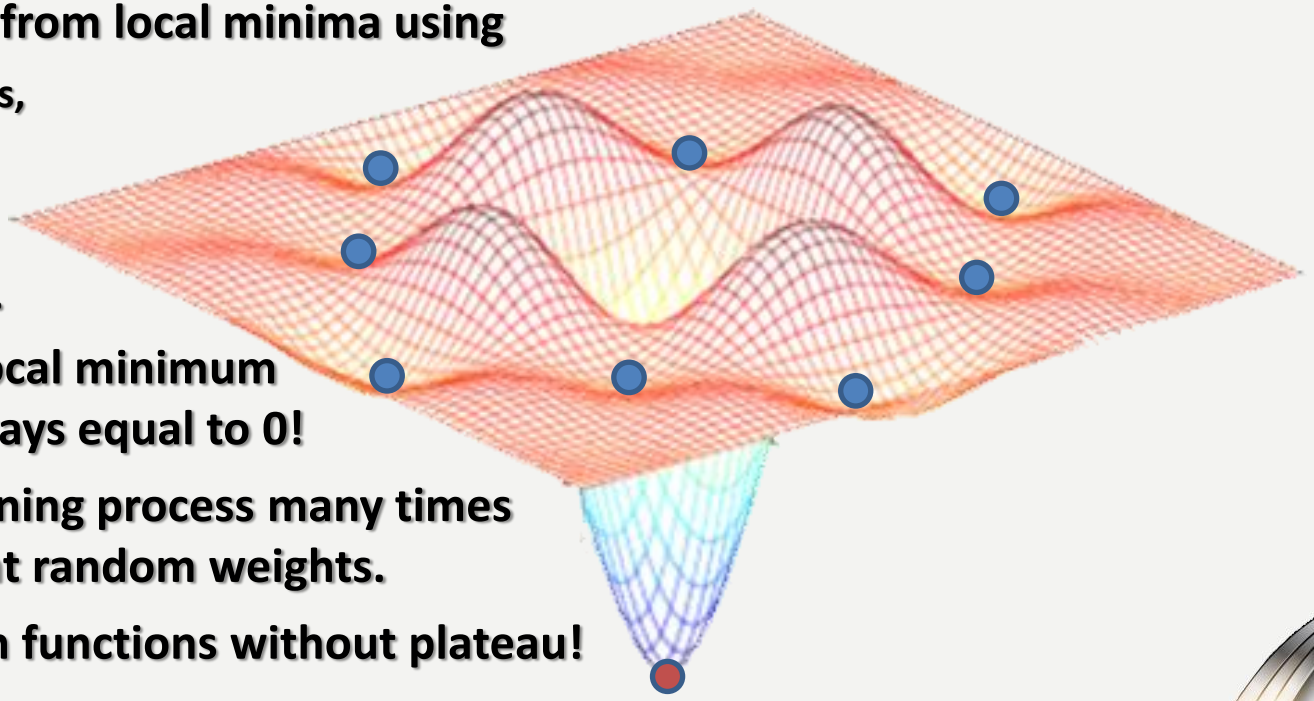
Local Minima and Saddle Points

How to train the network without stacking in local minima or saddle points?

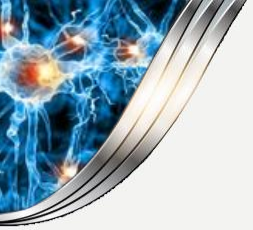
Local Minima vs. Global Minimum

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 or saddle points of the cost function.**
- **We can use such loss functions that are not prone to local minima.**
- **Normalization speeds up the training and better avoids local minima.**
- **We can try to escape from local minima using**
  - ✓ **smaller mini-batches,**
  - ✓ **momentum,**
  - ✓ **RMSprop,**
  - ✓ **Adam optimizer etc.**
- **The gradient in any local minimum or saddle point is always equal to 0!**
- **We can also start training process many times starting from different random weights.**
- **We can use activation functions without plateau!**

