Kaunas University of Technology

Faculty of Mechanical Engineering and Design

Deep learning model for image classification and corrosion detection with Tensorflow API

Report

Lukas Lenktaitis

Project author

Prof. dr. Jolanta Baskutienė

Supervisor

Summary

Traditional manual methods of corrosion and rust detection in ships' hulls, metal construction buildings, and bridges are not only time-consuming and costly but also prone to inaccuracies. Manual inspection heavily relies on the skill and experience of inspectors, and even minor mistakes during the process can have significant consequences. While manual inspection can be partially replaced by drones or robot divers capturing images from various angles, fully analyzing the condition and identifying damaged surfaces can still require extensive time and resources from engineering teams. Moreover, overlooked surfaces can lead to critical problems over time, necessitating regular inspection intervals.

To address these challenges, the aim is to develop a Convolutional Neural Network (CNN) model using TensorFlow and Keras for corrosion detection in images. Objectives include explaining the operations and calculations in deep learning neural networks required for image recognition, analyzing traditional visual inspection methods in the industry, and comparing them with the advantages and disadvantages of the CNN model.

Visual inspection remains essential for identifying and evaluating surface flaws such as corrosion, contamination, irregular surface finish, and joint discontinuities. It is particularly effective in detecting critical surface cracks associated with structural failure mechanisms. Visual inspection methods utilize various equipment, from naked eye observation to interference microscopes, depending on the product and surface flaw type. By implementing CNN models for corrosion detection, the efficiency and accuracy of inspection processes can be significantly enhanced, ultimately improving maintenance practices in various industries.

Introduction

Corrosion and rust detection in hull of the ships, metal construction buildings or bridges manually without any computer vision or inspection are time consuming process and extremely expensive, nevertheless, most of the time it will not be accurate comparing with what deep learning models enable to inspect. Manual inspection's result depends on skill and experience of inspectors, and any even small mistake made during the inspection process can cause terrific consequences in the future and that is critically important. The manual inspection can be replaced partly by drones or robot divers taking pictures of different angles, and after composing all pictures, inspectors are able to go through all collected pictures and determine where surface is damaged by rust and where repairing is necessary. Fully analyze condition and determine damaged surfaces can take hundreds of hours by engineering teams, anyway, after some time not spotted surfaces will cause critical problems, that is why most inspection must be implemented over and over again by some period of time.

Aim: Develop a Convolutional Neural Network (CNN) model using TensorFlow and Keras for corrosion detection in images.

Objectives – explain operations and calculations in deep learning neural networks necessary for image recognition, analyze visual inspection methods in industry and compare it with created model by approving advantages or disadvantages of CNN model.

Requirements and Technical Specifications: Create a CNN model that can accurately detect corrosion in images.

Visual inspection is essential for identifying and evaluating surface flaws such as corrosion, contamination, irregular surface finish, and joint discontinuities (e.g., welds, seals, solder connections). It is especially effective in detecting critical surface cracks associated with structural failure mechanisms. Even when other inspection techniques are used, visual inspection often complements them. For example, during eddy current examination of process tubing, visual inspection verifies surface disturbances. Acid etching (macroetching) can also reveal structures invisible to the naked eye, as shown in Fig. 1. Different techniques are employed depending on the product and surface flaw type. Visual inspection methods utilize various equipment, from naked eye observation to interference microscopes for measuring scratch depth on polished surfaces.

1. Artificial Intelligence Implementation and Concept of Neural Networks

Nowadays artificial intelligence has developed to the level where human behavior can be easily manipulated and be repeated, it can deal with most difficult task where human has struggle to complete it or even is not able to understand it. Ability to cope with enormously complex difficulties lead to rapid growth in many industry fields, thus technology do not require additional work force, can be implemented without any big amount of an investment and can be customize or adjust in many ways. Artificial intelligence let to expand humanity to exceptional possibilities such as autonomous self-driving cars, natural language processing (NPL), image - language translations, demographic predictions and etc. In manufacturing processes AI is helping with visual recognition, defects detection, sound analyzing and in other fields. The concept of Artificial Intelligence is defined by ability which enables computers to mimic human behavior, however, AI cannot perform without its main core - machine learning which gives ability to learn without being programmed to do exact task and deep learning which make the computation of multi-layer neural networks feasible. To understand functionality and features of whole computational process of artificial intelligence it is necessary to analyze deep learning and its neural networking.

