KNOWLEDGE-BASED COMPUTATIONAL INTELLIGENCE AND DATA MINING AND BIOMEDICINE

Performance Improvement and Hyperparameter Optimization of Deep Neural Networks

Adrian Horzyk

horzyk@agh.edu.pl

AGH

AGH University of Science and Technology Krakow, Poland

Performance of DNN

When training DNN we usually struggle with the improvement of hyperparameters, structures and training models to achieve better training speed and final performance. We can try (some ideas):

- Collect more training data (and label them for supervised training).
- Diversify training data to represent a computational task better.
- Use different network architectures and different numbers of layers and neurons.
- Use different activation functions and different sequences of various layers.
- Experiment with various hyperparameters and try different combinations of them.
- User regularization, dropout, optimization methods (e.g. Adam optimizer).
- Train a chosen network longer with different or changing learning rates.

How to quickly and smarter choose between various training strategies?

We always have limited resources (time and computational power) to solve a given problem and must cut costs in the commercial implementations!

Orthogonalization

Orthogonalization:

- Is a clear-eyed process about what to tune and how to achieve a supposed effect.
- Is the process that let 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 time?
- Have you tried to fly a helicopter and/or a drone in the past? What is your experience?

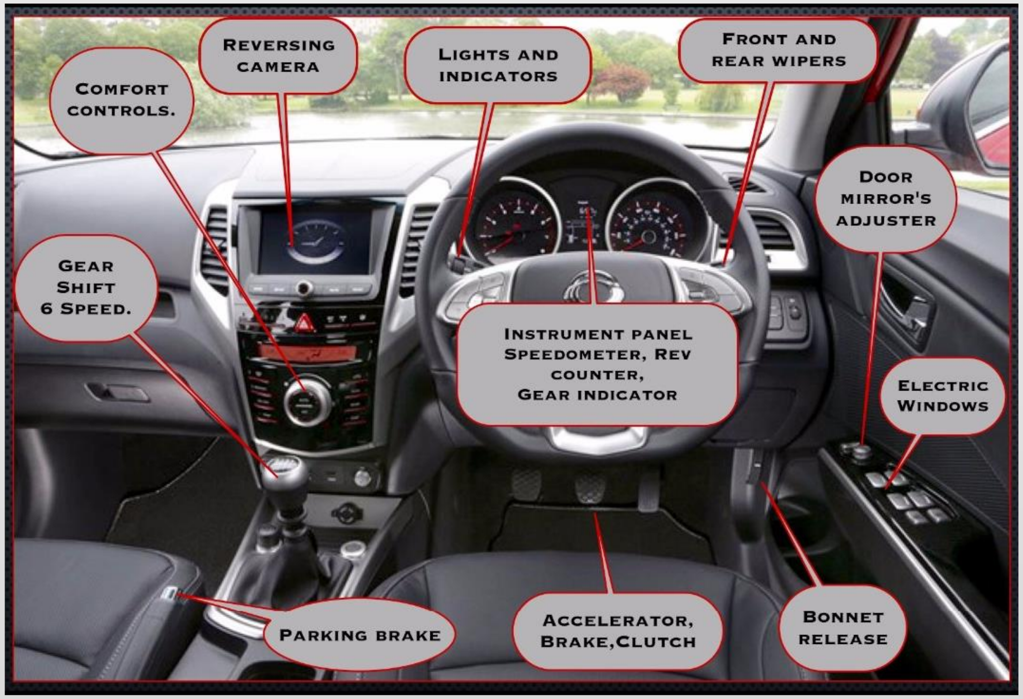
Car Controllers

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?

What 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 joystick) that can control a combination of parameters (like speed and direction) by the same move?



Chain of Machine Learning Goals

When developing and training the model we usually follow the following chain of goals:

1. **Fit a training set well on a cost function trying to achieve the human-level of performance.**
2. **Fit a dev set (validation set) well on a cost function to get good generalization properties verified during the training process.**
3. **Fit a test set well on a cost function to be sure that the generalization is good enough and validated on the data that were not used during the training process.**
4. **Next, we hope that the model will perform well in real world.**

- **If the model does not fit well in any of the first three steps, we need to know what we can do with the model, its hyperparameters and the training to achieve the goal!**
- **Therefore, we want to define knobs (hyperparameters, optimization strategies that can help us) for each step to control the training process to fit the model well.**

Knobs for Tuning the Training Process

In certain steps of the training process, we can use different knobs:

Fit a training set	Fit a dev set	Fit a test set	Fit on real-word data
Bigger network	Smaller network	Bigger dev set	Training data were not representative
Adam optimizer	Regularization		Cost function not enough well defined
Xavier initialization	Bigger training set		
Early stopping	Early stopping		
Add new features	Add new features	Add new features	

Single Number Evaluation Metric

When adapting the network we usually train it with different hyperparameters and comparing 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%

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

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

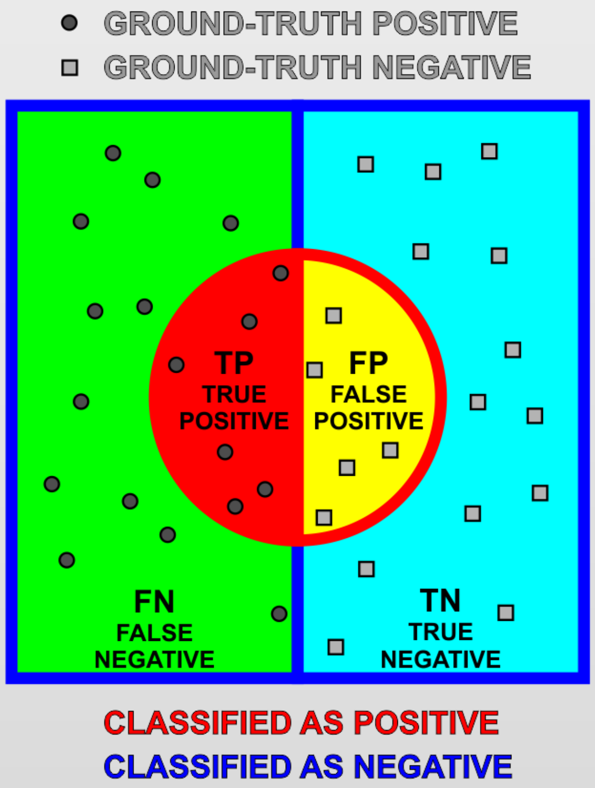
β - how many times recall is more important than precision
TP – true positive, FP – false positive, FN – false negative

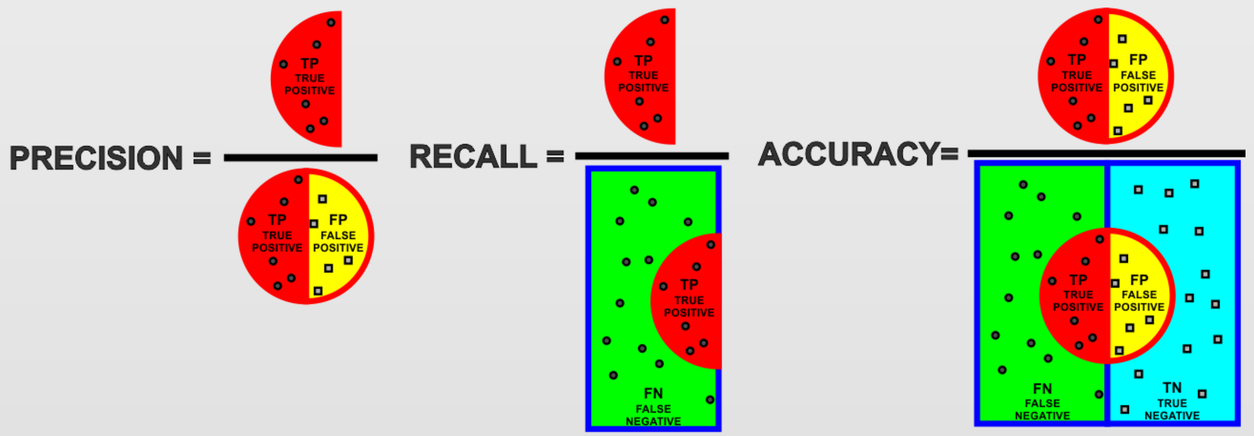
- **Which classifier from the above three is the best one?**
- **It turns out that there is often a trade-off between precision and recall, but we want to care about both of them!**
- **We sometimes use F1 Score that is a harmonic mean of the precision and recall.**

Precision and Recall in view of Confusion Matrix

Confusion Matrix groups the results of classification:

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





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 various criteria to find out the suitable classifier, e.g. we choose this one with the highest accuracy if its classification time is lower than 100 ms.**
- **The accuracy is optimized, while the classification time must be satisfied.**
- **So we have to do with multi-criteria optimization.**

Train, dev, and test sets distribution

Train, dev (validation), and test set should be set up in such a way that they share data of all distributions in the same way:

- When we would like to create a classifier (or another predictor) for data coming from the various data distributions, e.g. the whole world or different countries of the company, we should take care about way how the data are distributed to the train, dev, and test sets. On the other hand, we can train the model almost perfectly on the train data and validate it on the dev set, but it will not work on a test set!

Examples:

- If we train and validate the model on data coming from rich people, it will rather not work for people with low incomes and vice versa.
- If we train and validate the model for men, it will rather not work for women and vice versa.
- If we train and validate the model for data coming from Europe, it will rather not work for data coming from China or US and vice versa.

The train, validation and test target must be the same, i.e. train, dev, and test data must be taken from the same data distributions, i.e. they must be representative for the solved problem.

How to split data to train, dev, and test sets?

Old way of splitting data (for small datasets < 100 thousands), e.g.:

- Train set : dev set : test set = 60% : 20% : 20%
- Train set : dev set : test set = 70% : 15% : 15%
- Train set : dev set : test set = 80% : 10% : 10%

New way of splitting data (for large datasets used in deep learning):

- Train set : dev set : test set = 98% : 1% : 1%
- Because training data today have huge amount of training samples (> 1.000.000), so 1% is enough for validation or testing (1% from 1.000.000 is 10.000 of validation or testing examples), and thanks to it we can use more examples (data) for training!
- The test set should be big enough to give high confidence in the overall performance of the trained system or solved task.