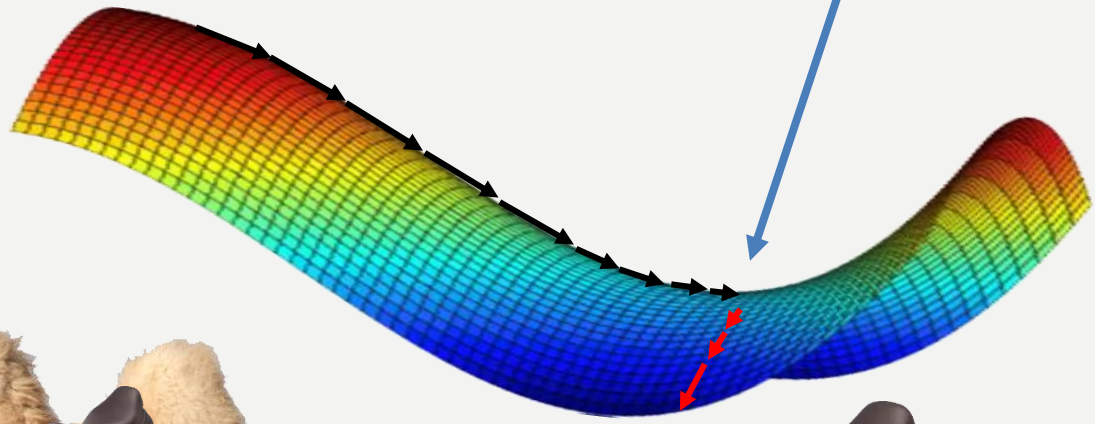


Saddle Points and Plateaus



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 or equal 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.**



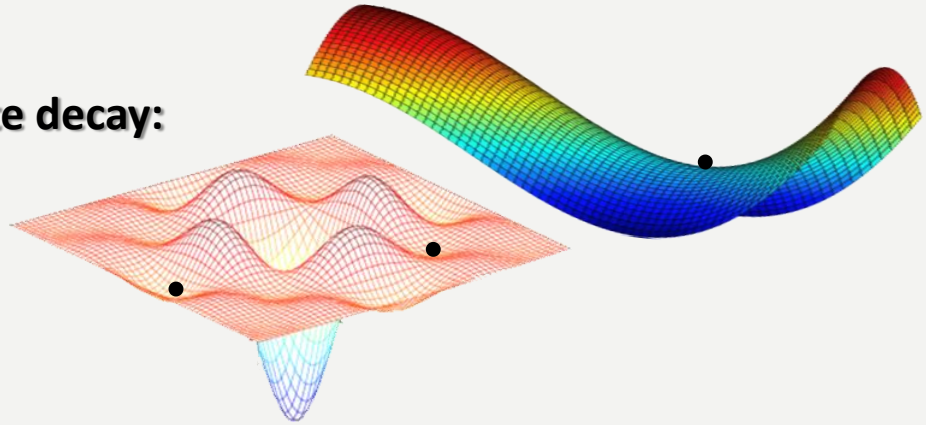
Learning Rate Decay

How to improve training process using on-line adaptation of the learning rate?

Learning Rate Decay

To avoid oscillation close to the minimum of the cost 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:**
  - $\alpha = \frac{k \cdot \alpha_0}{\sqrt{noepoch}}$



