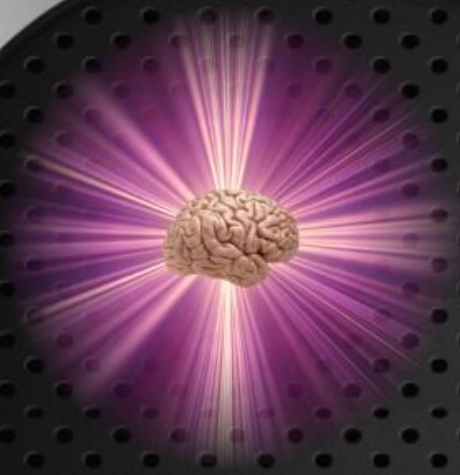
AGH University of Science and Technology
Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering
Department of Biocybernetics and Biomedical Engineering



Knowledge-based CI and ML in Biomedicine

Activation Functions, Optimizers and Transfer Learning

Adrian Horzyk
horzyk@agh.edu.pl

Google: Adrian Horzyk

Activation Functions of Neurons

Why to use different activation functions?

Which are the most efficient and how to use them?

Activation Functions of Neurons

We use different activation functions for neurons in different layers:

- **Sigmoid function is used in the output layer:**

  $$g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$$

- **Tangent hyperbolic function is used in hidden layers:**

  $$g(z) = tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

- **Rectified linear unit (ReLU) is used in hidden layers (FAST!):**

  $$g(z) = ReLu(z) = max(0, z)$$

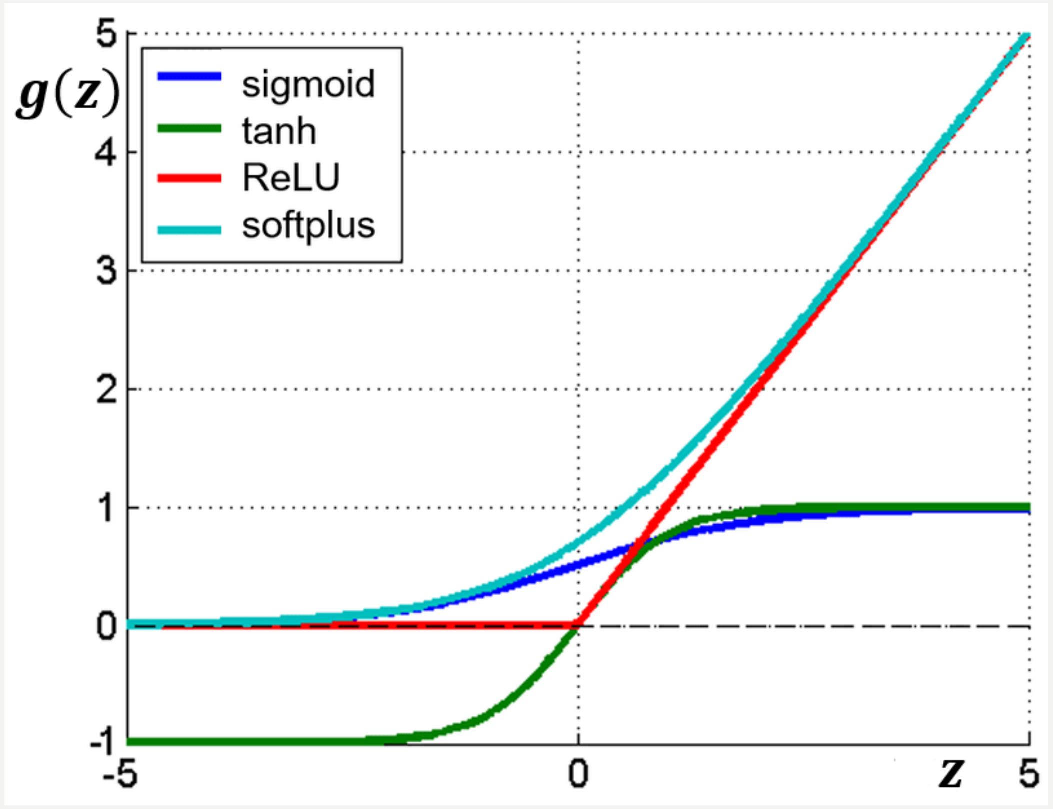
- **Smooth ReLu (SoftPlus) is used in hidden layers:**

  $$g(z) = SoftPlus(z) = log(1 + e^{z})$$

- **Leaky ReLu is used in hidden layers:**

  $$g(z) = LeakyReLu(z) = \begin{cases} z & if\ z > 0 \\ 0.01z & if\ z \le 0 \end{cases}$$

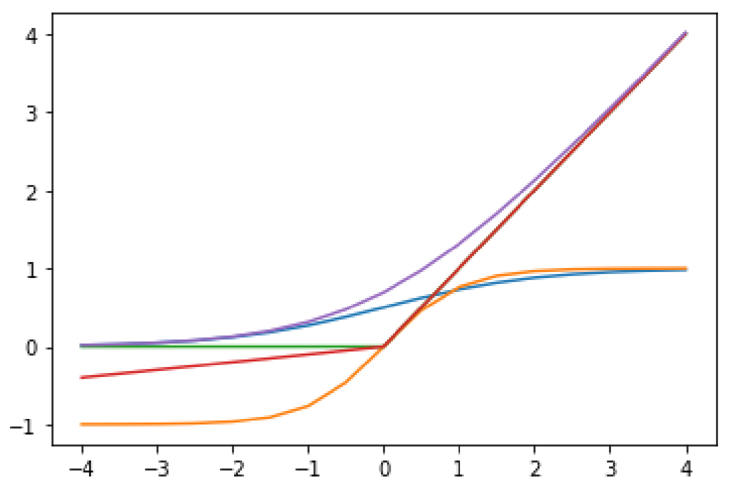
COMPARISON OF ACTIVATION FUNCTIONS



Activation Functions of Neurons

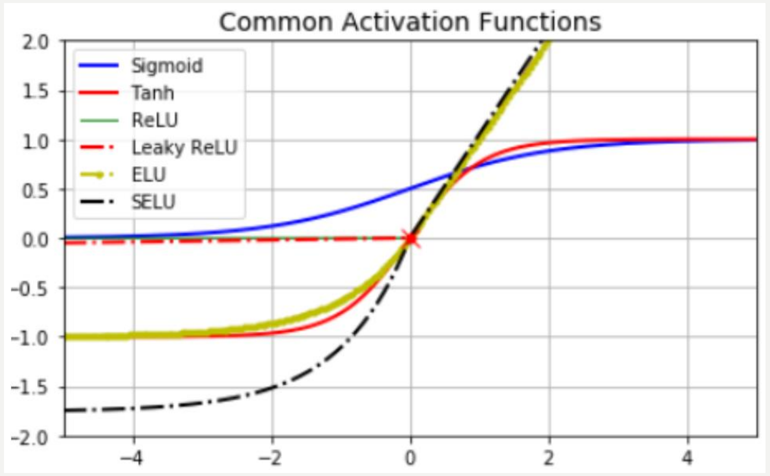
The most popular activation functions are defined as follows:

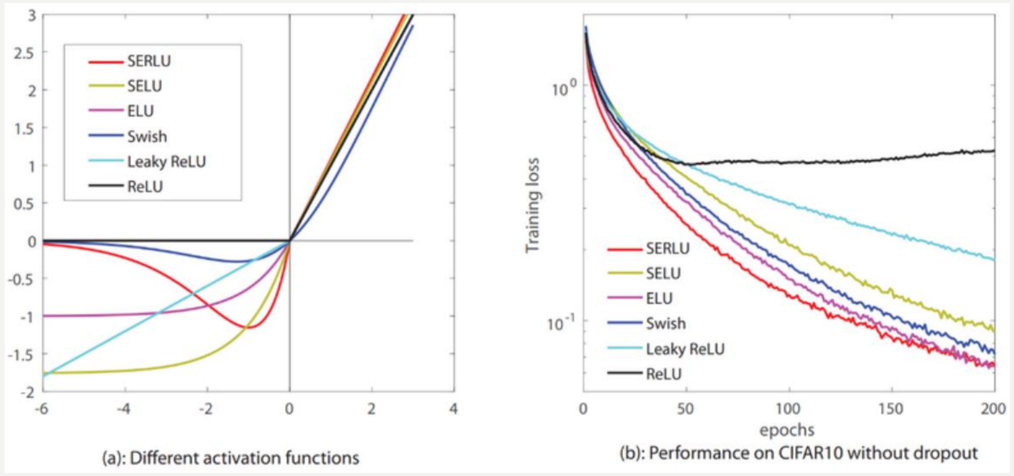
```python
import numpy as np

def sigmoid(x):
    s = 1 / (1 + np.exp(-x))   # use np.exp to implement sigmoid activation function that works on a vector or a matrix
    return s

def tanh(x):
    t = np.tanh(x)   # np.tanh to implement tanh activation function that works on a vector or a matrix
    return t

def relu(x):
    r = np.maximum(0, x)   # use np.maximum to implement relu activation function that works on a vector or a matrix
    return r

def leakyrelu(x, slope):
    l = np.maximum(x * slope, x)   # use np.maximum to implement leaky relu activation function that works on a vector or a m
    return l

def softplus(x):
    p = np.log(1 + np.exp(x))   # use np.log and np.exp to implement softplus activation function that works on a vector or a
    return p
```



Activation Functions of Neurons

Optimizing our model performance, we should try to use various activation functions because scientific papers and various experiments show that the change of activation functions substantially improve accuracy and decrease loss:





(a): Different activation functions
(b): Performance on CIFAR10 without dropout

You can try to improve your model trained on the CIFAR-10 dataset changing activation functions of neurons as presented in the above chart.

Two the most efficient activation functions for CIFAR-10 are:

- ELU (exponential linear unit): `model.add(Activation('elu'))`
- SERLU (scaled exponentially regularized linear unit).

SoftMax Activation Function

The SoftMax activation function (normalized exponential function) is used in the last layer of multinominal logistic regression or multi-class classification problems:

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 the 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, pointing to 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, $\sum_{j=1}^{n^{[l]}} a_j^{[L]} = 1$, in our case $0.042 + 0.842 + 0.002 + 0.114 = 1.0$

SoftMax Loss Function

When using SoftMax in the output layer, 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, and $y_c = 1$:

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

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

Then, the backpropagation step is started from:

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

SoftMax with Logistic Regression

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

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

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

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

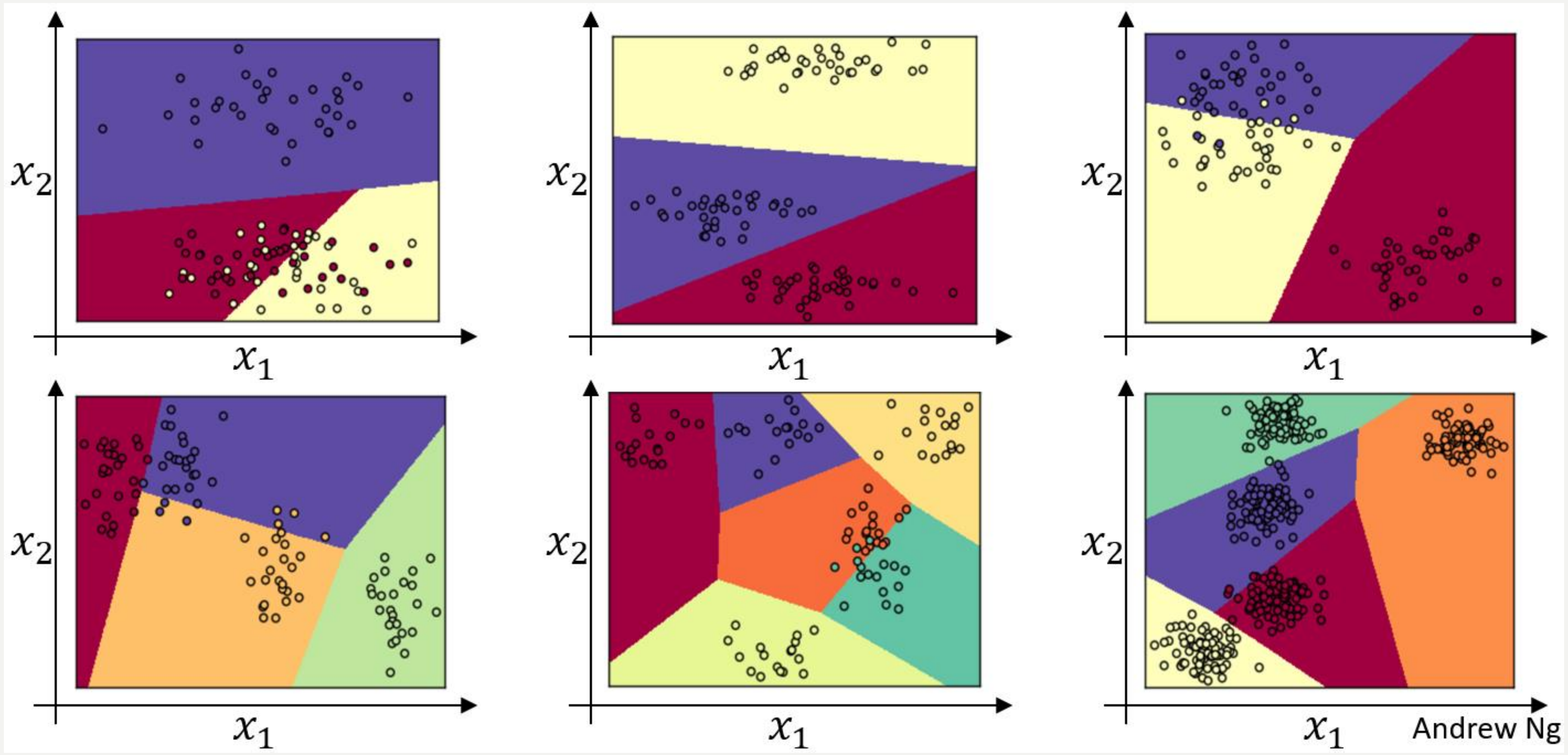
Consider another classification result that gives only one initial high estimation 0.88 for class 2 that is lower than 0.98. Which of these two classifications should we trust more (a) or (b) and why?