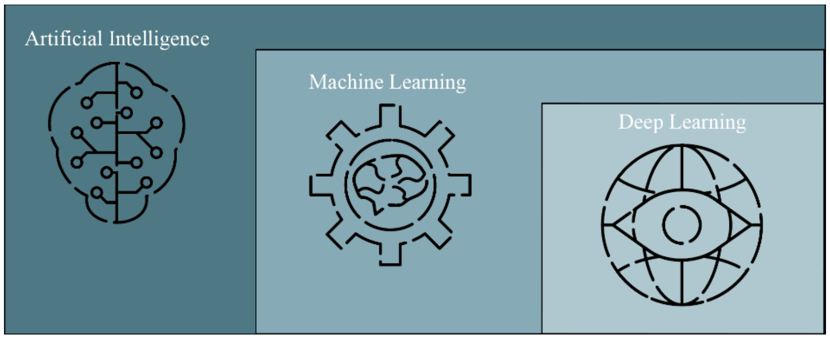


Figure 1. Hierarchy of artificial intelligence, machine learning and deep learning.

https://blogs.oracle.com/bigdata/difference-ai-machine-learning-deep-learning

1.1. Development period of Neural Networks

Neural Networks (NN) are computational models to manipulate the human nervous system functionality [3]. First model concept was announced and suggested in 1943 by W. S. McCulloch and W. H. Pitts.

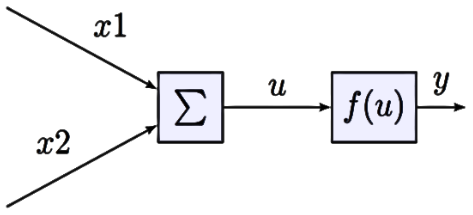


Figure 2. The neural network model of McCulloch and Pitts. []

The model was described as "nets with alterable synapses" [4], where it is created by two different binary states in which the neuron has excitation synapses and can be activated by it, otherwise, using inhibitory synapses, can be deactivated. If the summation of the states x_1 and x_2, u, is more than the threshold (θ) then the output will be one, otherwise the output will be zero [equation 1]

$$f(u) = \begin{cases} 0 : if \;\; u \geq \theta \\ 1 : if \;\; u < \theta \end{cases}$$

Equation 1. Activation and deactivation function of the neuron. []

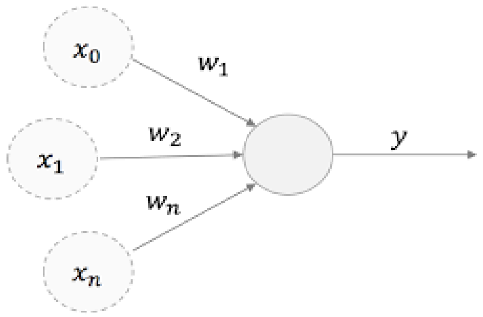


Figure 3. Hebbian learning model, where many inputs come out with a single output in one neuron. [https://www.bonaccorso.eu/2017/08/21/ml-algorithms-addendum-hebbian-learning/]

In 1949 Canadian psychologist Donald Hebb described the concept of Hebbian learning of NN model in the book "The Organization of Behavior".

Hebbian learning can be defined as the set of rules, according to which the weight associated with a synapse increases proportionally to the values of the pre-synaptic and postsynaptic stimuli at a given instant of time (fig. 3), by considering a linear neuron, therefore the output y is a linear combination of its input values x [equation 2].

$$y = \sum_{i=1}^{n} w_i x_i$$

Equation 2. Linear equation of Hebbian learning.

1.2. Frank Rosenblatt's Perceptron

In 1958 the perceptron was developed by the American psychologist Frank Rosenblatt. The concept of perceptron came from brain's neuron as the systems algorithm which classifies placed input and differs it into two different categories. The main reason was to implement an algorithm which is able to understand and learn from the values of given weights ***w***, where those weights are multiplied with applied input characteristics in order to come up with decision that is suitable or inappropriate, for making pattern classification useful by determining that applied data belongs to set of the class or not. Frank Rosenblatt's perceptron functionality depends on the unit step function (Heaviside step function) where output is labeled as the positive and negative class in binary classification where 1 and -1 is generated respectively. Activation function **g(z)** must be define that takes a linear combination (where classes can be separate by one line) of the input values **x** and weights **w** as input (**$z = w_1x_1+\dots+w_mx_m$**), and if **g(z)** is greater than a defined threshold **θ** we predict 1 and -1.

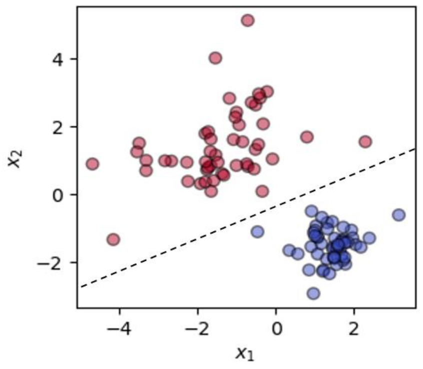


Figure 4. Example of a linear decision boundary for binary classification []

$$g(\boldsymbol{z}) = \begin{cases} 1\ if\ z \geq \theta \\ -1\ if\ otherwise \end{cases}$$