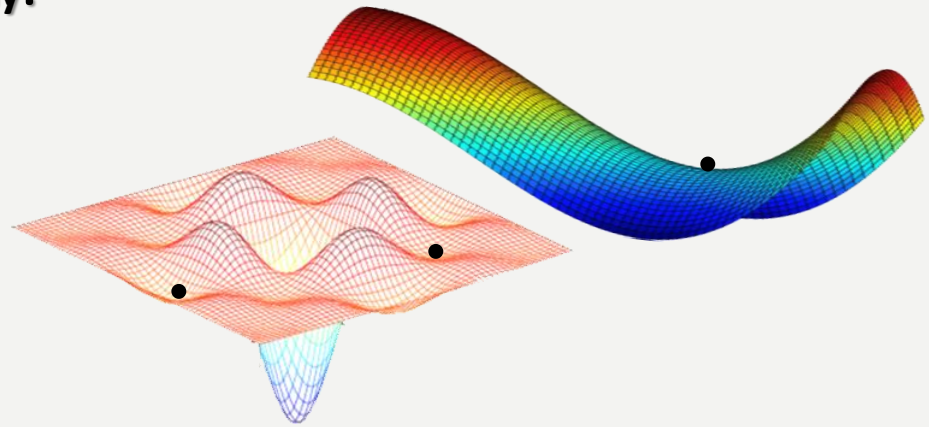
```
# Learning rate reduction during the training process: https://keras.io/callbacks/#reducelronplateau
learning_rate_reduction = ReduceLROnPlateau(monitor='val_acc', # quantity to be monitored (val_loss)
        factor=0.5, # factor by which the learning rate will be reduced. new_lr = lr * factor
        patience=5, # number of epochs that produced the monitored quantity with no improvement after
        verbose=1,  # 0: quiet, 1: update messages.
        min_lr=0.001) # lower bound on the learning rate
```

Learning Rate Decay

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

- **We can also use a staircase decay, decreasing a learning rate after a given number of epochs by half or in another way:**

```
def lr_schedule(epoch):
    lrate = 0.001
    if epoch > 50:
        lrate = 0.0005
    if epoch > 100:
        lrate = 0.0003
    return lrate
```



```
model.compile(loss='categorical_crossentropy', optimizer=opt_rms, metrics=['accuracy'])
model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size),\
                    steps_per_epoch=x_train.shape[0] // batch_size,epochs=125,\
                    verbose=1,validation_data=(x_test,y_test),callbacks=[LearningRateScheduler(lr_schedule)])
```