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

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

Possible results got by SoftMax

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

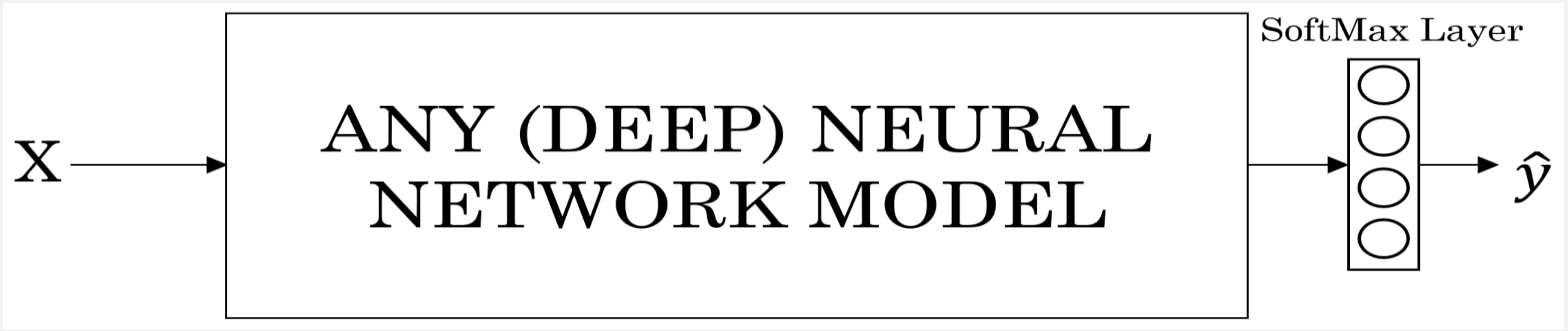


Can we trust such results or should we use deeper architecture to classify inputs with higher confidence?

SoftMax Regression

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

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



- Multi-class classification is when our dataset defines more than 2 classes, and the network answers should be not only 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. Hence, 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 exponentially 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.

Optimizers

How to control and optimize the training process?

Can we speed up the training process even more?

Exponentially Weighted Averages

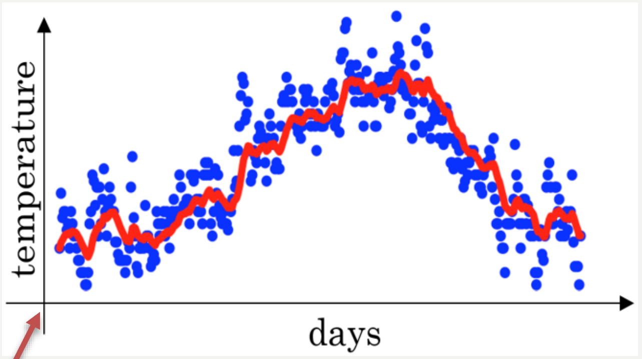
Exponentially Weighted (Moving) Averages is another much faster optimization algorithm than Gradient Descent:

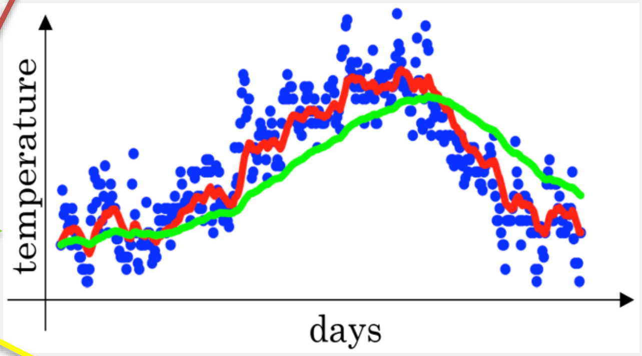
- **We compute weighted averages after the following formula:**
- $v_0 = 0$
- $v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot \theta_t$

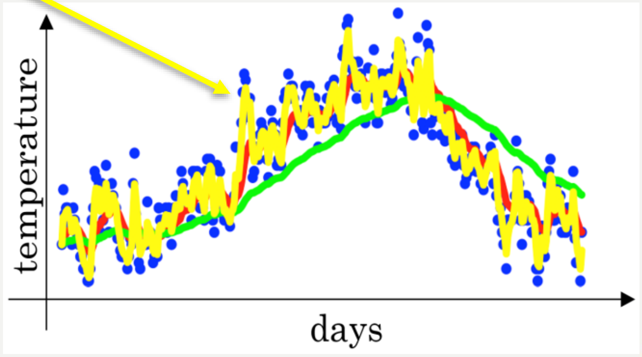
where β controls the number or previous steps that control the current value v_t :

- $\beta_{red} = 0.9$ (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.