Equation 3 []

$$\boldsymbol{z} = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^{n} x_jw_j = w^T x$$

Equation 4 []

Weight **w** is the attribute of the vector, x is a quantity of numerical value of the m-dimensional sample from the training dataset: [fig. 6]

$$w = \begin{bmatrix} w_1 \\ \dots \\ w_m \end{bmatrix} x = \begin{bmatrix} x_1 \\ \dots \\ x_m \end{bmatrix}$$

Equation 5

The implementation process of the notation is to define θ point which is moved to the side as shown in the graph [fig. 8] and define **$w_0 = -\theta$ and $x_0 = 1$.**

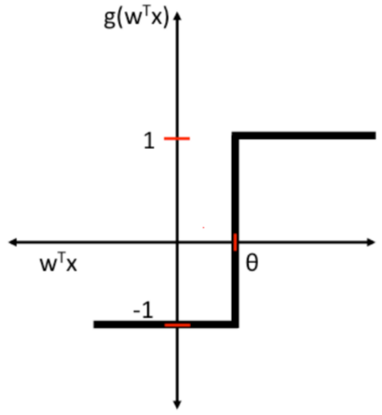


Figure 5

In order to come up with an output of the perceptron it must receive multiple signals which are calculated in certain threshold where the signal will not summon any reaction and remain still or it will be assumed as impactful product of numerical data. That was the idea of the perceptron algorithm [fig. 8] where computation of the weights as signals cause ability to learn by drawing linear decision boundary that allows to discriminate between the two linearly separable classes +1 and -1.

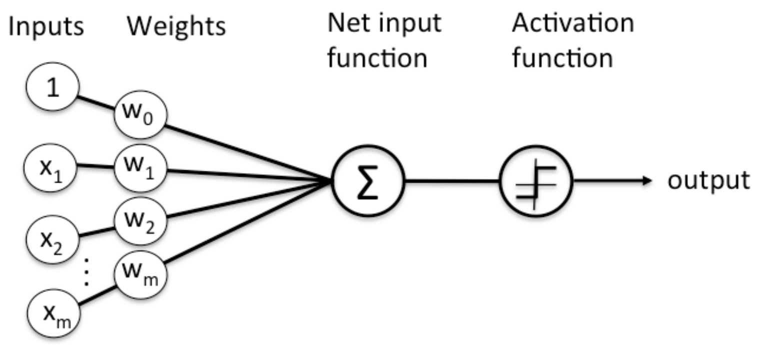


Figure 6. Rosenblatt's perceptron

Rosenblatt's perceptron rule can be defined and implemented with few following steps:

Initialize the weights to 0 or small random numbers.

1. For each training sample $\mathbf{x}^{(i)}$:
    1. Calculate the output value.
    2. Update the weights. []

The value of the output is predicted by class label in the unit step function (defined output as shown in the equation) and in order to define weight's update, equation is expressed:

$$w_j = w_j + \Delta w_j$$

Equation 6

The process of weights updating every time when increment is changed can be defined by the learning rule:

$$\Delta w_j = \eta(target^{(i)} - output^{(i)})x_j^{(i)}$$

Equation 7

Where symbol eta (η) is the learning rate and the value of it cannot exceed 1 or be below 0 and known as constant between those two numbers, (a constant between 0.0 and 1.0), "target" (Y) is the true class label, the "output" (Ŷ) is the predicted class label []. All weights of the vectors are simultaneously updated and then further operation is proceeded. For two dimensional perceptron update is written as shown in equation 8; 9; 10 respectively:

$$\Delta w_0 = \eta(\mathrm{Y}^{(i)} - \hat{\mathrm{Y}}^{(i)})$$

Equation 8

$$\Delta w_1 = \eta\left(\mathrm{Y}^{(i)} - \hat{\mathrm{Y}}^{(i)}\right)X_1^{(i)}$$

Equation 9

$$\Delta w_2 = \eta(\mathrm{Y}^{(i)} - \hat{\mathrm{Y}}^{(i)})X_2^{(i)}$$

Equation 10

The equation of the perceptron learning rule can be easily tested by adding weights, in the equations 11 and 12 are shown the perceptron predictions when the class labels were correctly classified and the weights will remain the same and none change will occur:

$$\Delta w_j = \eta\left(-1^{(i)} - -1^{(i)}\right)x_j^{(i)} = 0$$

Equation 11

$$\Delta w_j = \eta\left(1^{(i)} - 1^{(i)}\right)x_j^{(i)} = 0$$

Equation 12

Sometimes problems occur when weights are over-pushed towards positive or negative positions respectively to the target:

$$\Delta w_j = \eta\left(1^{(i)} - -1^{(i)}\right)x_j^{(i)} = \eta(2)x_j^{(i)}$$

Equation 13

$$\Delta w_j = \eta\left(-1^{(i)} - 1^{(i)}\right)x_j^{(i)} = \eta(-2)x_j^{(i)}$$

Equation 14

The Convergence of the perceptron only possible with two linearly separable classes, if those two classes cannot be split by a linear decision boundary into two groups, the maximum passes (threshold) of the given dataset should be defined for the most accuracy of the decision boundary.