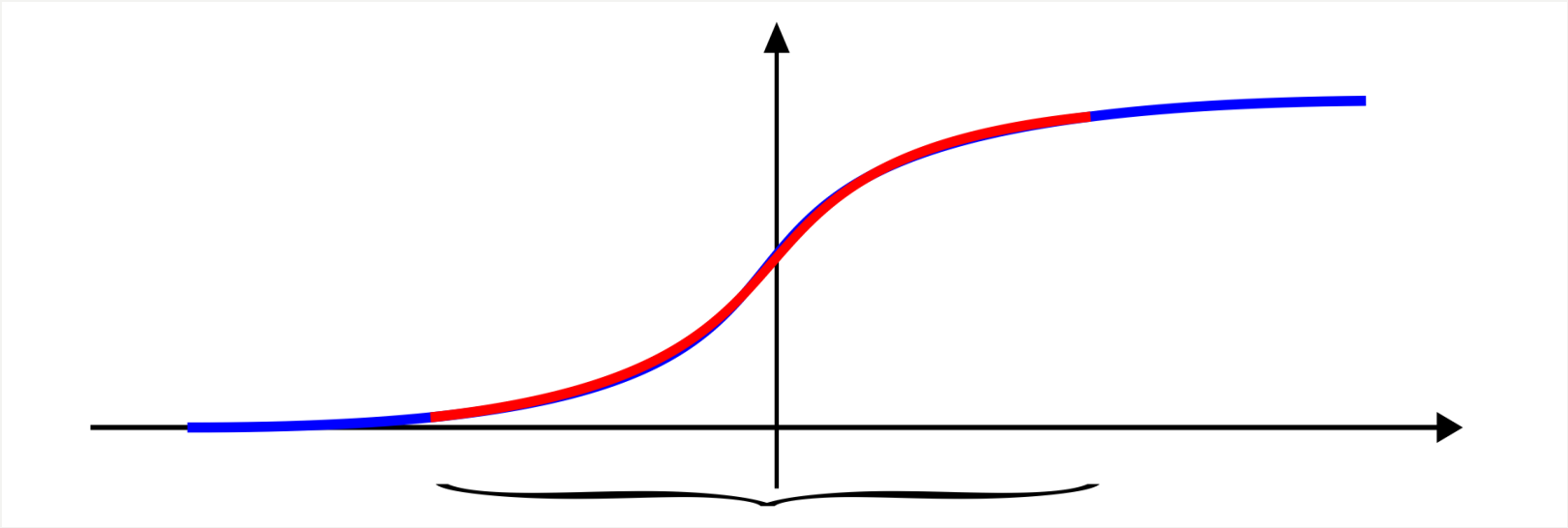
Initialization of Weights

How to accelerate the training process?

Initialization of Weights

We initialize weights with small random values:

- to put the values of activation functions in the range of the largest variance, which speeds 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.

Standardization and Normalization

Make the data similarly influencing on the network and speed up the training process!

Data Standardization

Standardization is an operation commonly used in statistics, which consists in rescaling data of each element of the set against the mean values and standard deviation in accordance with the formula:

$$y_i = \frac{x_i - m}{\sigma}$$

$x = [x_1, x_2, \ldots, x_N]$ – is the N-element vector of the source data,

$y = [y_1, y_2, \ldots, y_N]$ – is the N-element data vector after standardization,

m – is the average value determined from these data,

σ – is the standard deviation.

As a result of **standardization**, we get a vector of features which average value is zero, while the standard deviation is equal to one.

It should not be used for data about standard deviation close to zero!

Data Normalization

Normalization is the data scaling with respect to extreme values (min and max) of a given data vector, usually to the range [0, 1] (sometimes to [-1, 1]) according to the following formula:

$$y_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

$x = [x_1, x_2, \ldots, x_N]$ – is the N-element vector of the source data,

$y = [y_1, y_2, \ldots, y_N]$ – is the N-element data vector after normalization.

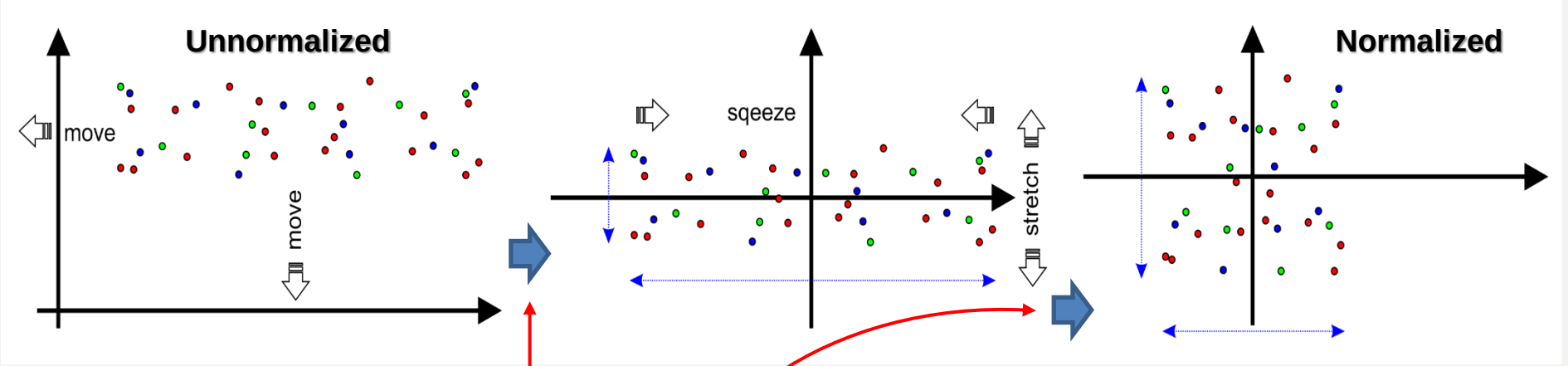
Normalization is sensitive to outliers and large scatter because then the right data will be squeezed in a narrow range, which can significantly hamper their discrimination!