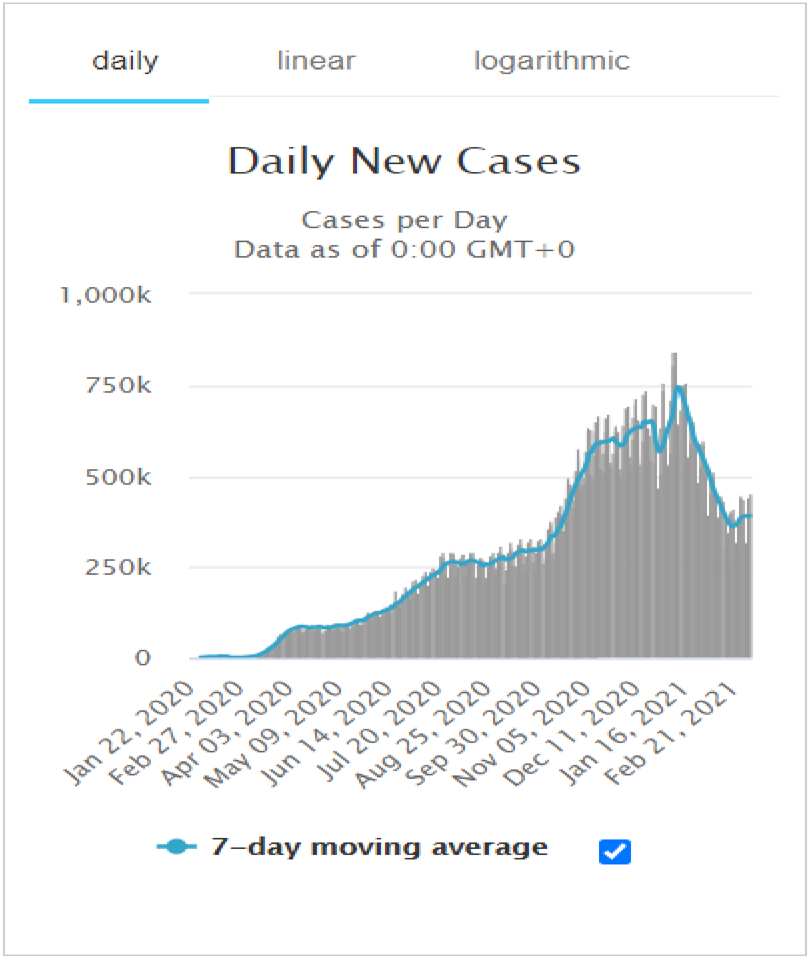


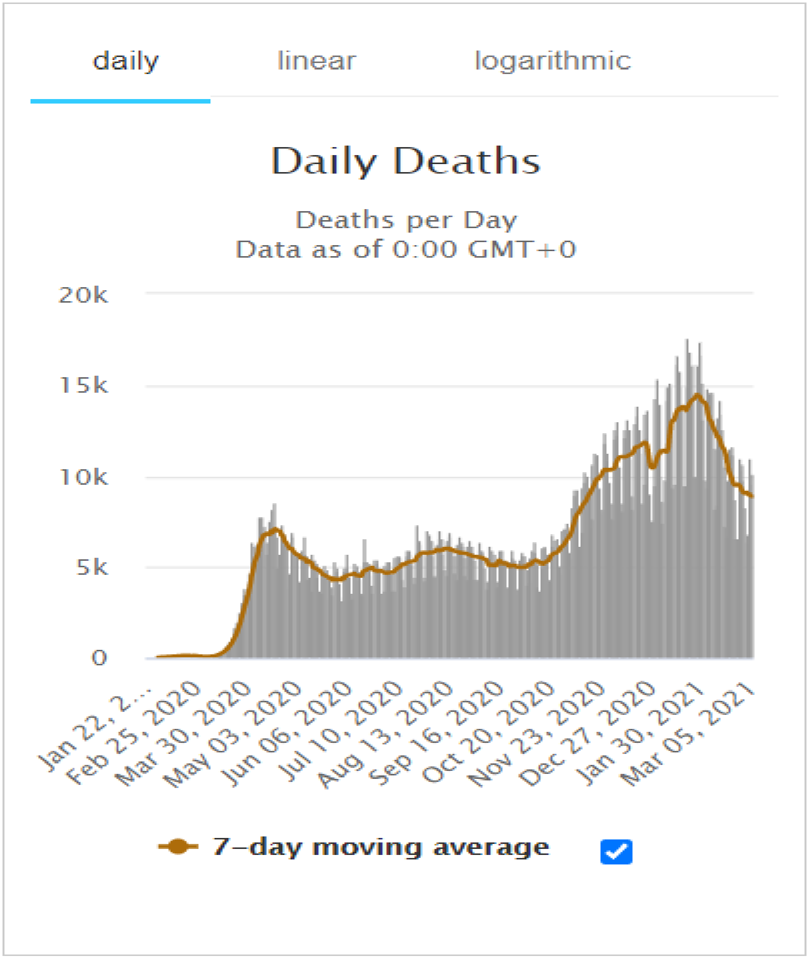


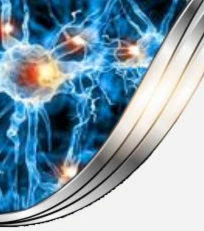


COVID Trend Smoothing

Exponentially Weighted Averages are used on the web page for calculating the 7-day moving averages of COVID daily new cases and daily deaths:







Exponentially Weighted Averages

Why we call this algorithm Exponentially Weighted Averages:

When we substitute and develop the formula:

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

we get the following:

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

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

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

Bias Correction for Exponentially Weighted Averages

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

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

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

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

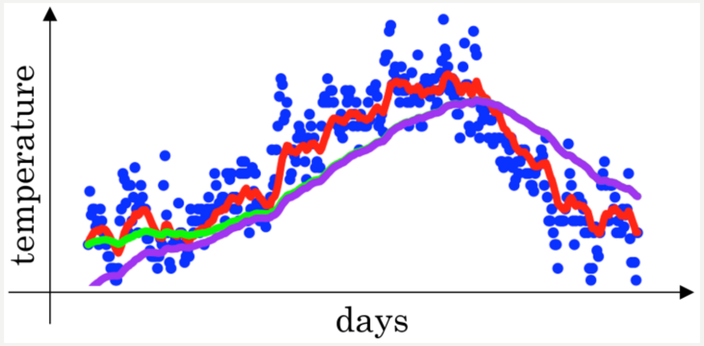
To avoid this, we use the correction factor (green curve) $1 - \beta^t$:

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

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

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

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.
- $W := W - \alpha \cdot v_{dw}$
- $b := b - \alpha \cdot v_{db}$
- $v_{dW} := \underbrace{\boxed{\beta}}_{\text{friction}} \cdot \underbrace{\boxed{v_{dW}}}_{\text{velocity}} + \underbrace{\boxed{(1-\beta) \cdot dW}}_{\text{acceleration}}$
- $v_{db} := \beta \cdot v_{db} + (1-\beta) \cdot db$
- The quotient $(1-\beta)$ is often omitted:
- $v_{dW} := \beta \cdot v_{dW} + dW$
- $v_{db} := \beta \cdot v_{db} + db$
- Hyperparameters: α, β, typical values of coefficients: $\alpha = 0.1, \beta = 0.9$
- Bias correction is rarely used with momentum, however might be used.
-