1.3. Adaptive Linear Neurons and the Delta Rule

The Adaptive Linear Neuron (most known as Adaline) is a binary classification algorithm and a single layer neural network. Adaline was published by Bernard Widrow and his doctoral student Tedd Hoff (in 1960) after Rosenblatt's perceptron algorithm.

Importance of Adaline is for better understanding and improvement of machine learning algorithms in the main concept by acquiring minimized the continuous cost function. The difference between Perceptron and Adaline is that the weights in Adaline are updating on a linear activation function (fig. 7), where with perceptron the unit step function is used.

Linear activation function (equation 15) in Adaline is the net input equals to its own identity function (equation 16):

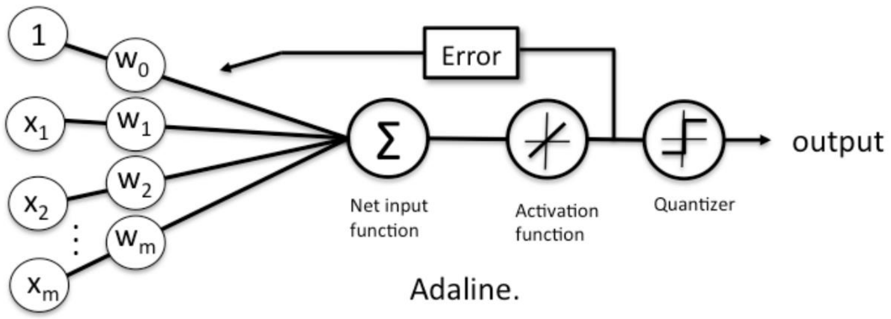


Figure 7

$$g(z) = z$$

Equation 15

$$z = w^T x$$

Equation 16

The most important advantage the linear activation function has over the perceptron unit step function is that it is differentiable [X]. The ability to differentiate allows to minimize the cost function J(w) in order to increment and update given weights. The cost activation function J(w) is defined as the sum of squared errors (SSE), which is similar to the cost function that is minimized in ordinary least squares (OLS) linear regression [X] as shown in the equation 19.

$$J(w) = \frac{1}{2}\sum_{i}(\mathrm{Y}^{(i)} - \hat{\mathrm{Y}}^{(i)})^2$$

Equation 17

Gradient descent is used for minimization of SSE cost function which is mostly use in machine learning field to find the local minimum of linear systems.

To understand how gradient descent is able to find local minimum, the convex function for one single weight must be generated and as shown in the figure 8, gradient descent is moving down until global or local minimum is found. At every step, the opposite direction of the gradient is taken, and the step size is determined by the value of the learning rate as well as the slope of the gradient.

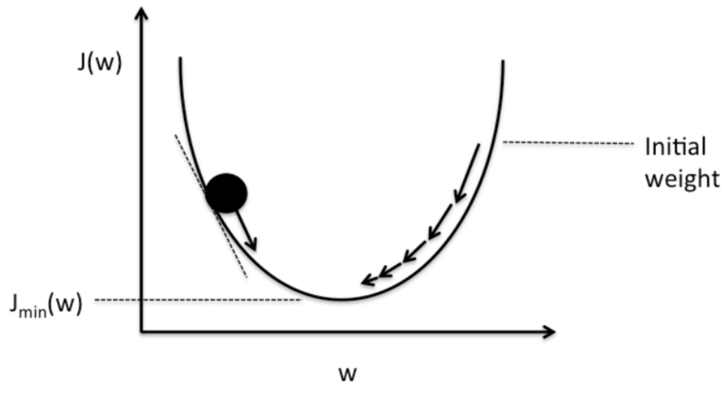


Figure 8

To derive the Adaline learning rule, as shown previously, each step is updated and taken by moving into opposite direction of the gradient $\Delta w = -\eta\nabla J(w)$, hence, it is necessary to calculate the partial derivative of the cost function for each weight in the weight vector: $\Delta w_j = -\eta \frac{\partial J}{\partial wj}$.