Different Treatment of Training Examples

Training examples might be treated in the same or different way, i.e. they can influence the training process with different strength:

- **We can modify the definition of the error function in such a way to add the strengthening factor $s^{(i)}$ for each training example to let it influence on (impact) the training process with a different strength:**
- $J(w,b) = \frac{1}{\sum_{i=1}^{m} s^{(i)}} \sum_{i=1}^{m} s^{(i)} \cdot L\left(a^{(i)}, y^{(i)}\right)$
- **In this way, we can avoid some unwanted classifications.**

Data attributes can also have different values for the training process (e.g. we want the gender not to influence the training process much), so we can bind different strengths with different attributes, weakening these which should have reduced influence on the classification process and strengthening those which are especially important from the classification point of view.

Sometimes we even avoid using some attributes like race, sex, age, disabilities, health condition, political or religious affiliation etc. not to discriminate some groups of people or other objects because of the law or equal opportunities.

Human-level Error and Perfromance

Suppose that we try to classify some medical images:

The classification can be made by different humans or their teams:

Classification made by:		Produces the error
(a)	Typical human	25%
(b)	Typical doctor (expert)	4%
(c)	Experienced doctor (expert)	1%
(d)	Team of experienced doctors (experts)	0.4%

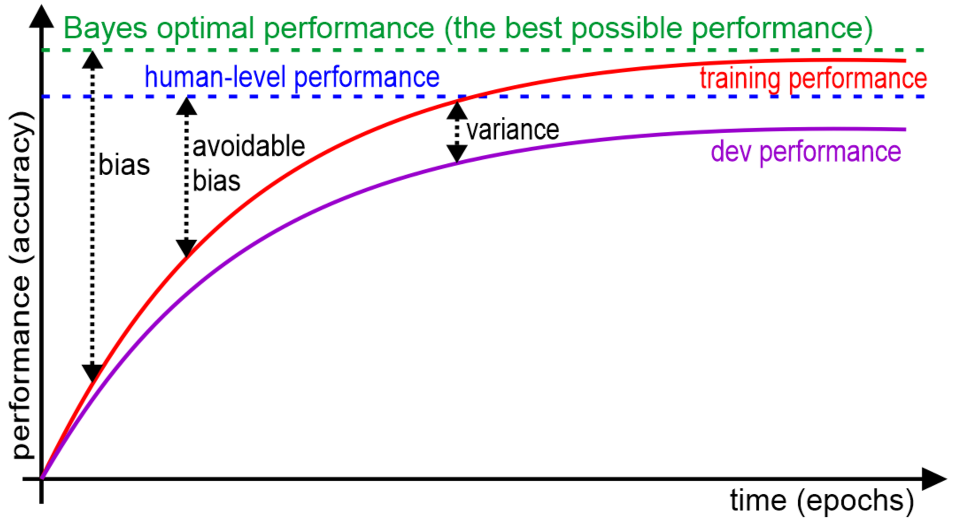
Human-level error (d) is defined as the lowest possible error that might be achieved by the human team of the best experienced experts.
We assume that nobody contemporarily can do it better!

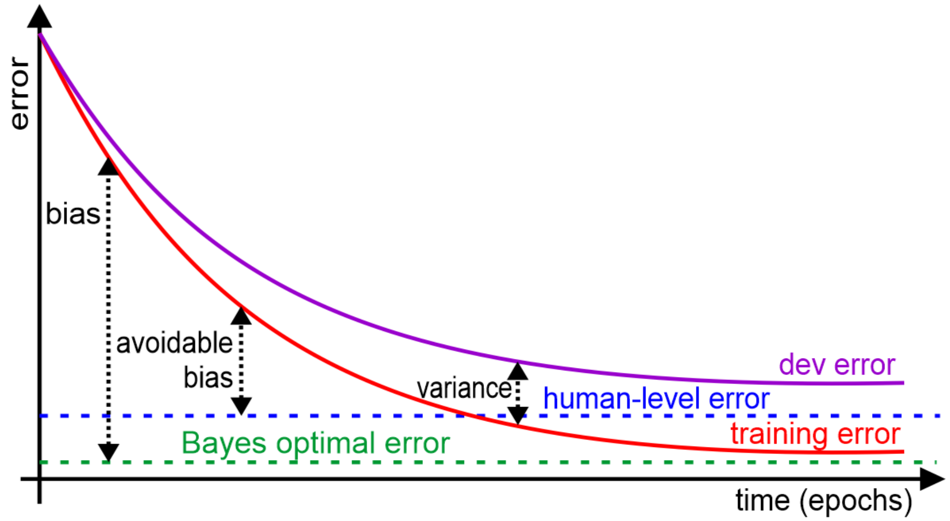
Human-level Performance

Avoidable bias is an error defined by the difference between the training error and the human-level error:
avoidable bias = **training error** – **human level error**

Bias is an error defined by the difference between the training error and the Bayes error: **bias** = **training error** – **Bayes level error**

Variance is an error defined by the difference between the dev error and the training error: **variance** = **dev error** – **training error**





Human level-error and performance are defined by the human team of the world's best experts:

- Then cannot be contemporary surpassed by any human or the human team.
- If they would be surpassed in the future, then they automatically set a new human-level performance and a new human-level error.

Bayes optimal (the best possible) error and performance are defined by the blurred and noisy training examples that nobody and nothing can recognize or differentiate them due to the low quality:

- They can be never surpassed: Bayes optimal error ≤ Human-level error and Bayes optimal performance ≥ Human-level performance
- The Bayes optimal performance can be higher than a human-level performance.
- The Bayes optimal error can be lower than a human-lever error.
- It is many times very close to the human-level performance, where humans are very good at.
- Sometimes it is equal to the human-level performance when data are labelled by humans, so we cannot surpass this level in these cases.

Error Analysis of the Model

Analysis of the collected results and error levels allows us to look for a better solution by the implementation of some tips and tricks to improve the model performance.

Let's analyse a few examples:

Error Type	Model A	Model B	Model C	Model D
Bayes error*	0.5%	0.5%	0.5%	0.5%
Human-level error	0.7%	0.7%	0.7%	0.7%
Bias / Avoidable Bias	3.0% / 2.8%	3.0% / 2.8%	0.5% / 0.3%	0.3% / 0.1%
Training error	3.5%	3.5%	1.0%	0.8%
Variance	4.3%	0.8%	4.0%	0.2%
Dev error	7.8%	4.0%	5.0%	1.0%
Conclusion:	**High Bias High Variance First focus on Bias**	**High Bias Low Variance Focus on Bias**	**Low Bias High Variance Focus on Variance**	**Low Bias & Variance Quite well-trained model**

* Bayes error is not always known because the human-level abilities sometimes disallow to determine it. It might be experimentally determined or known due to the constructed training data set.

Tips and methodology:

We should not try to decrease variance if the avoidable bias is still high.
First, we should always decrease bias and when it is low enough, start decreasing variance.

Minimizing Bias and Variance

During the model development and tuning, we try to minimize bias and variance.

Bias tells us about the ability of the model to adapt to the training data.

- **Bias is the basic evaluation of the quality of the constructed model. If bias is poor, the model should be reconstructed and/or we should use bias-rising methods to minimize it.**

Variance defines the generalization property of the model and tells us how well it can generalize about training data, dealing with a dev set and probably also with a test set and real-world data.

- **If variance is poor, the constructed model is useless, because the main goal of machine learning is the generalization that allows us to use the trained models to real-world data.**
- **We should reconstruct the model and/or adapt variance-rising methods to achieve smaller variance.**

Human-level performance reflects the quality of training data and the wisdom and experience of the humanity to solve the problem of a given kind.

- **If the training data quality is poor or training data are contradictory, we cannot achieve better performance because nobody (event the team of human experts) can do the given task better.**
- **We can try to rise the quality of the training data (train set) to rise the human-level performance.**
- **Sometimes training data are described/defined by too small subset of attributes that do not allow to differentiate them enough (ambiguity producing contradictions) and to discriminate them in the classification process. In this case, we should redefine the train set adding new attributes (features) that describe the train objects in such a way that the diversity of them allow us to discriminate them.**

Surpassing Human-level Performance

Generally, it is not easy to surpass the human-level performance, especially for various perceptions, speech and image recognition etc.

Surpassing human-level performance is possible for many problems:

- **Product recommendations**
- **Online advertising**
- **Predicting transit time in logistics**
- **Loan approvals**
- **Many big data problems where humans cannot analyse them**
- **Non-natural perception tasks that were not evolved in humans over millions of years**
- **Various structural data requiring complex comparisons and analysis**
- **etc.**

It occurs when the achieved training error (e.g. 0.3%) is less than the human-level error (e.g. 0.5%).

It may be difficult to establish how much the human-level performance might be overpassed and what would be the final performance and the bias of training that could be still avoided.

Guidelines for Minimizing Errors

What can we do when the quality indicators of the model are low?

What are the guidelines for minimizing errors and improving performance?

Bayes error or Human-level error is high	Training error is high Bias is high	Dev (validation) error is high Variance is high
Clean data (remove possible bugs, inaccuracy, blurred data, outliers etc.)	Better or bigger network architecture (more parameters to enrich the too simple model)	Better or smaller network architecture (less parameters avoiding overshooting)
Add discriminating attributes	Xavier initialization	Regularization (L2, dropout)
Remove ambiguity (contradictory training data)	Use better optimization algorithms (e.g. Adam, RMSprop, momentum)	Bigger training set of the same distributions as the dev set
Reconstruct training data	Train longer (more epochs)	Early stopping
	Transfer learning	Transfer learning
	Better hyperparameter (network structure) search	Better hyperparameter (network structure) search
		Data augmentation

Error Analysis and Overgoing Troubles

When you train the network, trying to implement various tips and tricks, but you are still unsatisfied of the achieved results, you can try to analyse results, e.g. incorrectly classified examples, and overgo these troubles implementing special routines into them:

- Check to which classes belong incorrectly classified examples? Are they of one or more classes? Do one class patterns prevail in them or not?
- Focus your effort on the most numerous incorrectly classified examples of one class because it can help you to decrease the error the most (ceiling) if you succeed.
- Are trained classes represented evenly in the training set? If not, try to balance the size of all classes, e.g. using augmentation to the less numerous classes or to reduce unevenly the learning rates implemented to various classes taking into account the number of examples which represent them.
- You can try to strengthen the training process for the incorrectly classified examples, e.g. use different learning rates for various training examples, i.e. the bigger learning rates for examples that are difficult to train.
- Check what the neurons of the network represent and whether the classification is not based on the object surrounding instead of the classified object self.
- Finally, try to find out all possible categories of errors and count up their occurrences:

Example	Too big	Blurry	Mislabeled	Cars	Data Distribution 1	Weak representation of this class	Comments
1		✓	✓				
2				✓		✓	
…					✓		
% of total:	15%	42%	18%	32%	12%	18%	