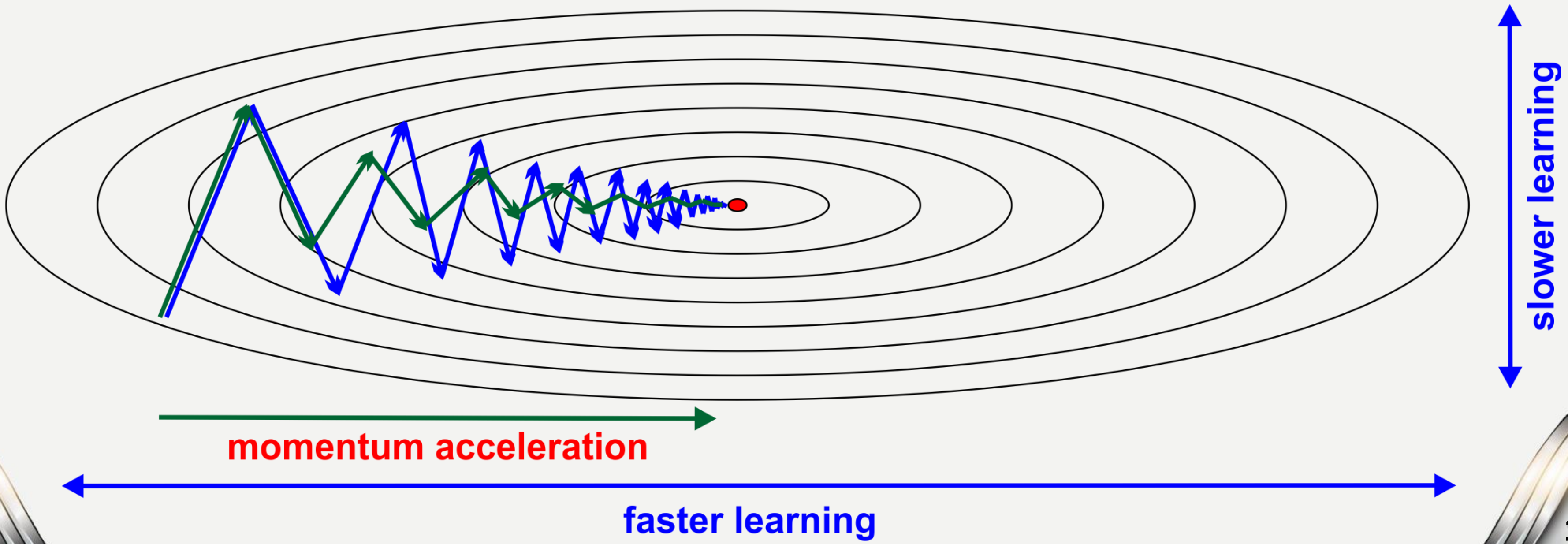
Gradient Descent with Momentum

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

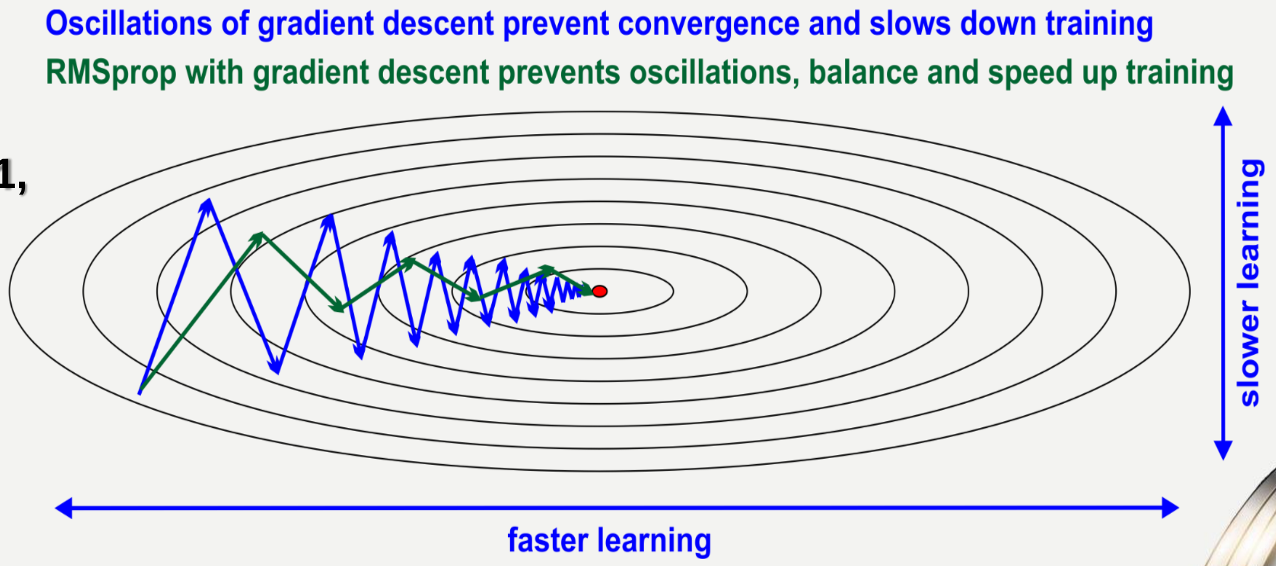
Momentum with gradient descent prevents oscillations and speed up training



Root Mean Square Propagation

Root Mean Square Propagation (RMSprop):

- Computes the exponentially weighted average of the squares of the derivatives
- $s_{dW} \coloneqq \beta \cdot s_{dW} + (1-\beta) \cdot dW^2$ *where dW^2 is element-wise*
- $s_{db} \coloneqq \beta \cdot s_{db} + (1-\beta) \cdot db^2$ *where db^2 is element-wise*
- Parameters are updated in the following way:
- $W \coloneqq W - \alpha \cdot \frac{dW}{\sqrt{s_{dW}}}$ $\quad$ $b \coloneqq b - \alpha \cdot \frac{db}{\sqrt{s_{db}}}$
- *Where $\sqrt{s_{dW}}$ and $\sqrt{s_{db}}$ balance the convergence process independently of how big or how small are dW, db, s_{dW}, and s_{db}.*
- optimizers.RMSprop(
    learning_rate=0.001,
    rho=0.9,
    momentum=0.0,
    epsilon=1e-07,
    centered=False,
    name="RMSprop",
    **kwargs)


Oscillations of gradient descent prevent convergence and slows down training
RMSprop with gradient descent prevents oscillations, balance and speed up training
slower learning
faster learning

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)

```
optimizers.Adam(
    lr=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name="Adam")
```

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

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

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

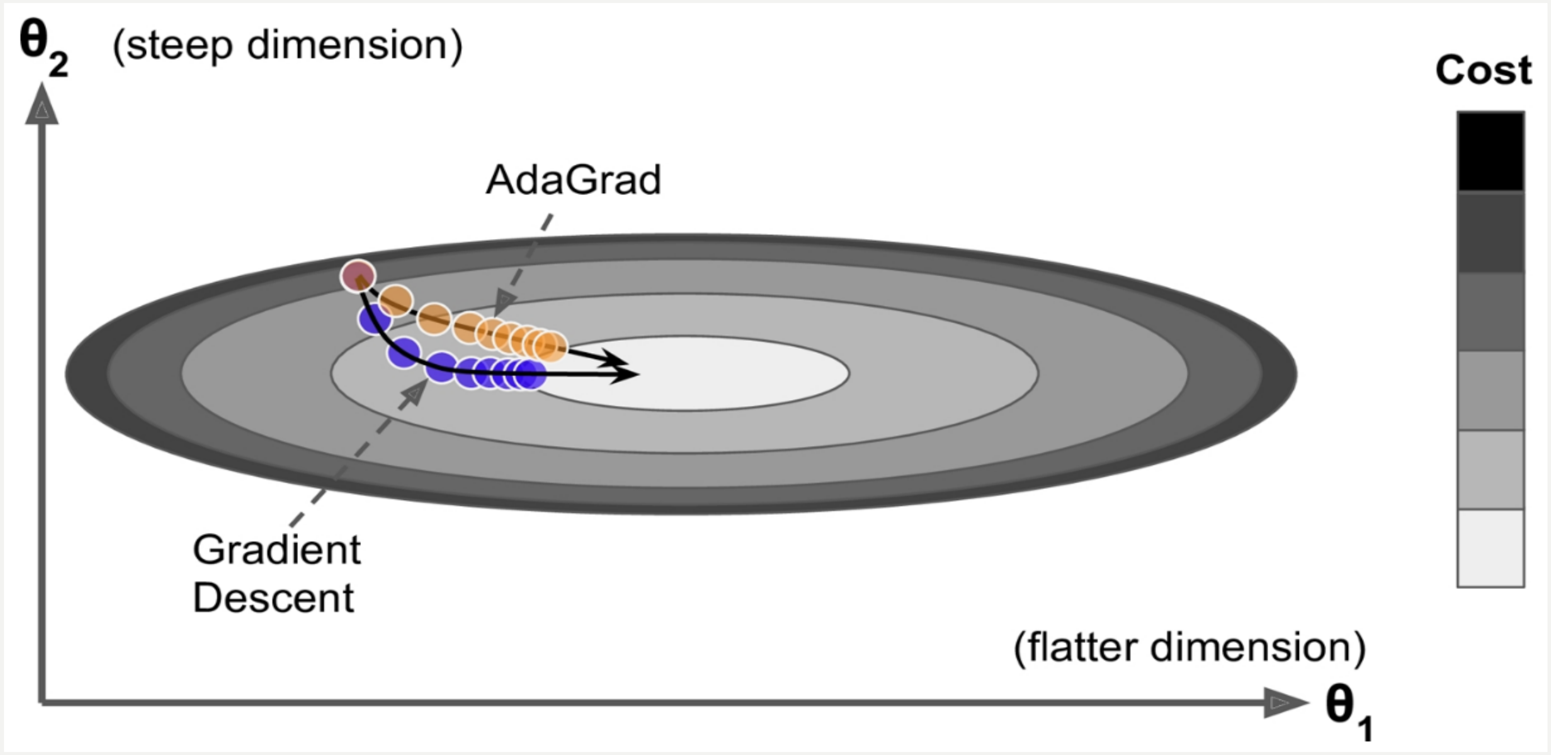
$$W := W - \alpha \cdot \frac{v_{dW}^{corr}}{\sqrt{s_{dW}^{corr}} + \varepsilon} \qquad b := b - \alpha \cdot \frac{v_{db}^{corr}}{\sqrt{s_{db}^{corr}} + \varepsilon}.$$