The calculation of partial derivative for the SSE cost function of the exact weight can be obtained (where Y = target, o = output) as follows:

$$\frac{\partial J}{\partial w} = \frac{1}{2}\frac{\partial}{\partial w_j}\sum_i \left(\mathrm{Y}^{(i)} - o^{(i)}\right)^2 = \frac{1}{2}\sum_i \frac{\partial}{\partial w_j}(\mathrm{Y}^{(i)} - o^{(i)})^2 = \frac{1}{2}\sum_i 2\left(\mathrm{Y}^{(i)} - o^{(i)}\right)\frac{\partial}{\partial w_j}\left(\mathrm{Y}^{(i)} - o^{(i)}\right)^2$$
$$= \sum_i \left(\mathrm{Y}^{(i)} - o^{(i)}\right)\frac{\partial}{\partial w_j}\left(\mathrm{Y}^{(i)} - \sum_i w_j x_j^{(i)}\right) = \sum_i \left(\mathrm{Y}^{(i)} - o^{(i)}\right)(x_j^{(i)})$$

Equation 18

By moving in direction $\left(\mathrm{Y}^{(i)} - o^{(i)}\right)(x_j^{(i)})$ increases error, so the opposite direction $\left(o^{(i)} - \mathrm{Y}^{(i)}\right)(x_j^{(i)})$

By adding obtained result into learning rate the equation is derived as follows:

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = -\eta\sum_i \left(t^{(i)} - o^{(i)}\right)\left(-x_j^{(i)}\right) = \eta\sum_i \left(t^{(i)} - o^{(i)}\right)x_j^{(i)}$$

Equation 19

After calculation, simultaneous weight update can be applied with perceptron rule:

$$w_j = w_j + \Delta w_j$$

Equation 20

The learning rule and the perceptron rule are identical, however, there are two main differences why it cannot be considered the same:

1. The symbol "o" has value as a real number and not a class label as in the perceptron learning rule.
2. The weight update is calculated based on all samples in the training set (instead of updating the weights incrementally after each sample), which is why this approach is also called "batch" gradient descent.

1.4. Multi-layer neural networks

A multi-layered perceptron (abbreviation MLP) is mostly used neural network in deep learning from all known neural networks even nowadays. MLP ability to solve the variety of problems is highly preferable in many areas such as stock prediction and analysis, classification and identification of images, unwanted detection of spam, and even in voting predictions. The story of multi-layered perceptron has begun in 1986 when Geoffrey Hinton, David Rumelhart, and Ronald Williams published a paper "Learning representations by back-propagating errors", which introduced:

1. Backpropagation, a procedure to repeatedly adjust the weights so as to minimize the difference between actual output and desired output

2. Hidden Layers, which are neuron nodes stacked in between inputs and outputs, allowing neural networks to learn more complicated features

Explaining multi-layer neural networks and obtaining output for input data set $X\colon x_1, x_2, \ldots, x_m$, with weight set $W^1 : w\frac{1}{1}, w\frac{1}{2}, \ldots, w\frac{1}{m}$, by summing up given data and adding bias result is:

$$z = \sum_{i=1}^{m} w_i x_i + bias$$

Equation 21

After obtaining numerical value of *z* an activation function *f(z)* is applied for the first hidden layer's neuron $h\frac{1}{1}$, where $h\frac{1}{1} = f(z)$ and $f(z)$ cannot be step function. The calculation must be repeated for the second hidden layer's neuron $h\frac{1}{2}$ which is construct by $W^2 : w\frac{2}{1}, w\frac{2}{2}, \ldots, w\frac{2}{m}$. It must be applied for every neuron using input data $W^n : w\frac{n}{1}, w\frac{n}{2}, \ldots, w\frac{n}{m}$, then action is continuously used for other hidden layers and neurons as shown in figure 9 to obtain final output $\hat{y}$.

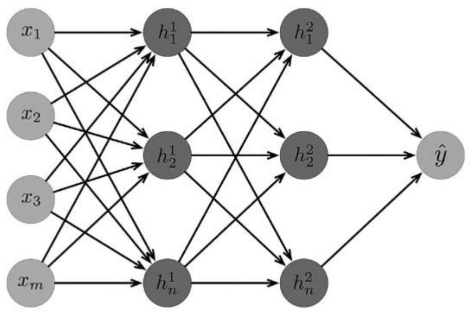


Figure 9. Neural network with 2 hidden layers []

By explaining why step function is not usable in multi-layer neural networks, because result will be only 0 or 1, where other activation functions such as sigmoid, tanh or relu are able to extract value between 0 and 1 which gives possibility to group features by numerical values. This feature is especially helpful in prediction of the output where result is decided by set of the value and the only way to be approved is to reach the fixed boundaries. By using sigmoid activation function, everything that is in range between 0 and 1 without any negative value, output is obtained by sigmoid properties:

1. The function itself is differentiable (the slope of two given points in the sigmoid curve is easily found).
2. The derivative of the function is not monotonic, but the function itself is monotonic.