Cleaning and Correcting Mislabeled Data

Deep learning algorithms are usually robust, so the random errors and mislabeled training data should not spoil much the training process, but if there is a lot of incorrectly labeled data, they should be corrected:

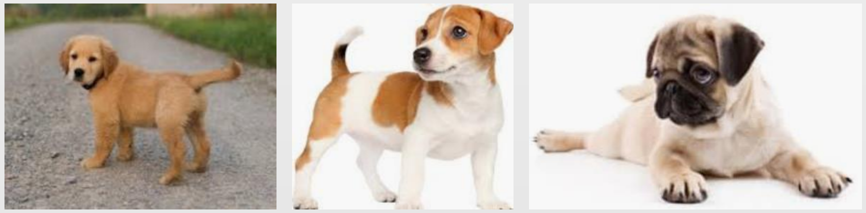
- **How to correct the training set when it consists of thousands/millions of examples?**
- **If the number of mislabeled examples is not too big (> 10%), we can try to learn the model using all correctly and incorrectly labeled examples, then filter out all misclassified examples and correct or remove those which are mislabeled, next, continue or start the training process from scratch again and again until we correct enough mislabeled data and achieve satisfying results of training the model.**
- **We can also use unsupervised training method to cluster training data, next, in each cluster, filter out all differently labeled examples to the most numerous class(es) represented be each cluster, and correct the mislabeled examples.**
- **If training data contain blurry or misleading examples, we can also remove them from the training set (cleaning it). Such examples are removed during the error analysis of the filtered out incorrectly classified examples. After removing of such examples, we start the training process again and again until we remove enough poor-quality examples and achieve satisfying results of training the model.**

Combining Data from Different Distributions

When training the model using data from different distributions, we should construct training, dev and test sets from all distributions!

EXAMPLE (a possible data mismatch problem):

High-quality data distribution



Low-quality data distribution

Distribution 1: 100000 examples

Distribution 2: 20000 examples

Take the data from both distributions together and shuffle them:

Distribution 1+2: 120000 examples

Next, split them into training, dev and test sets:

Training | Dev | Testing

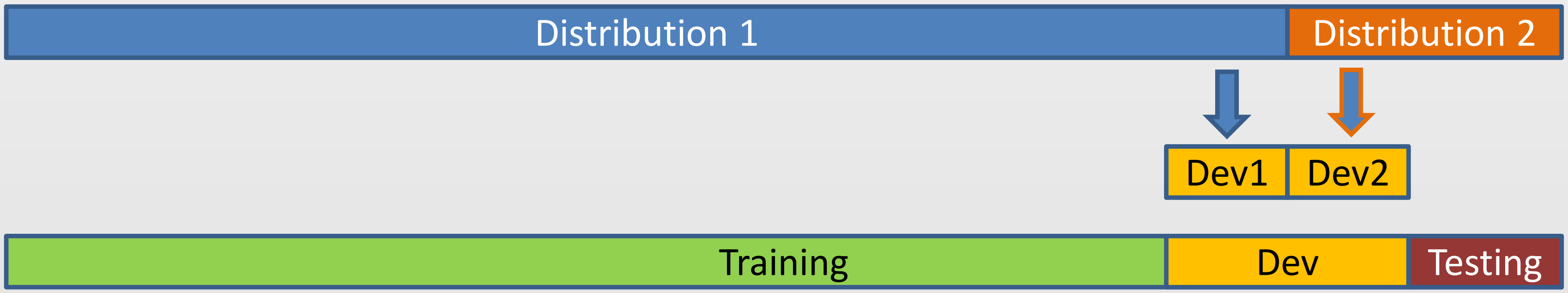
If you don't put the data together and shuffle them, they might be trained, validated, and tested on different distributions and dev and test results might be very poor:

Distribution 1 | Distribution 2

Training | Dev | Testing

Identifying Data Mismatch Problem

If the dev set is composed from various distributions, but the training set is taken only from one distribution (or a subset of distributions), then dev error on the subsets of dev sets will differ!



In this case, we usually achieve the following:

human-level error < training error < dev1 error < dev error < dev2 error < testing error

This indicates the data mismatch problem, i.e. the model has been trained on a limited subset of distributions (not all distributions).

The difference between testing error and dev error indicates the overfitting problem.

Overcome the Data Mismatch Problem

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

- We can add typical noise to training data.
- Blur training data.
- Add some distortions to training data.

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

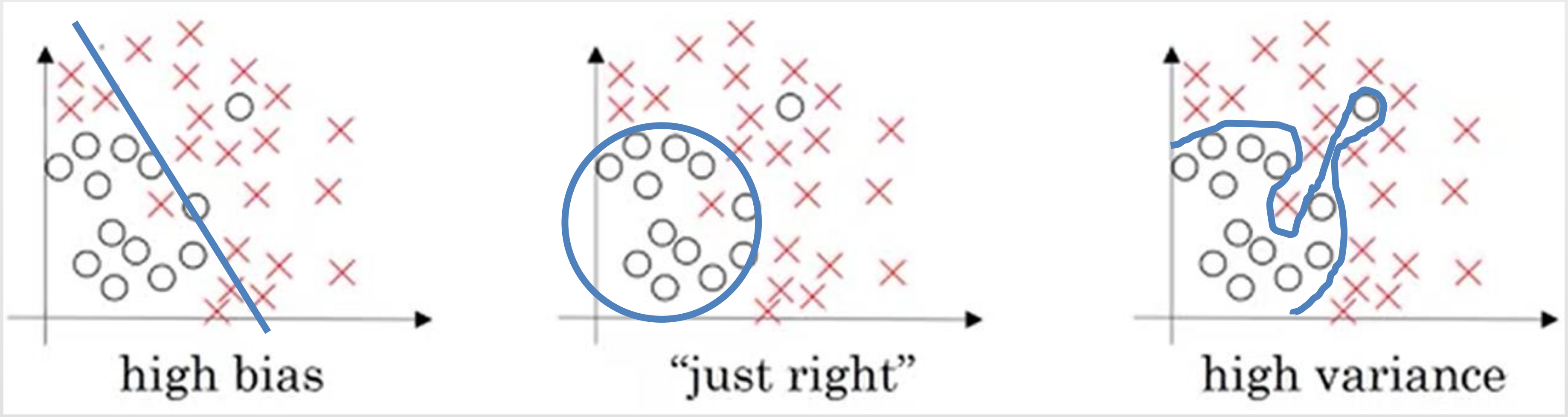
Test examples for checking generalization of the trained model

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.
- 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”.

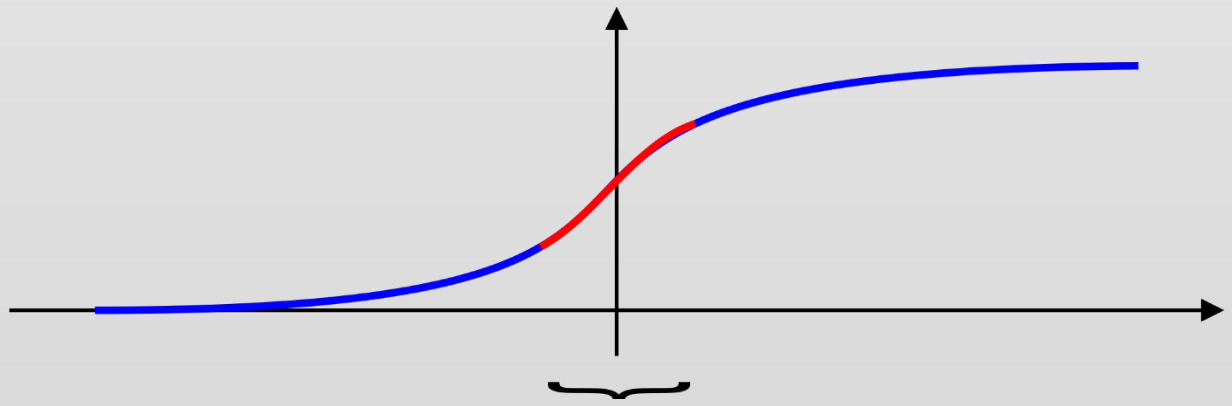
Regularization prevents overfitting

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

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

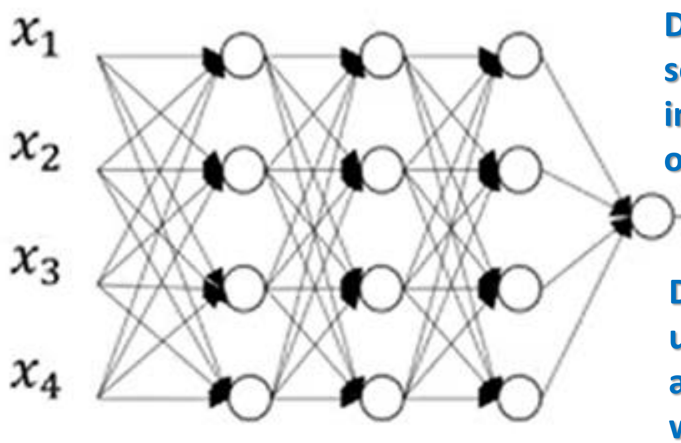
because we want to minimize the above cost function during the training! So the network will compose of nearly linear (not very complex) functions.

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:



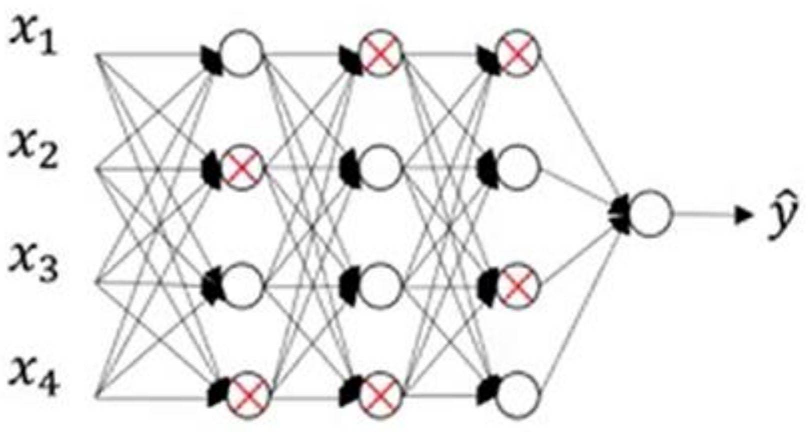
Dropout Regularization

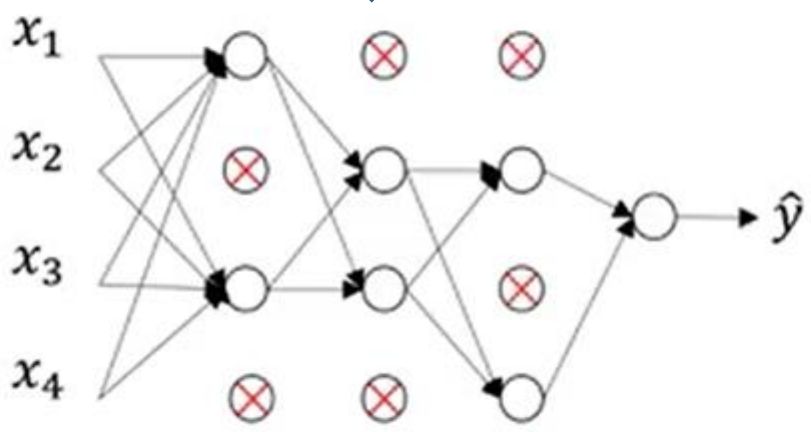
Dropout regularization switches off some neurons with a given probability, not using them temporarily during propagation and backpropagation steps forcing the network to learn the same by various combinations of neurons in the network:



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.