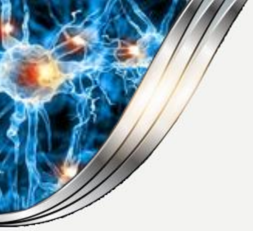
AdaGrad Optimization

Adaptive gradient descent (AdaGrad) decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes.

AdaGrad frequently performs well for simple quadratic problems, but it often stops too early when training neural networks.

```
optimizers.Adagrad(
    learning_rate=0.001,
    initial_accumulator_value=0.1,
    epsilon=1e-07,
    name="Adagrad",
    **kwargs)
```

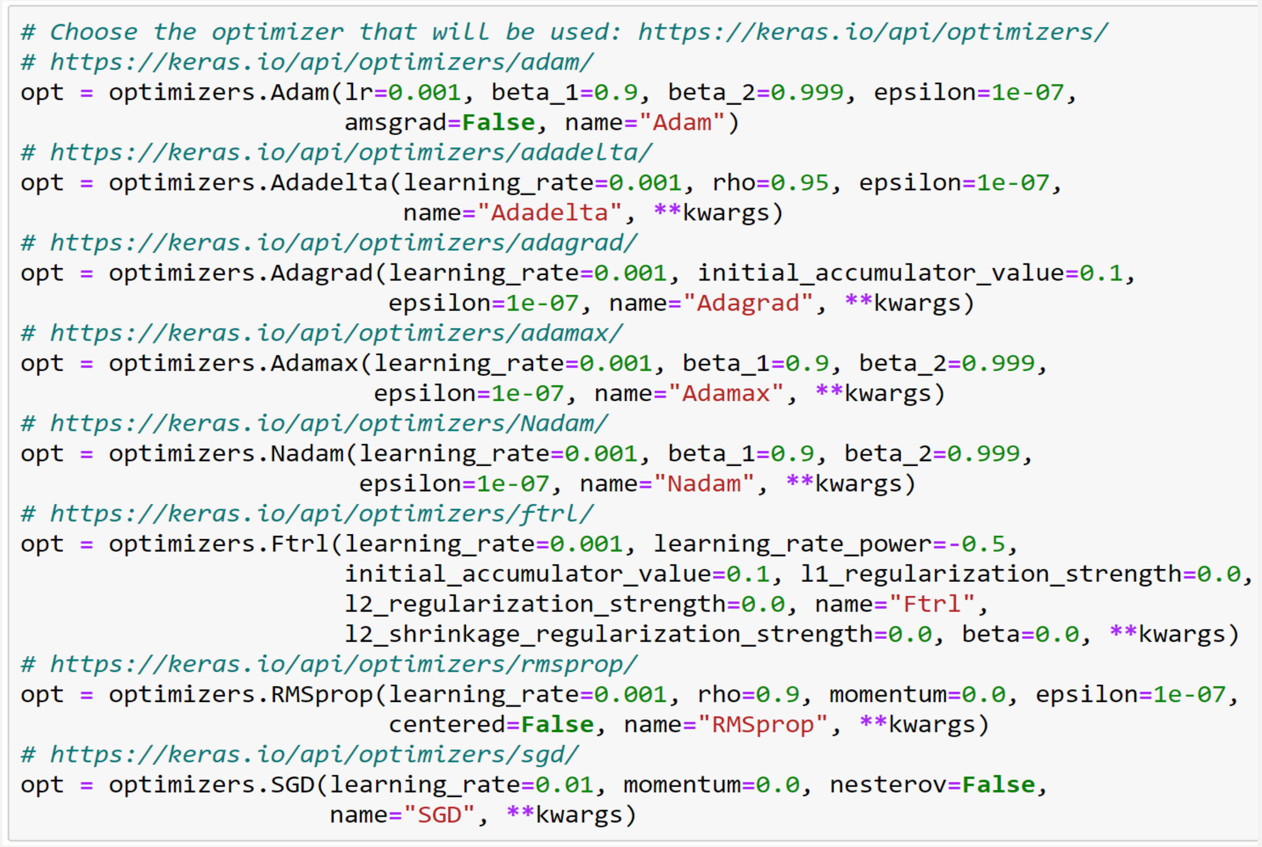




Optimizers

Today, we have many different optimizers, which we can use to fit our models. They can speed up the training process or allow for the easier escape of the local minima or saddle points:

- Nesterov accelerated gradient
- Momentum
- RMSprop
- Adam
- Adagrad
- Adadelta
- AdaMax
- Nadam
- AMSGrad



```
# Choose the optimizer that will be used: https://keras.io/api/optimizers/
# https://keras.io/api/optimizers/adam/
opt = optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07,
                      amsgrad=False, name="Adam")
# https://keras.io/api/optimizers/adadelta/
opt = optimizers.Adadelta(learning_rate=0.001, rho=0.95, epsilon=1e-07,
                          name="Adadelta", **kwargs)
# https://keras.io/api/optimizers/adagrad/
opt = optimizers.Adagrad(learning_rate=0.001, initial_accumulator_value=0.1,
                         epsilon=1e-07, name="Adagrad", **kwargs)
# https://keras.io/api/optimizers/adamax/
opt = optimizers.Adamax(learning_rate=0.001, beta_1=0.9, beta_2=0.999,
                        epsilon=1e-07, name="Adamax", **kwargs)
# https://keras.io/api/optimizers/Nadam/
opt = optimizers.Nadam(learning_rate=0.001, beta_1=0.9, beta_2=0.999,
                       epsilon=1e-07, name="Nadam", **kwargs)
# https://keras.io/api/optimizers/ftrl/
opt = optimizers.Ftrl(learning_rate=0.001, learning_rate_power=-0.5,
                      initial_accumulator_value=0.1, l1_regularization_strength=0.0,
                      l2_regularization_strength=0.0, name="Ftrl",
                      l2_shrinkage_regularization_strength=0.0, beta=0.0, **kwargs)
# https://keras.io/api/optimizers/rmsprop/
opt = optimizers.RMSprop(learning_rate=0.001, rho=0.9, momentum=0.0, epsilon=1e-07,
                         centered=False, name="RMSprop", **kwargs)
# https://keras.io/api/optimizers/sgd/
opt = optimizers.SGD(learning_rate=0.01, momentum=0.0, nesterov=False,
                     name="SGD", **kwargs)
```

Transfer Learning

How to use well-trained models to similar problems when having a small number of data?

Transfer Learning

Transfer learning is a common and highly effective approach to deep learning on small image datasets using a pre-trained network that is simply a saved network previously trained on a large dataset, typically on a large-scale image classification task.

If this large dataset is general enough (sufficiently covers input data space), then the spatial feature hierarchy learned by the pre-trained network can effectively act as a generic model of our visual world.

Hence, its features can prove useful for many different computer vision problems, even though these problems might involve completely different classes from those of the task for which the original network was trained.

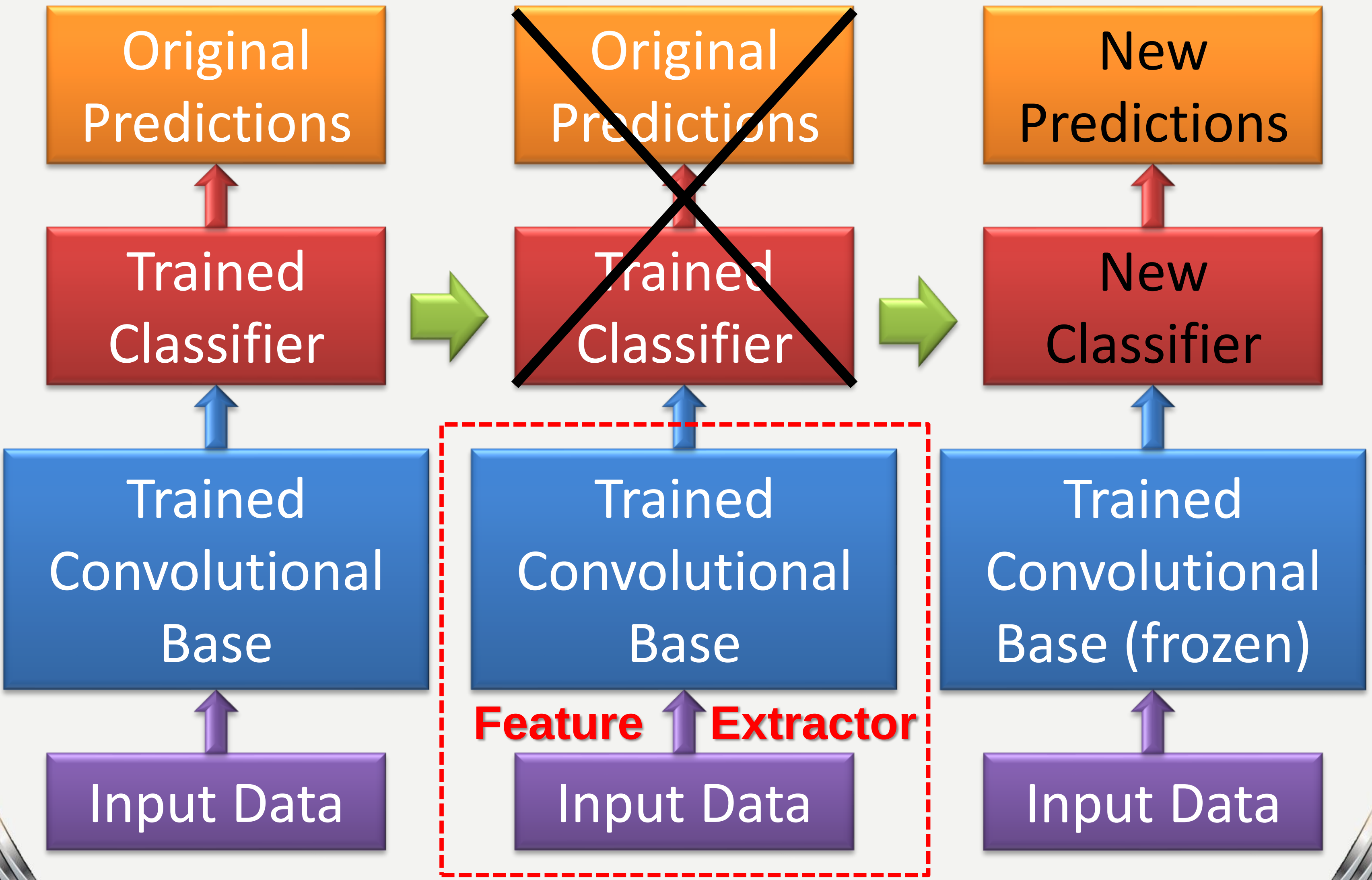
For instance, one might train a network on ImageNet and then re-purpose this trained network for identifying, e.g., furniture items or animals in images.

Such portability of learned features across different problems is a key advantage of deep learning compared to many older shallow learning approaches, and it makes deep learning very effective for small-data problems.

There are two ways to use a pre-trained network:
feature extraction and fine-tuning.