3. Continuous phase of the output when $0 \leq f(z) \leq 1$.
4. Whenever output stays positive and by summing up obtained outputs value is equal to 1, $0 \leq f(z) \leq 1$

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

Equation 22

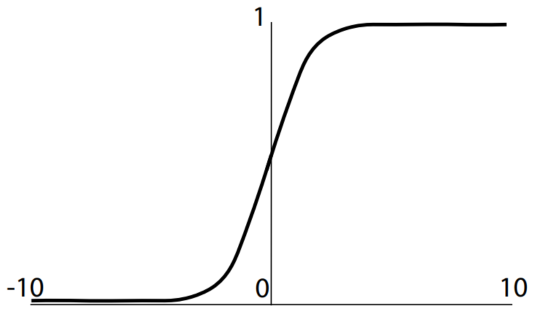


Figure 10. Sigmoid (logistic regression) function []

Other activation function is tanh (hyperbolic tangent activation function). Tanh has the same features such sigmoid, but the function range is from -1 to 1, and tanh has the same shape (stays sigmoidal) as sigmoid which allows to extract values in the same way and the negative value of the inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph []:

1. The function itself is differentiable (the slope of two given points in the sigmoid curve is easily found).
2. The derivative of the function is not monotonic, but the function itself is monotonic.
3. The tanh function is mainly used classification between two classes [].
4. Continuous phase of the output when, $-1 \leq f(z) \leq 1$

$$f(z) = \tanh(z)$$

Equation 23

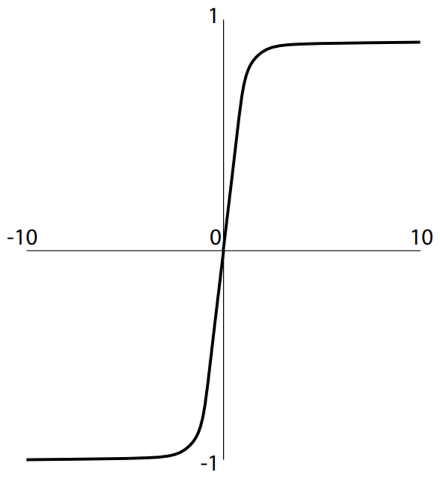


Figure 11. Tanh (hyperbolic tangent) activation function []

The most popular activation function is ReLU (rectified linear unit) activation function. The ReLU is half rectified it means that the function with negative value is always obtaining 0, when function is positive and value is above or equal to zero, function produce numerical value (equation 24) and it has range from 0 to infinity. Some other features of rectified linear unit activation function:

1. The function and its derivative both are monotonic.
2. Any feed with positive numerical value returns the same
3. ReLu overcomes the vanishing gradient problem, allowing models to learn faster and perform better.

$$f(\boldsymbol{z}) = \begin{cases} 0 & if\ x < 0 \\ x & if\ x \geq 0 \end{cases}$$

Equation 24.

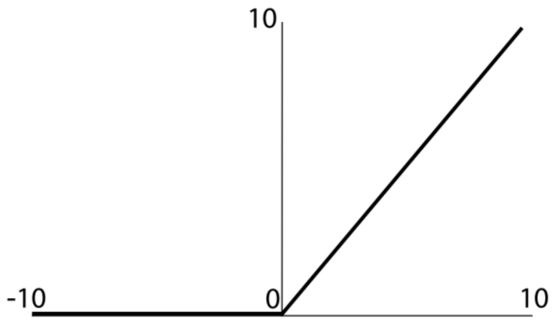


Figure 12. ReLU (rectified linear unit) activation function []

The issue of the rectified linear unit activation function is inability to processed all the given data, because all values with negative output is vanished and the model is unable to properly use data and train from it. Thus, data is deleted immediately, ReLu activation function plots graph by not mapping the negative values appropriately.

Comparing Adaline (use linear unit step activation function in the figure 13. pic. 1) and multi-layer neural networks (use non-linear activation function such as sigmoid, tanh, relu in the figure 13. pic. 2), the obtained result shows that with multi-layer neural network complex problems are solved much better and it gives possibility to create complex, non-linear decision boundaries that allow us to tackle problems where the different classes are not linearly separable.

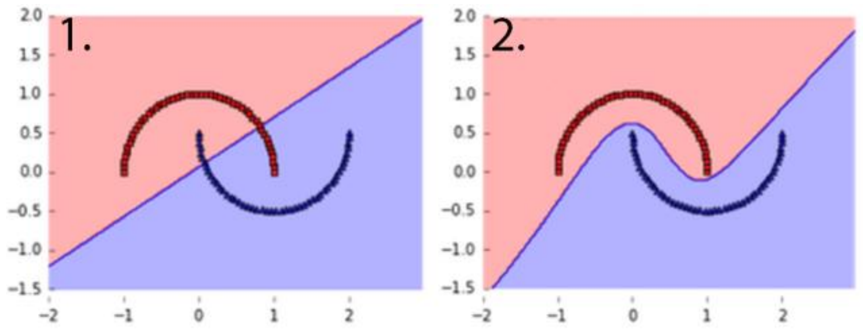


Figure 13. Comparison of Adaline (1) and multi-layer neural networks ability (2) to solve problems.