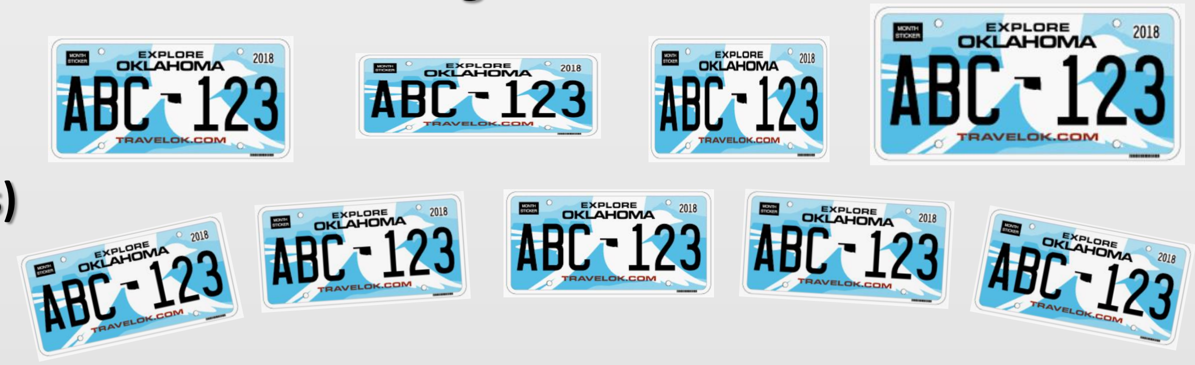


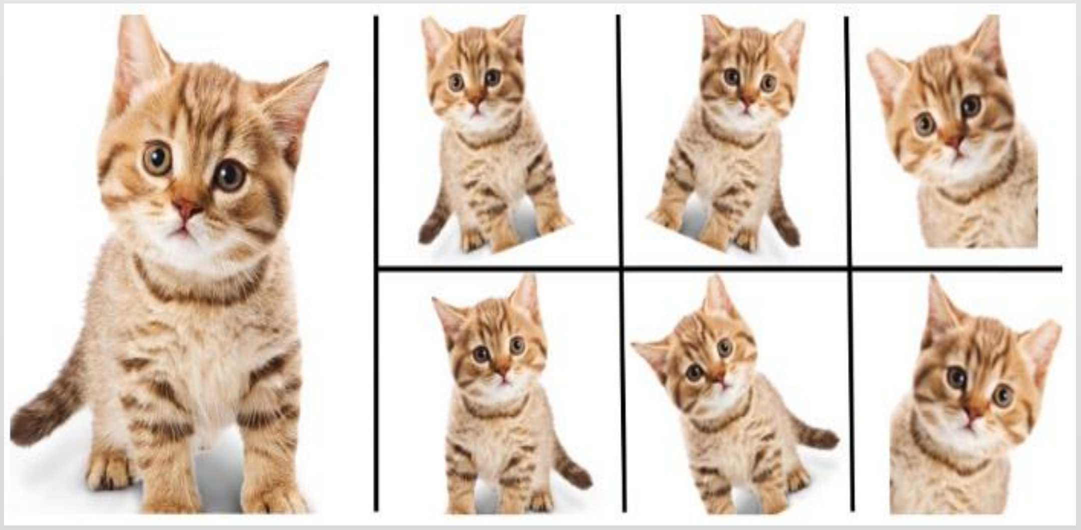
Implementing dropout regularization the input stimuli of neurons are weaken according to the number of the shut off neurons (i.e. the chosen probability of dropout on average, e.g. p = 0.25), so the stimulation must be higher to achieve the right stimulation of the neurons, e.g. the classification neurons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

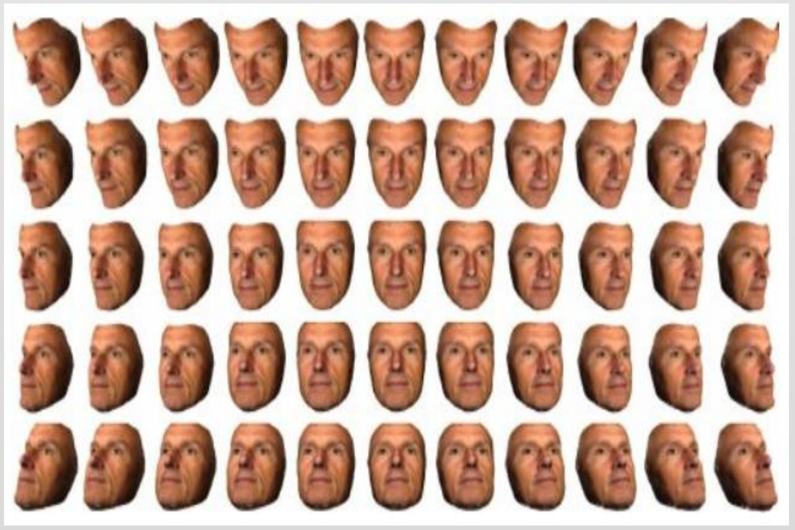
Data Augmentation

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**

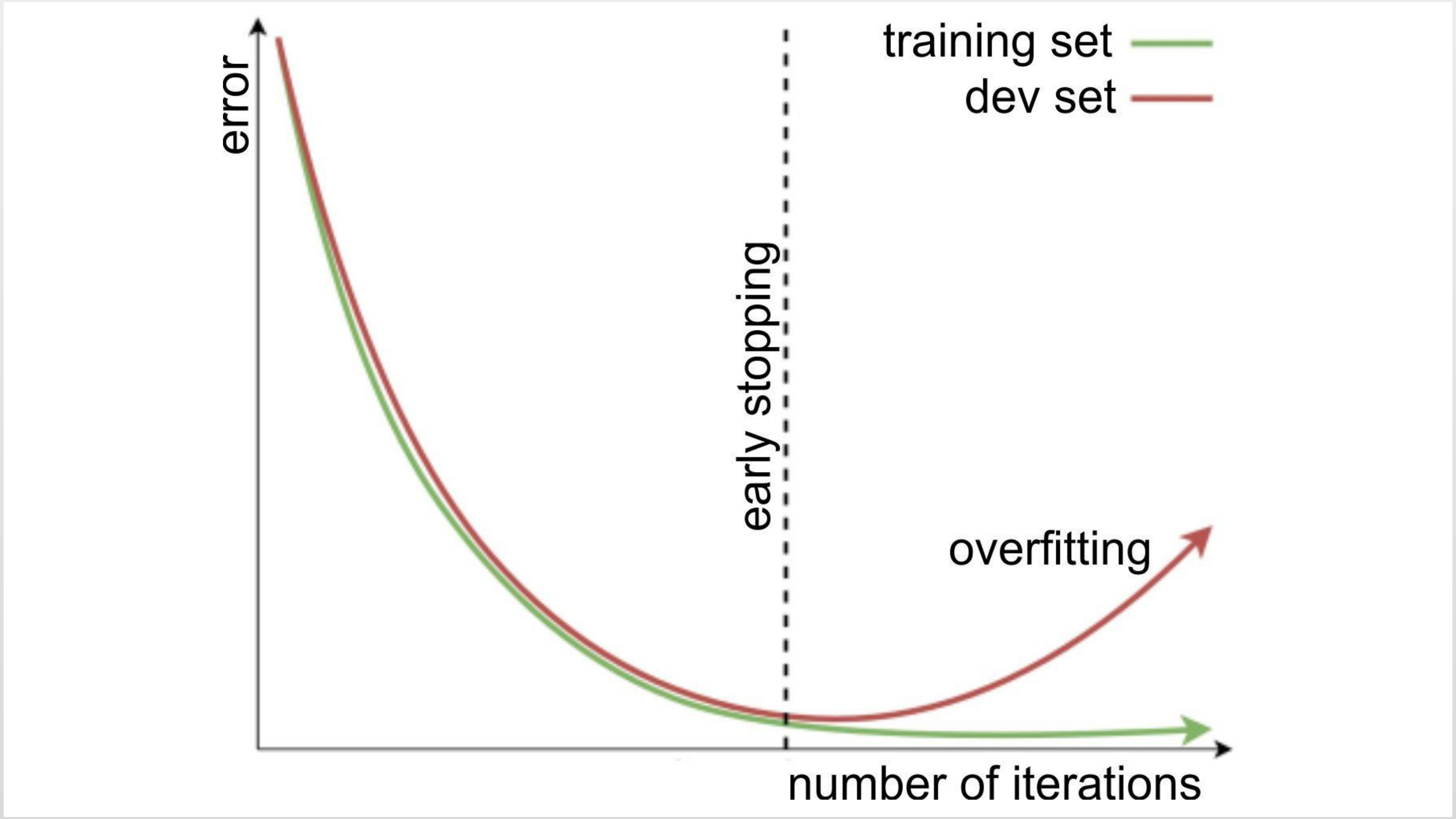






Early Stopping

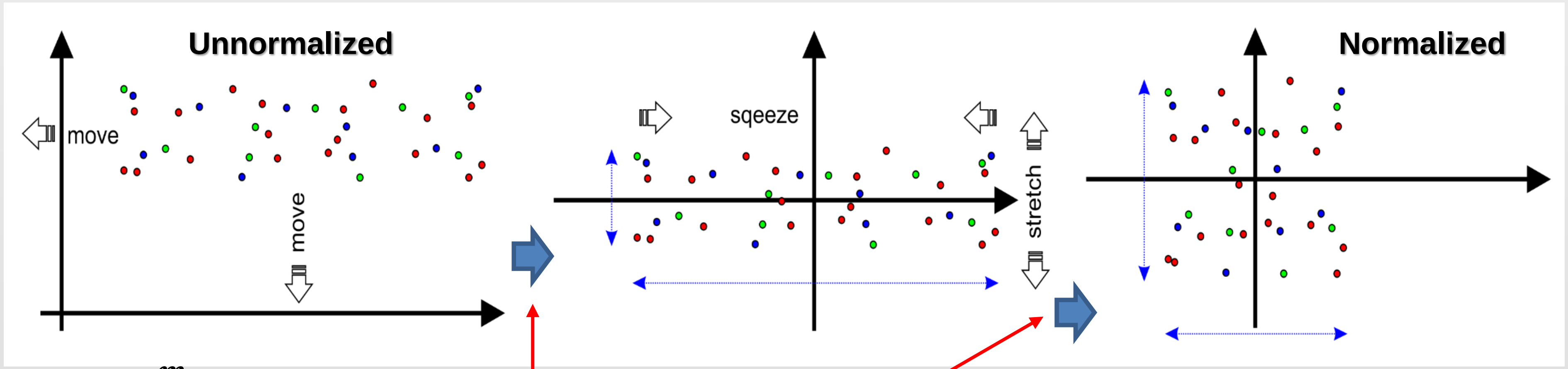
We can also use "early stopping" of the training routine before the error on the dev set starts to grow:



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.