Normalization is sometimes necessary to use a method that requires input or output data to fall within a certain range, e.g. using sigmoidal functions or hyperbolic tangent.

Standardization and Normalization of Training Data Sets

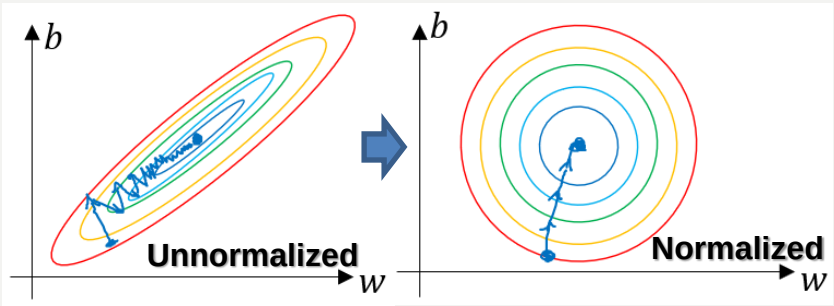
Standardization and normalization:

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



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

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

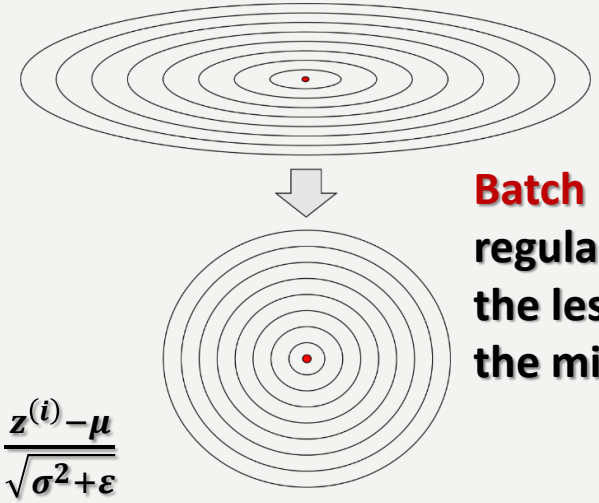


The training process is faster and better when training data are normalized!

Standardization and 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:
- $x^{\{t\}} \rightarrow z^{[1]} \rightarrow \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}\left(\tilde{z}^{[1]}\right) \rightarrow z^{[2]} \rightarrow \tilde{z}^{[2]} \rightarrow a^{[2]} = g^{[2]}\left(\tilde{z}^{[2]}\right) \dots$
- and we apply it usually for $t \in \{1, \dots, T\}$ 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 $\beta^{[l]}$.

Batch Normalization and Standardization

How do we use normalization inside layers in Keras models?

We simply add it before the layer where it should be used when defining the model:

```
model.add(BatchNormalization())
```

It usually helps to improve the model:

```
model.add(Conv2D(128, (3,3), padding='same', kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Conv2D(128, (3,3), padding='same', kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('elu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.4))
```

We can also use standardize our data using the following formulas, which transforms the train and test datasets using mean and standard deviation:

```
mean = np.mean(x_train, axis=(0,1,2,3))
std = np.std(x_train, axis=(0,1,2,3))
x_train = (x_train - mean) / (std + 1e-7)
x_test = (x_test - mean) / (std + 1e-7)
```

Batches, Mini-batches and Training Types

How to speed up the training and avoid local minima and saddle points?