1.5. Recurrent neural network

Recurrent neural networks (abbreviation RNN) is the algorithm of sequential data, this is the first algorithm that remembers its input, due to an internal memory, which makes it perfectly suited for machine learning problems that involve sequential data. RNN history begins in 1980, however, lack of computational power recurrent neural networks were not able to spread in popularity and were not very applicable. After ten years, in 1990, when computers became more powerful along with necessity of data collection and new RNN's model were created - long short-term memory (LSTM), which started to show real potential of RNN application. Internal memory capacity allows RNN's to recognize and remember important information from the given input and reuse it whenever acquired data is useful. This feature allows to produce very accurate and precise output and predict what choice should be selected next that is why RNN is the preferred algorithm for sequential data like time series, speech, text, financial data, audio, video, weather and much more. The sequence and context can be more analyzed and understood much deeper comparing with other algorithms, because of its hidden state and information flow typically as follows (figure 14):

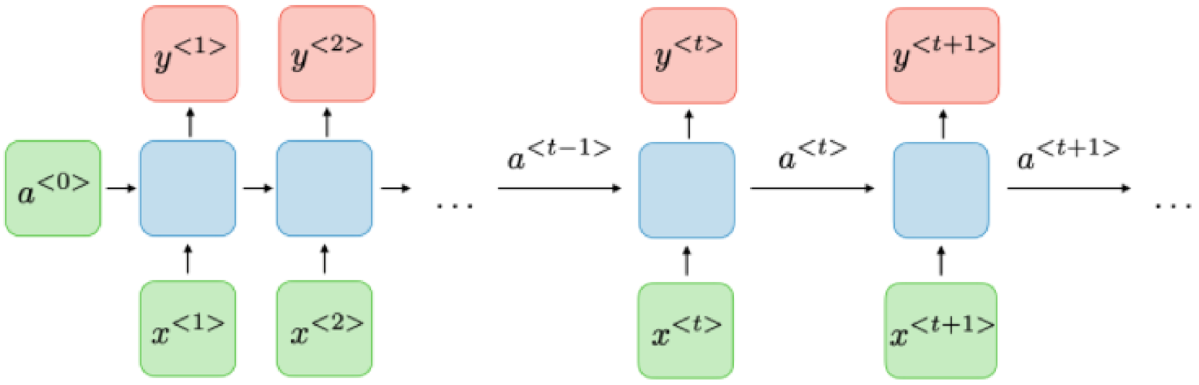


Figure 14.

For each timestep *t*, the activation a^t and the output y^t are expressed as follows (equation 25 and 26):

$$a^t = g_1(W_{aa}a^{(t-1)} + W_{ax}x^t + b_a)$$

Equation 25.

$$y^t = g_2(W_{ya}a^{(t)} + b_a)$$

Equation 26.

Where $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ – coefficients, g_1, g_2, - activation functions [].

A recurrent neural network can repeatedly teach itself with multiple generated product in the same node by going through cycles and passing a message. Analyzing the recurrent neural network's recursive relationship, its process must be divide into three unit weights: one for the inputs $x^{(t)}$, another for the outputs of the previous time step $y^{(t)}$, and the other for the output of the current time step $a^{(t)}$ []. The weights are the same as in perceptron or multi-layer neural network, and those parameters transforms data form input to network hidden layer in training process (figure 15):

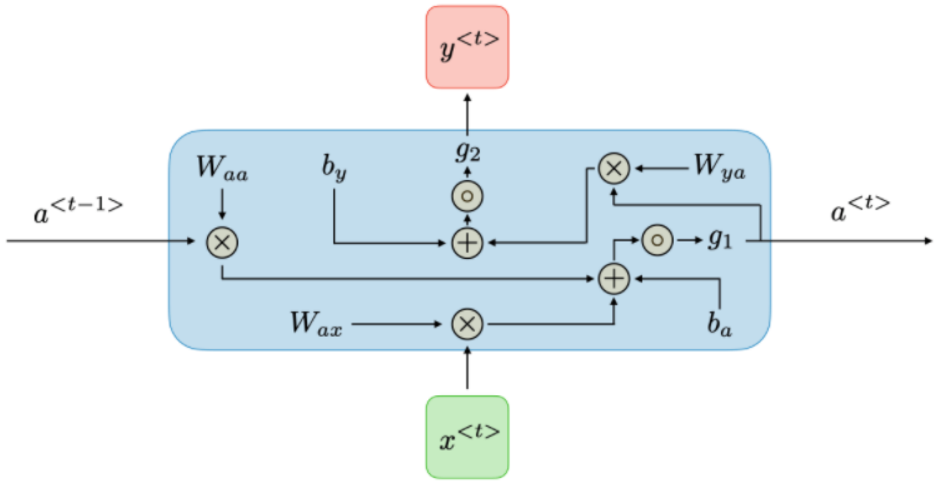


Figure 15.