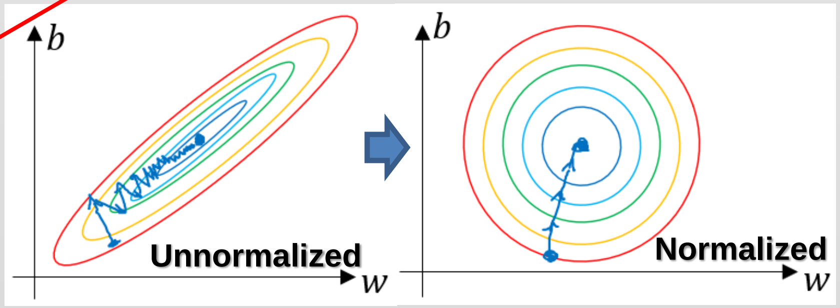


$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$x := x - \mu$$

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

$$x := x/\sigma$$

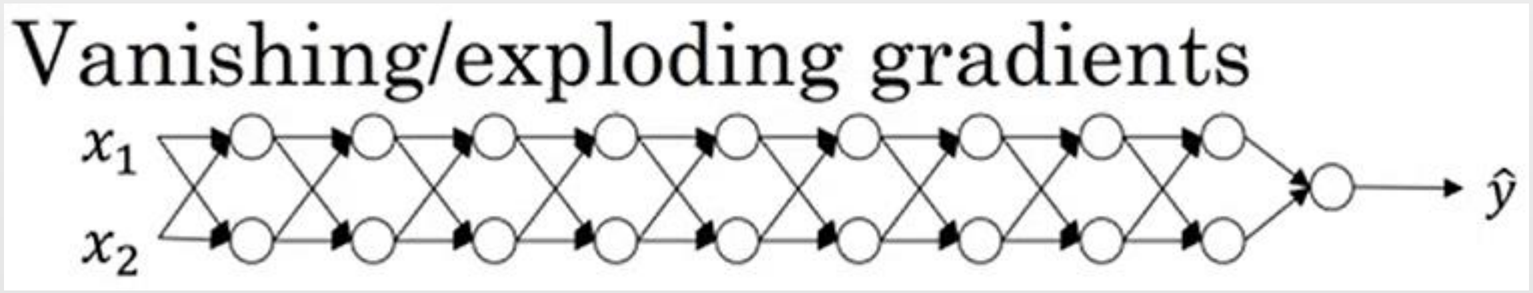


- The training process is faster and better when training data are normalized!

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**

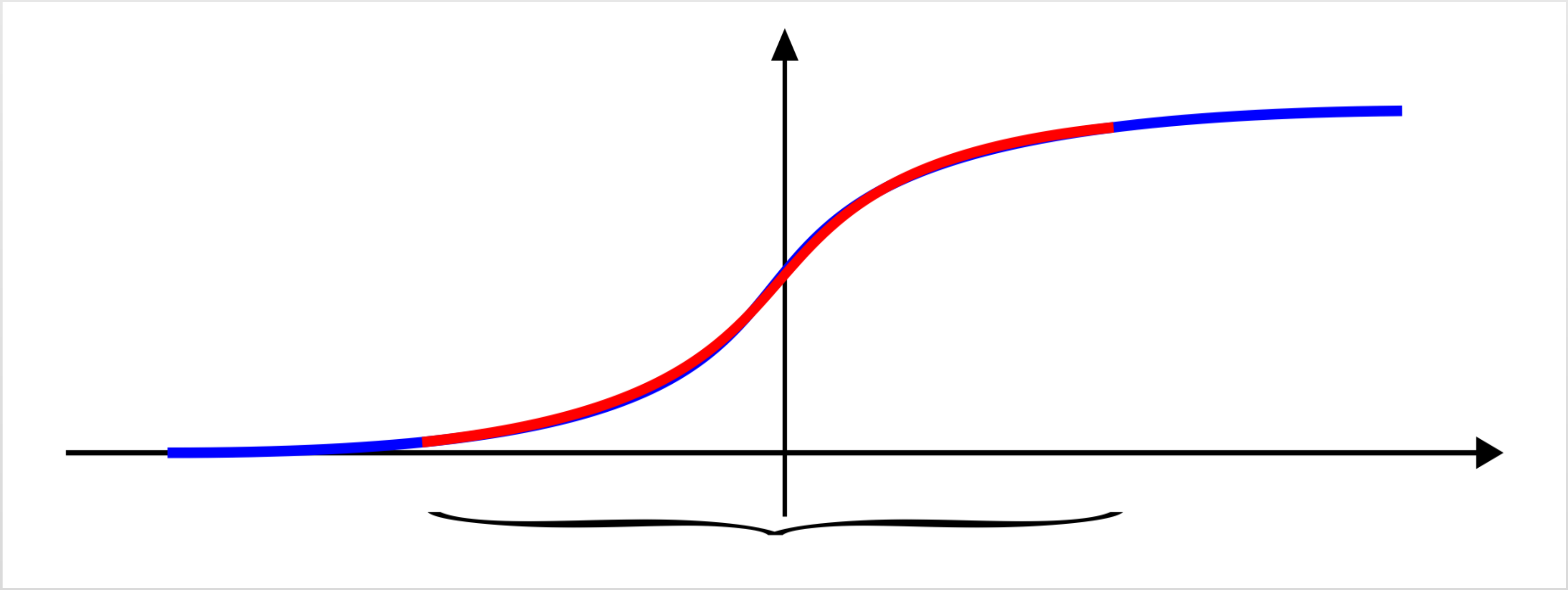


because today we use deep neural networks that consist of tens of layers!

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):

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

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):

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

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

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:

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

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

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

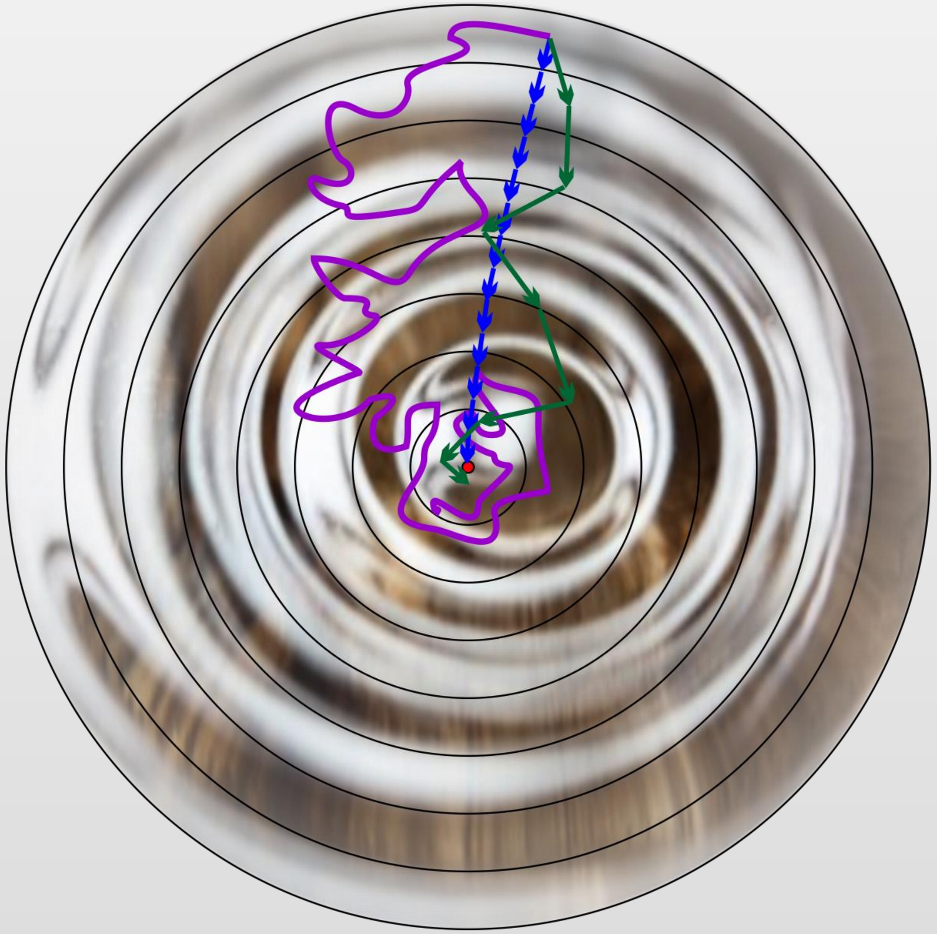
$$\boldsymbol{Y}^{\{1\}} \qquad \boldsymbol{Y}^{\{2\}} \qquad \boldsymbol{Y}^{\{3\}} \qquad \boldsymbol{Y}^{\{m/batchsize\}}$$