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 (e.g. weights) immediately after the presentation of each training example.
  In this case, training process might be unstable.
- Batch (off-line) training – when we update parameters (e.g. weights) only once 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 training 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):

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

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):

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

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

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:

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

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

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

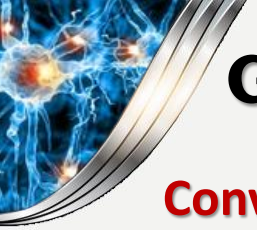
$$Y^{\{1\}} \qquad Y^{\{2\}} \qquad Y^{\{3\}} \qquad Y^{\{m/batchsize\}}$$

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

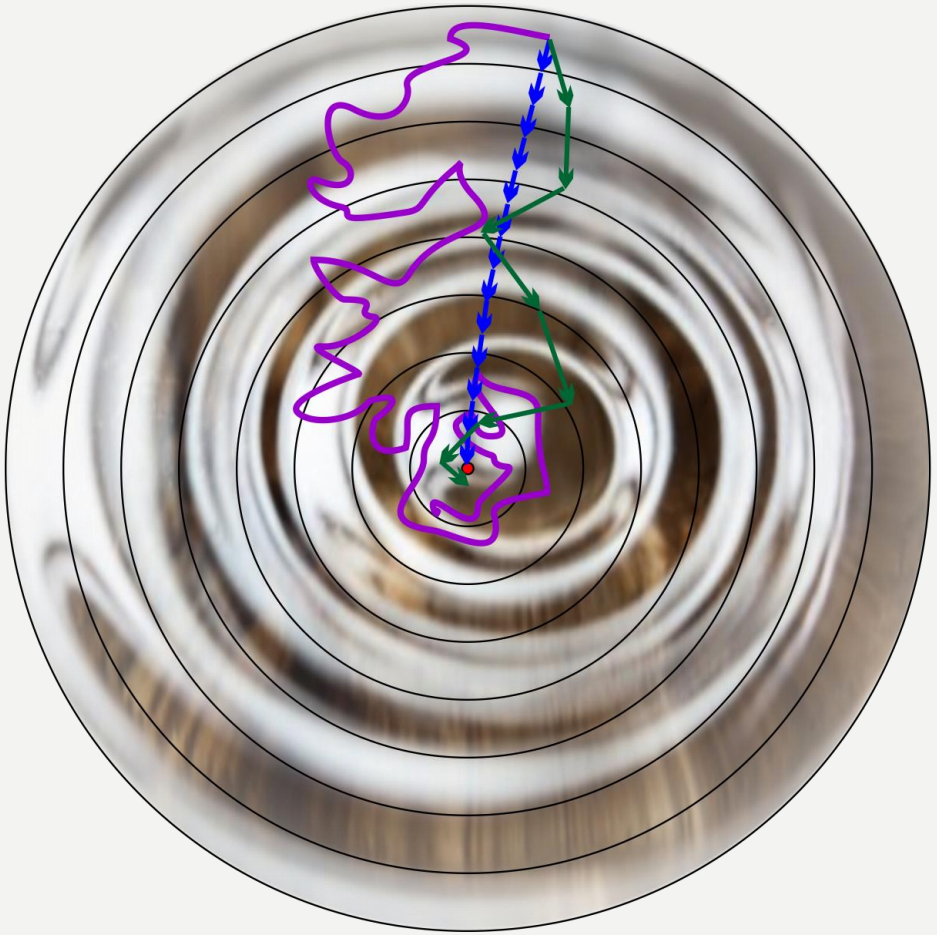
If m= 20.000.000 training examples and the mini-batch size is 1000, we get 20.000 mini-batches (i.e. training steps for each full training dataset presentation, called training epoch), where T = $m/batchsize$.

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

Graphical Interpretation of Mini-batches



Convergence of the training process depends on the size of mini-batches.