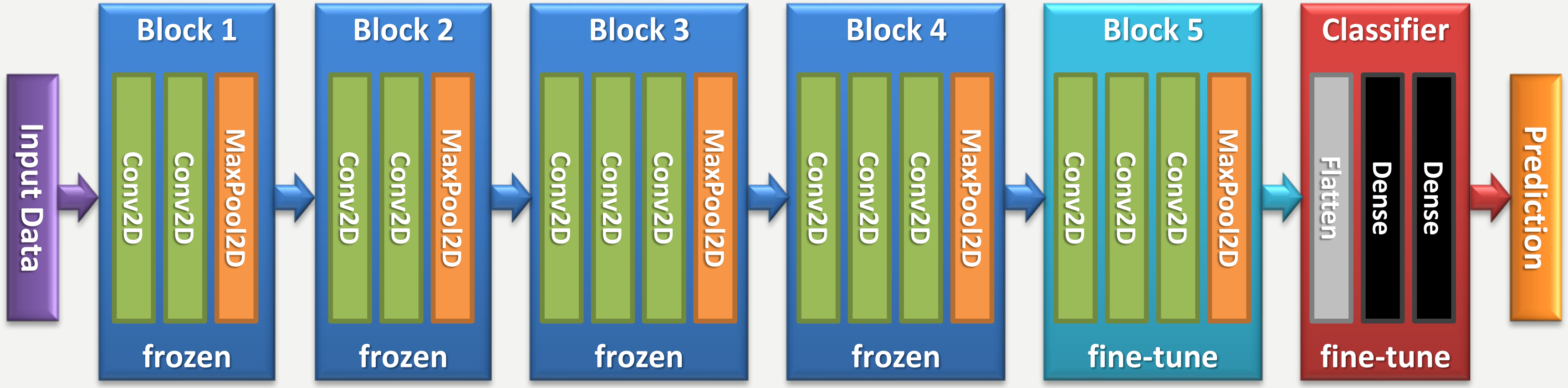
Transfer Convolutional Base

We usually use well-trained **convolutional bases** of the previously-trained networks, which **extract** various hierarchy **features** (`conv_base.trainable = False`) and adapt them to new (e.g. classification) tasks creating better models



Transfer Learning using VGG-16 with the frozen and fine-tuned layers

We can use selected number of the frozen convolutional layers (Blocks 1 – 4, disallowing changes of parameters) and fine-tune the unfrozen convolutional layers (Block 5) together with the Dense classifier layers in the training process:



This approach allows the classifier to preserve the most generic convolutional layers representing the most general lower-order features and adapt the more specific and higher-order convolutional features represented by the top filters which will better respond to a new classification task.

To fine-tune the higher-order filters, we first need to train the classifier, and when it is already quite well-trained, then unfreeze a few top convolutional layers (e.g. in Block 5) and next continue training with a small learning rate to limit the magnitude of the modifications that will be made in Block 5.

Transfer Learning

When working with transferred models, we have to:

1. **Upload the transferred model or import it from the library:**

```
from keras.applications import VGG16

conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(150, 150, 3))
```

2. **We have to freeze the convolutional base to prevent its changes before we start training the randomly initialized classifier:**

```
conv_base.trainable = False
```

3. **Next, we add the frozen convolutional base to the model:**

```
model3 = models.Sequential()
model3.add(conv_base)
model3.add(layers.Flatten())
model3.add(layers.Dense(256, activation='relu'))
model3.add(layers.Dense(1, activation='sigmoid'))
```

Fine-tuning Steps

Steps for fine-tuning a network are as follow:

1. Add your custom network (e.g., a dense classifier) on top of an already trained network base (conv_base).
2. Freeze the base network. `conv_base.trainable = False`
3. Train the part (e.g. classifier) you added on top.
4. Unfreeze last layers (a block of layers) in the network base.
5. Jointly train both these unfrozen layers and the part you added.
6. While the results are not yet satisfactory and reprezented features in the last layers do not suite to our problem:
    1. Unfreeze a few additional last layers (the next block) in the network base.
    2. Jointly train both these unfrozen layers and the part you added.

Freezing and Unfreezing Layers

When we want to fine-tune the network, we must unfreeze some convolutional layers. However, we should consider that:

- Earlier layers in the convolutional base encode more generic, reusable features, while layers higher up encode more specialized features.
  It is more useful to fine-tune the more specialized features, as these are the ones that need to be repurposed on our new problem.
  There would be fast-decreasing returns in fine-tuning lower layers.
- The more parameters we are training, the more we are at risk of overfitting.
  In our case, the convolutional base has usually over millions of parameters, so it would be risky to attempt to train it on a small dataset.

```
conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

Guideles, Tips and Tricks for Improving Performance

What to do when the standard methods of struggling with model performance have failed?

Performance and Final Tips & Tricks

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 (in a different way).**
- **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.**
- **Use regularization, dropout, optimization methods (e.g. Adam optimizer).**
- **Train a chosen network longer with different or changing learning rates.**
- **Compare results achieved for various architectures and the other hyperparameters.**

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!

How to implement Dropout?

The original paper on Dropout provides experimental results which provide a number of useful heuristics to consider when using dropout in practice:

- **Use a small dropout value of 0.2 – 0.5 of neurons with 0.2 providing a good starting point, however, for small networks or small layers 0.1 might be the right value.**
  Too low probability has minimal effect on the model,
  and when it is too high, it results in underfitting by the network.
- **Use a larger network to give the model more of an opportunity to learn independent representations.**
- **Use dropout on incoming (visible) as well as hidden units (layers).**
  Application of dropout at each layer of the network has shown good results.
- **Use a large learning rate with decay and a large momentum, increasing your learning rate by a factor of 10 to 100 and use a high momentum value of 0.9 or 0.99.**
- **Constrain the size of network weights. A large learning rate can result in very large network weights. Imposing a constraint on the size of network weights such as max-norm regularization with a size of 4 or 5 has been shown to improve results.**

Problems with many Outliers

- Outliers are data that do not match the data model represented by other data.
- Outliers often fall out outside the range of variation of other data for one or more attributes.
- Outliers are responsible for the rise of bias and variance, especially when there are many of them! However, normally, models deal with a small number of outliers!
- Sometimes outliers are unusual combinations of common data, which are within the limits of the variation of individual attributes, but this variation is so strange that it is not compatible with the other combinations, e.g., for classification problems.
- Outliers may arise as a result of errors, anomalies (e.g., in measurement), or specific (sometimes interesting) phenomena.
- There is no strict mathematical definition of outliers, as they usually depend on the nature of the data and the subjective assessment.
- Outliers are usually removed or replaced with zero, average, median, ...
- The median is quite robust to data outliers, but the average is not.
- It uses a Vinsor's average in which selected extreme observations are replaced by the minimum and maximum values from the remaining data, respectively.

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? If yes, consider using strengthening factor.
- 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 numbers of examples which represent them.
- You can try to strengthen the training process for the incorrectly classified examples, e.g. use different strengthening factors for the training 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%	

- Use different knobs to fit a training, validation (dev) and testing sets and to real data final check.

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.**

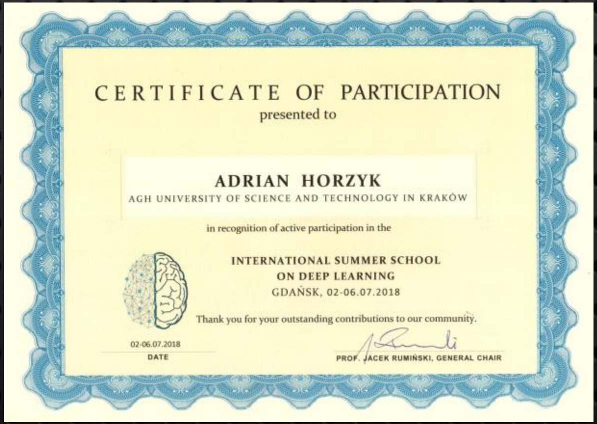
BIBLIOGRAPY

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/ahdydkbcidmb.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.

AGH

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

[illegible]
PROF. JACEK RUMIŃSKI, GENERAL CHAIR