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

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

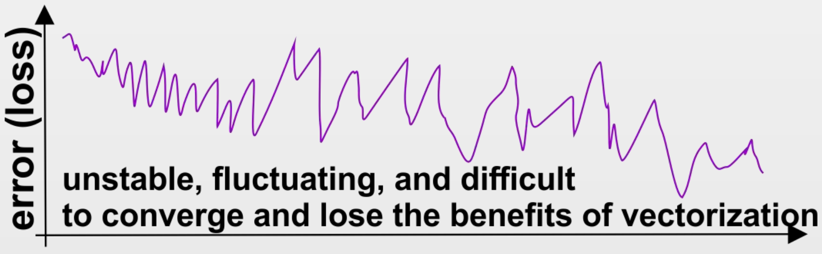


usually very large oscillations close to the minimum

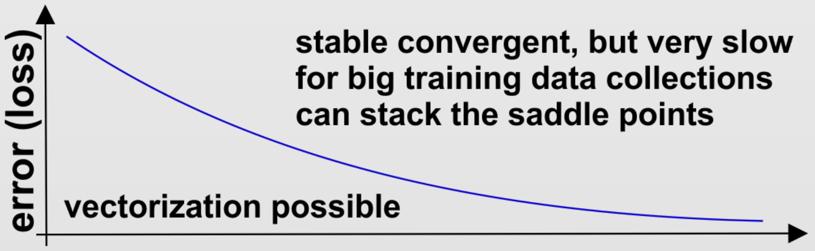
almost no oscillations close to the minimum

possible small oscillations close to the minimum

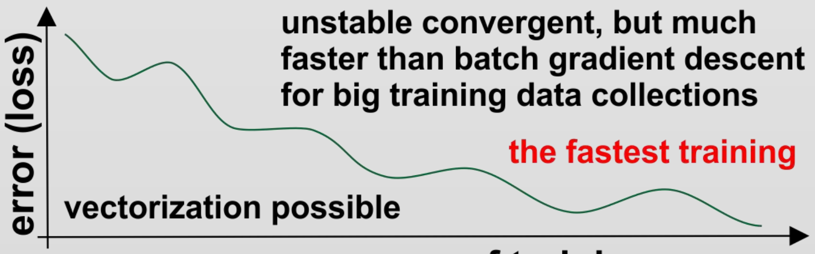
Stochastic Gradient Descent



Batch Gradient Descent



Mini-Batch Gradient Descent



convergence of training

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.