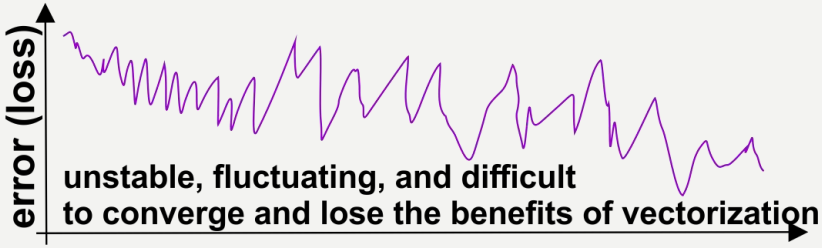


usually very large oscillations close to the minimum

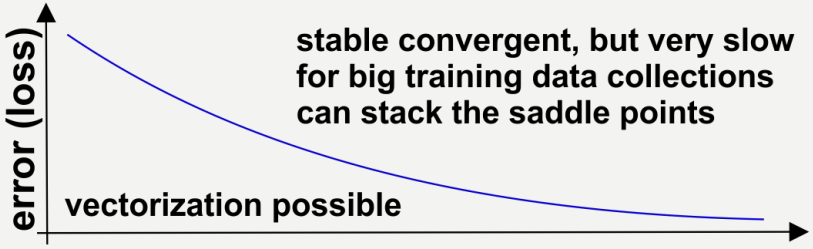
almost no oscillations close to the minimum

possible small oscillations close to the minimum

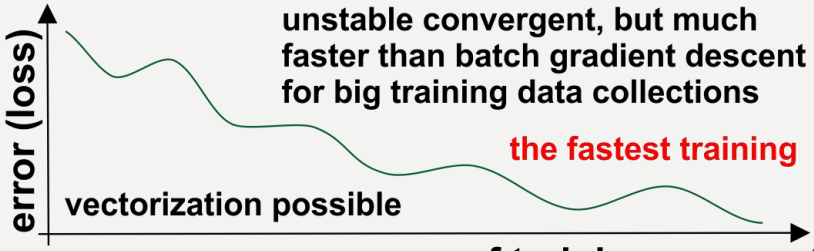
Stochastic Gradient Descent



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/multiplication of parallel cores in the GPU unit, so it is typically a power of two:

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

because then such mini-batches can be processed time-efficiently in one or more parallel steps dependently of the number of parallel cores of the GPU.

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

$dJW_j /= m$

$W_j -= \alpha \cdot dJW_j$

$dJB /= m$

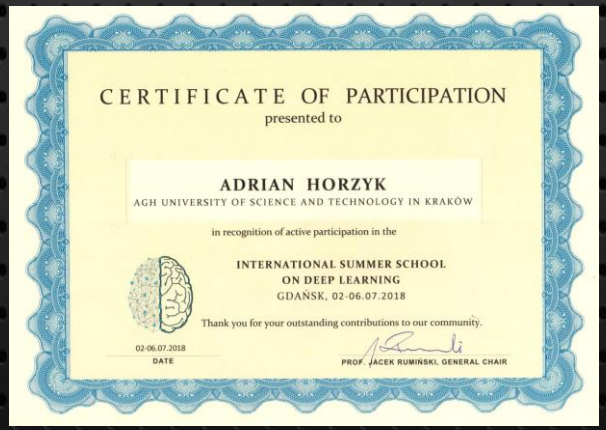
$B -= \alpha \cdot dJB$

BIBLIOGRAPY

1. Francois Chollet, “Deep learning with Python”, Manning Publications Co., 2018.
2. Ian Goodfellow, Yoshua Bengio, Aaron Courville, “Deep Learning”, MIT Press, 2016, ISBN 978-1-59327-741-3.
3. Home page for this course: http://home.agh.edu.pl/~horzyk/lectures/ahdydci.php
4. Nikola K. Kasabov, Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence, In Springer Series on Bio- and Neurosystems, Vol 7., Springer, 2019.
5. Holk Cruse, Neural Networks as Cybernetic Systems, 2nd and revised edition
6. R. Rojas, Neural Networks, Springer-Verlag, Berlin, 1996.
8. Convolutional Neural Network (Stanford)
9. Visualizing and Understanding Convolutional Networks, Zeiler, Fergus, ECCV 2014.

BIBLIOGRAPY

10. IBM: https://www.ibm.com/developerworks/library/ba-data-becomes-knowledge-1/index.html
11. NVIDIA: https://developer.nvidia.com/discover/convolutional-neural-network
12. JUPYTER: https://jupyter.org/



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

PROF. JACEK RUMIŃSKI, GENERAL CHAIR