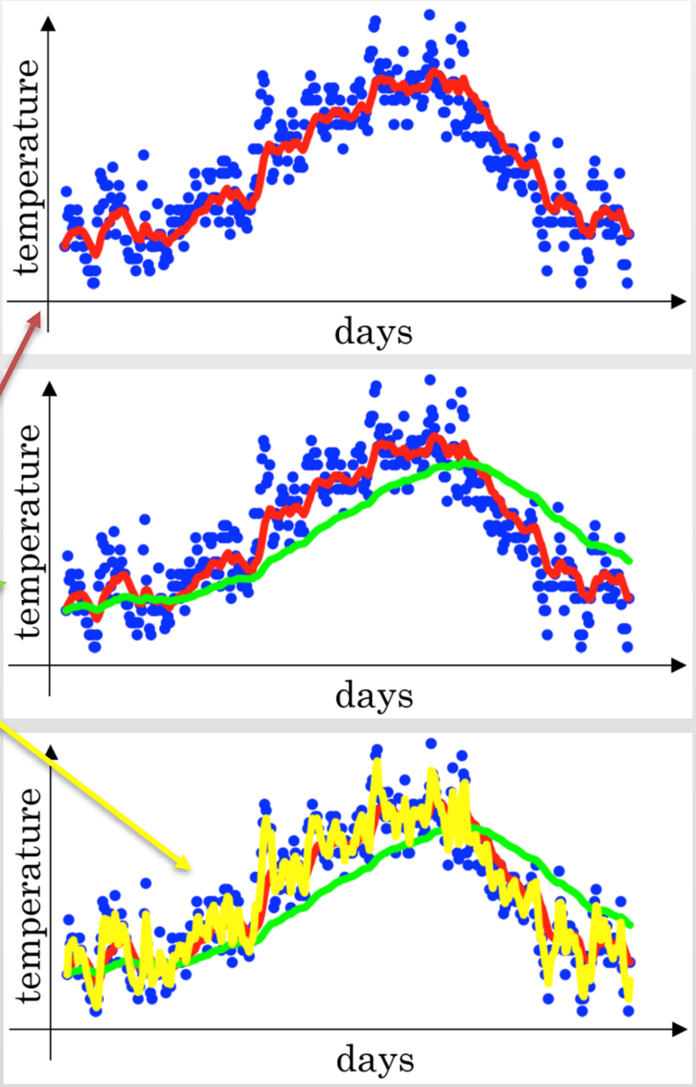
Mini-batches are used for big training dataset (ten or hundred thousands and millions of training examples) to accelerate computation speed.

```
repeat
  $J = 1$
  for $j = 1$ to $n_x$
    $dJW_j = 0$
  $dLB = 0$
  for $t = 1$ to $T$
    $Z^{\{t\}} = W^T X^{\{t\}} + B$
    $A^{\{t\}} = \sigma\left(Z^{\{t\}}\right)$
    $J += -\left(Y^{\{t\}} \, log\, A^{\{t\}} + \left(1 - Y^{\{t\}}\right) log\left(1 - A^{\{t\}}\right)\right)$
    $dJZ^{\{t\}} = A^{\{t\}} - Y^{\{t\}}$
    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):

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

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

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

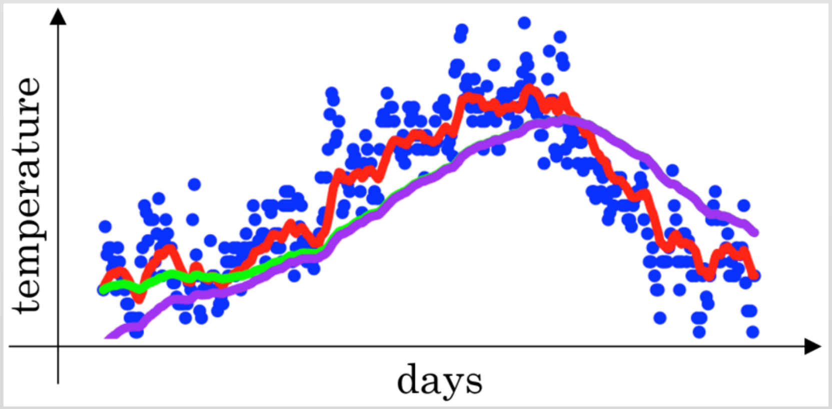
To avoid this, we use the correction factor (green curve) $1 - \beta^t$:

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

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

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

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.
- $W := W - \alpha \cdot v_{dw}$
- $b := b - \alpha \cdot v_{db}$
- $v_{dW} := \underbrace{\boxed{\beta}}_{\text{friction}} \cdot \underbrace{\boxed{v_{dW}}}_{\text{velocity}} + \underbrace{\boxed{(1-\beta) \cdot dW}}_{\text{acceleration}}$
- $v_{db} := \underbrace{\boxed{\beta}}_{\text{friction}} \cdot \underbrace{\boxed{v_{db}}}_{\text{velocity}} + \underbrace{\boxed{(1-\beta) \cdot db}}_{\text{acceleration}}$
- 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.

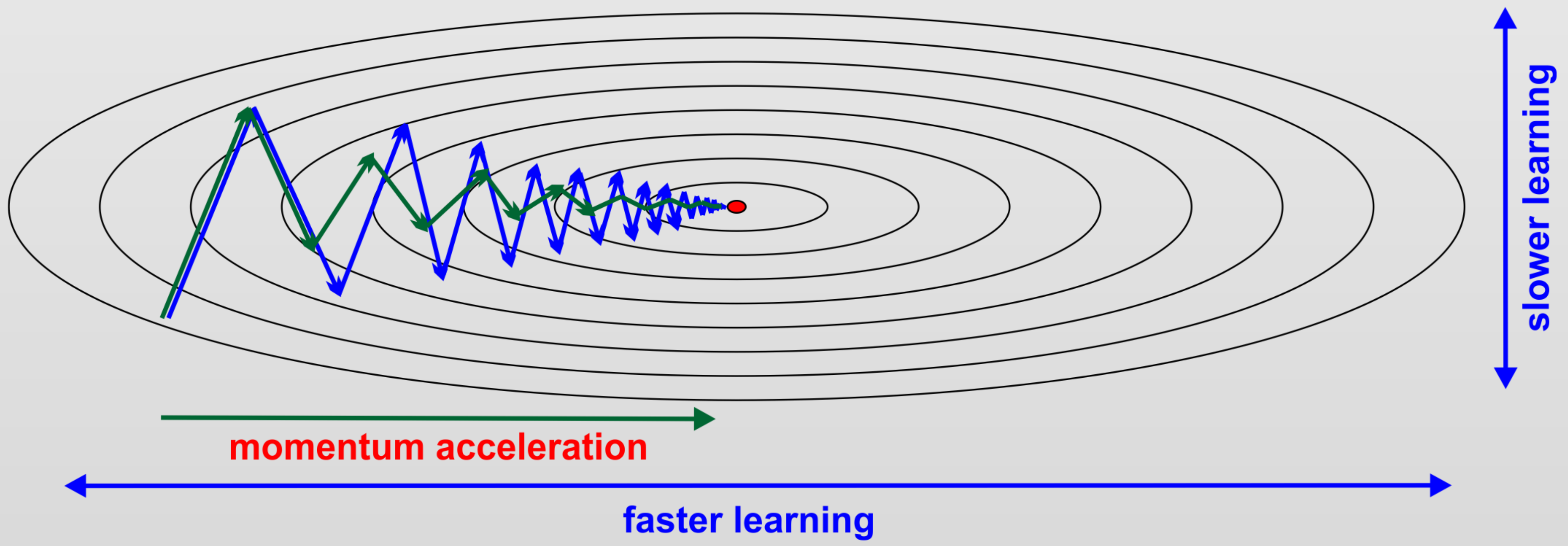
Gradient Descent with Momentum

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



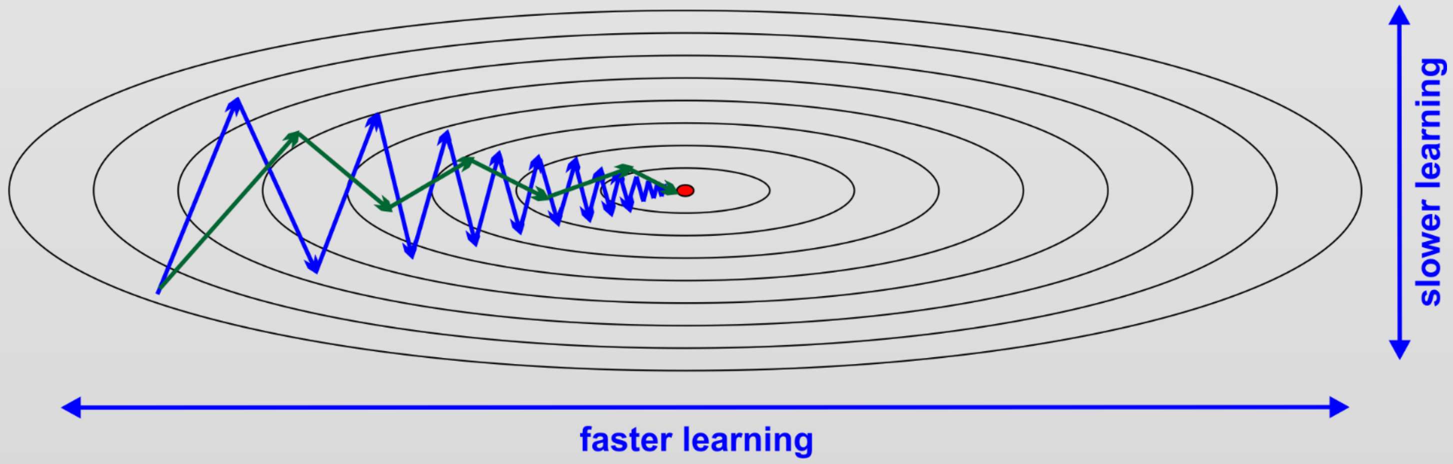
Root Mean Square Propagation

Root Mean Square Propagation (RMSprop):

- Computes exponentially weighted average of the squares of the derivatives
- $s_{dW} := \beta \cdot s_{dW} + (1-\beta) \cdot dW^2$ ***where dW^2 is element-wise***
- $s_{db} := \beta \cdot s_{db} + (1-\beta) \cdot db^2$ ***where db^2 is element-wise***
- Parameters are updated in the following way:
- $W := W - \alpha \cdot \frac{dW}{\sqrt{s_{dW}}}$ $b := b - \alpha \cdot \frac{db}{\sqrt{s_{db}}}$
- ***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:**

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

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

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

Learning Rate Decay

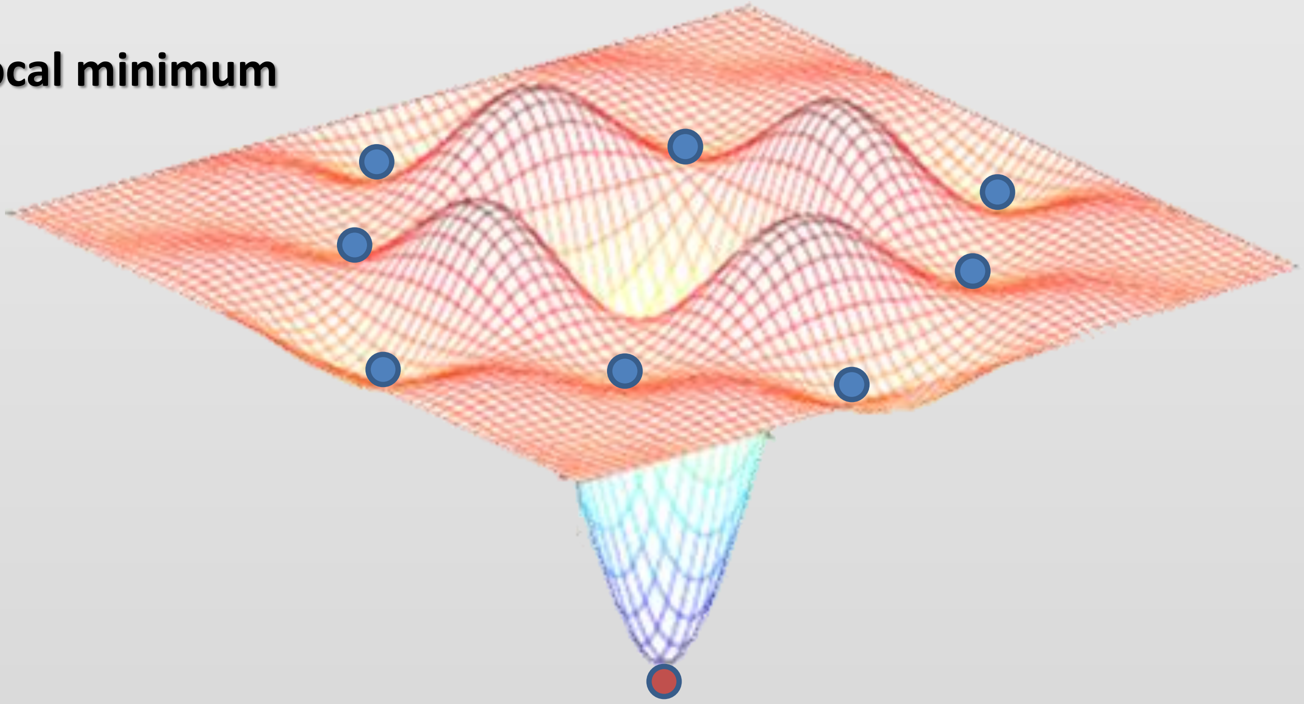
To avoid oscillation close to the minimum of the loss function, we should use non-constant learning rate but its decay, e.g.:

- **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:

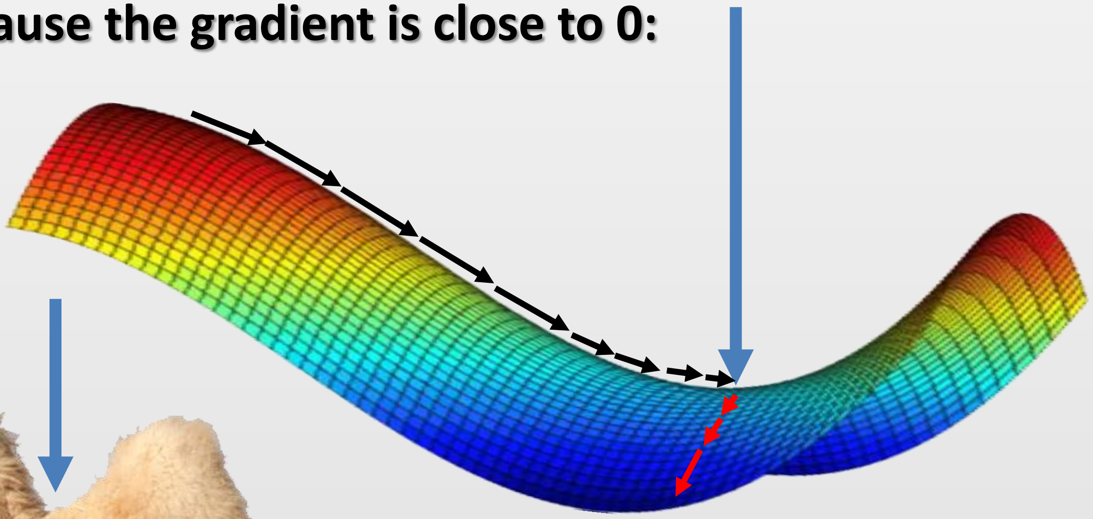
- **We must avoid stacking in local minima of the loss function.**
- **We can try to define such a loss function to have no local minima.**
- **We can try to escape from local minima using mini-batches, momentum, RMSprop, Adam optimizer.**
- **The gradient in any local minimum is always equal to 0!**



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.**

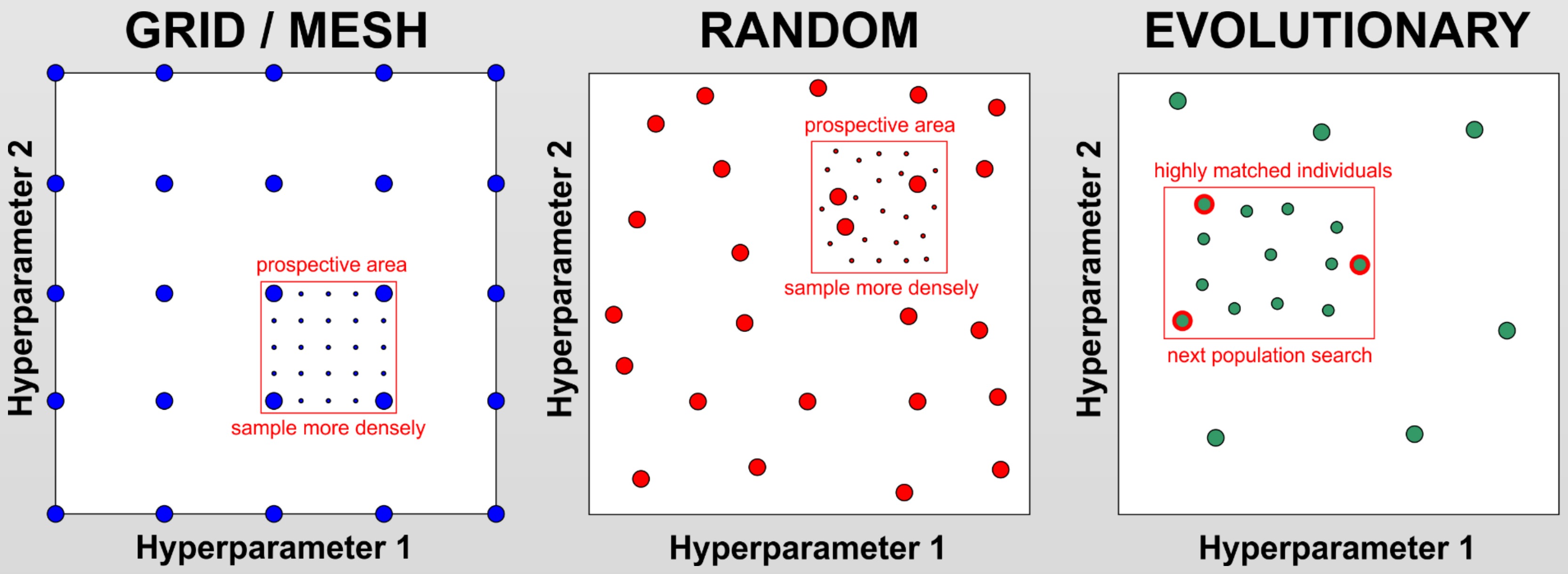


Optimization of Hyperparameters

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).**

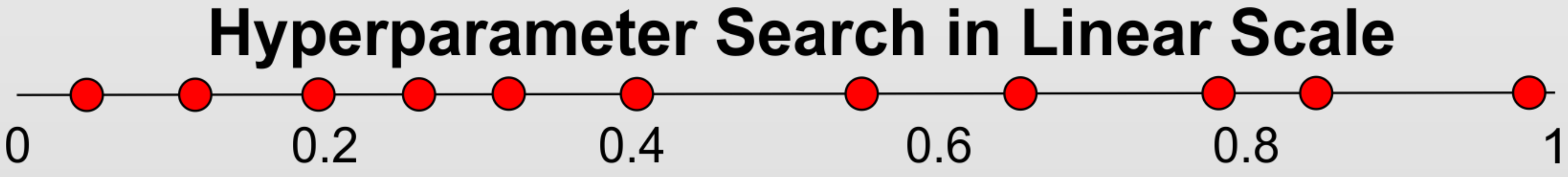


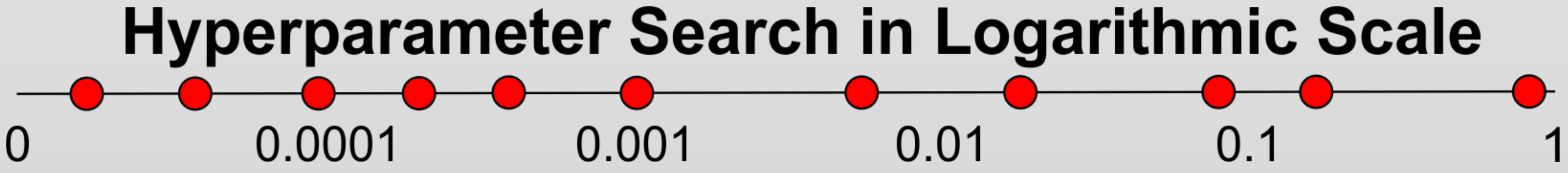
Scale of Hyperparameter Optimization

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.:

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

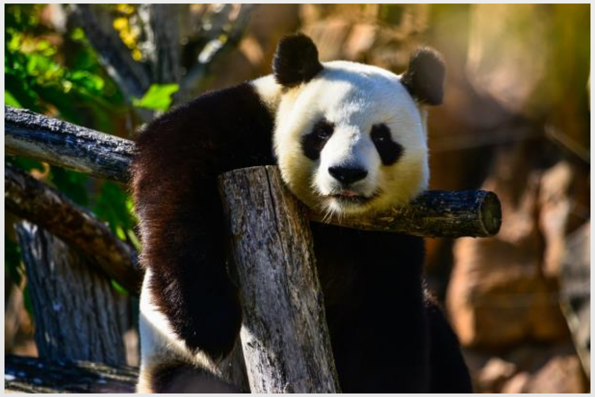




Approaches to Choose Hyperparameters

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.**

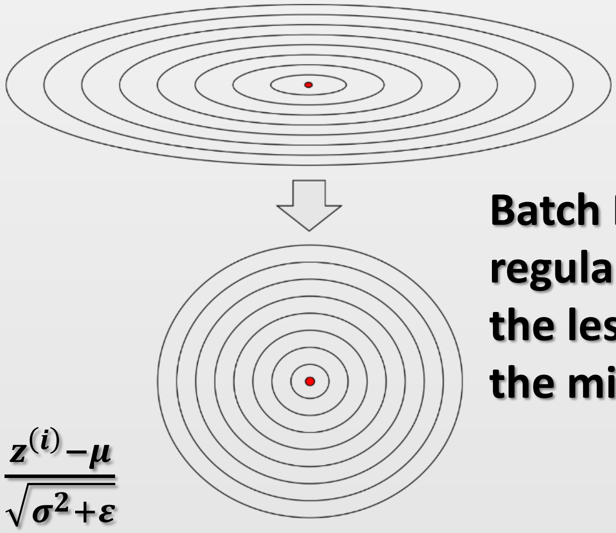


- **Many models train in parallel (Caviar strategy) – check many models using various combinations of the hyperparameters and choose the best one automatically. If you have enough computational resources, you can afford this model.**

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.

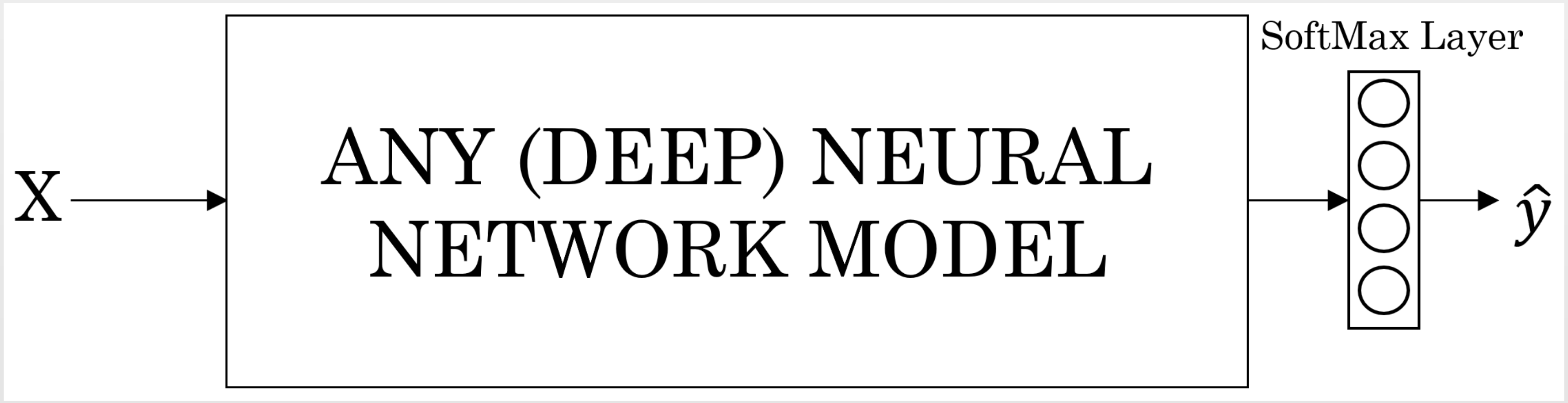
where β, γ are trainable parameters ($\beta^{[l]} := \beta^{[l]} - \alpha \cdot d\beta^{[l]}$, $\gamma^{[l]} := \gamma^{[l]} - \alpha \cdot d\gamma^{[l]}$) of the model, so we use gradients to update them in the same way as weights and biases.

- **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:**
- $x^{\{t\}} \rightarrow z^{[1]} \rightarrow \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \rightarrow z^{[2]} \rightarrow \tilde{z}^{[2]} \rightarrow a^{[2]} = g^{[2]}(\tilde{z}^{[2]}) \ldots$
- **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: $\hat{a}^{[L]} = g^{[L]}\left(z^{[L]}\right) = e^{z^{[L]}}$

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:

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

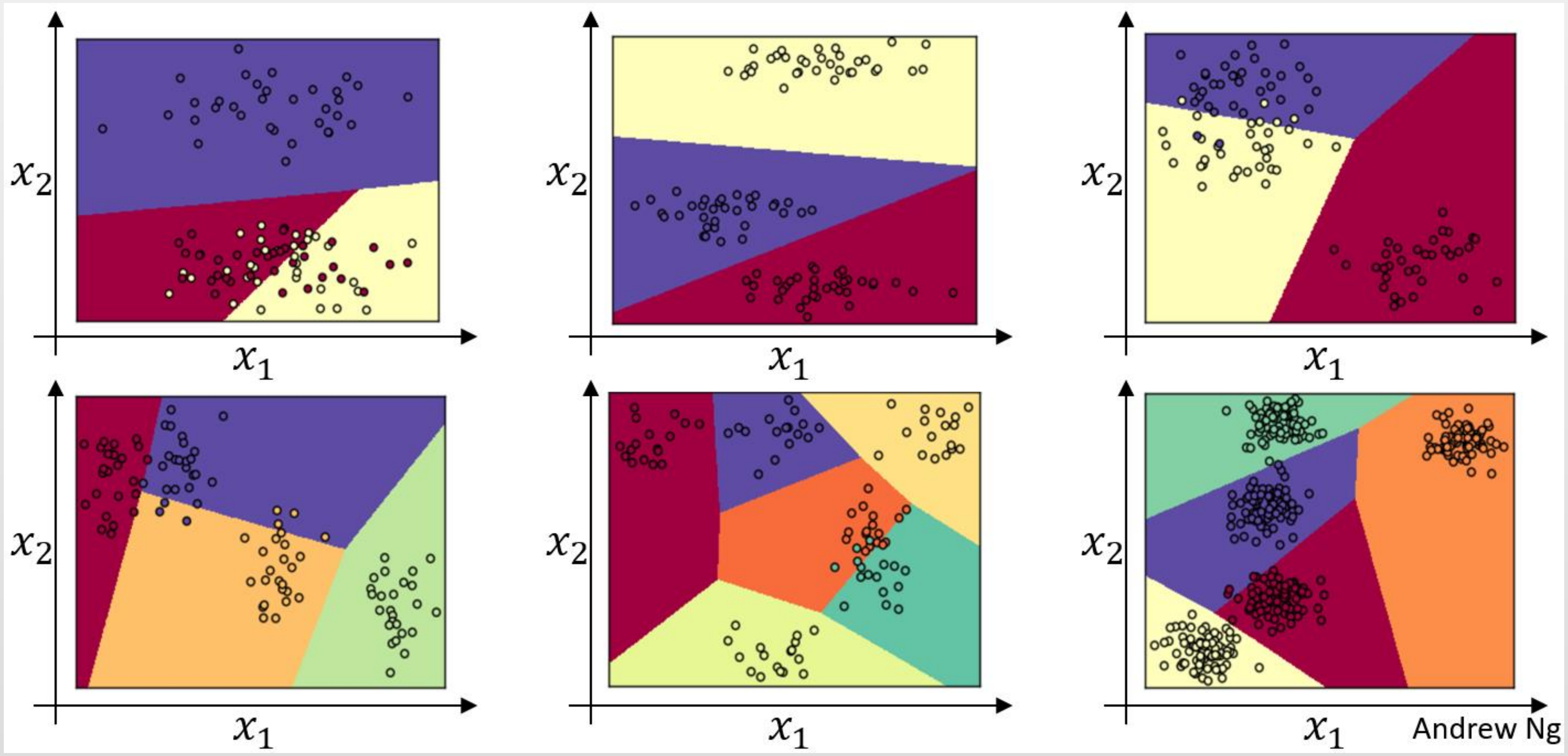
So the goal of the training is intuitively fulfilled.

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:

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

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:

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

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

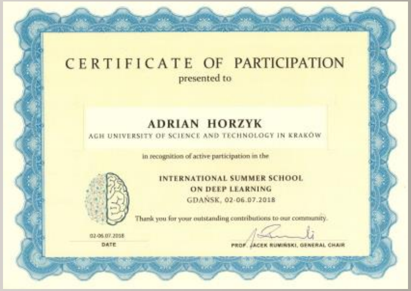
Bibliography and Literature

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



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