To optimize algorithm, the function evaluation in the set of weights is referred as the objective function. The method to minimize the error where the inputs in objective function are increase or decrease in order to match distribution of the target variable and optimize the process, called a cost function (loss function), and numerical value calculated in that difference is named as loss. Cost function in the recurrent neural network is defined in all timesteps as the loss for every timestep as follows (equation 27) []:

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}(\hat{y}^{\boldsymbol{t}}, y^t)$$

Equation 27

Other necessary application of recurrent neural network is backpropagation through time (abbreviation BPTT), it is a process of the backpropagation training algorithm in RNN where the sequence is using gradient descent to calculate an error. A recurrent neural network is shown one input each timestep and predicts one output. It works by unrolling all input timesteps, each timestep has one input timestep, one copy of the network, and one output [https://machinelearningmastery.com/gentle-introduction-backpropagation-time/]. Errors are formed during each timestep and afterwards calculation is applied. The whole system's loop is back to the start and weights are upgraded. Backpropagation is done at each point in time, at timestep T, the derivative of the loss $\mathcal{L}$ with respect to weight matrix W is expressed as follows (equation 28):

$$\frac{\partial \mathcal{L}^T}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}^T}{\partial W}$$

Equation 28 []

In purpose to solve gradient problem particular gates can be used for any kind of RNN implementation. Gates are designated as gamma Γ and are written in form as follows (equation 29):

$$\Gamma = \sigma(Wx^t + Ua^{(t-1)} + b$$

Equation 29 []

Where W, U, b are coefficients specific to the gate and σ is the sigmoid function []. Most common used are shown in the table 1:

Type of gate	Role	Used in
Update gate Γ_u	How much past should matter now?	GRU, LSTM
Relevance gate Γ_r	Drop previous information?	GRU, LSTM
Forget gate Γ_f	Erase a cell or not?	LSTM
Output gate Γ_o	How much to reveal of a cell?	LSTM

Table 1. []

Two types of gating are used – Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) to handle the vanishing gradient problem in RNN, where GRU is not that complex, has forget gate, is faster, however it has less parameters comparing to LSTM (it has output gate). The structure and equations of GRU and LSTM are listed in the table 2.

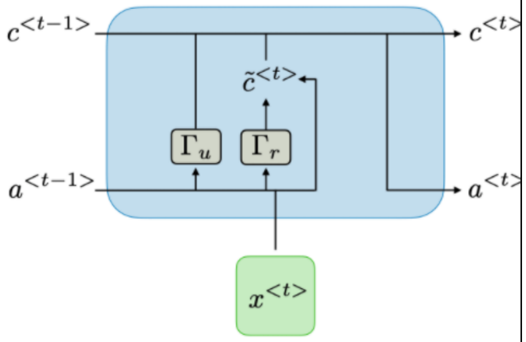
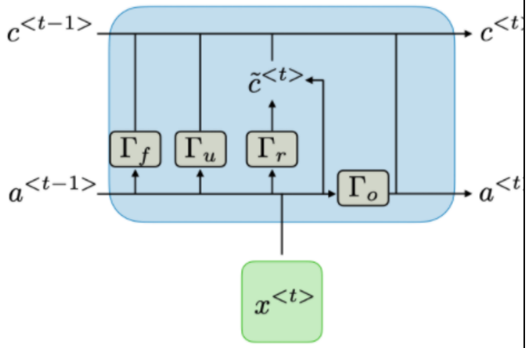
Characterization	Gated Recurrent Unit (GRU)	Long Short-Term Memory (LSTM)
$\hat{c}$	$\tanh(W_c[\Gamma_r * a^{(t-1)}, x^t + b_c)$	$\tanh(W_c[\Gamma_r * a^{(t-1)}, x^t + b_c)$
c^t	$\Gamma_u * \hat{c}^t + (1 - \Gamma_u) * c^{(t-1)}$	$\Gamma = \sigma(Wx^t + Ua^{(t-1)} + b$
a^t	c^t	$\Gamma_o * c^t$
Dependencies	 $c^{<t-1>}$ $c^{<t>}$ $\tilde{c}^{<t>}$ Γ_u Γ_r $a^{<t-1>}$ $a^{<t>}$ $x^{<t>}$	 $c^{<t-1>}$ $c^{<t}$ $\tilde{c}^{<t>}$ Γ_f Γ_u Γ_r Γ_o $a^{<t-1>}$ $a^{<t}$ $x^{<t>}$

Table 2.[]

Recurrent neural networks are solving most difficult problems in nowadays artificial intelligence world, its applications provide more efficient and accessible solutions for IT and engineering infrastructures, however, gradient vanishing problem remains important, where information vanish and become insignificant in a long term. Some pros and cons are listed in table 3:

Advantages	Drawbacks
• Possibility of processing input of any length • Model size not increasing with size of input • Computation takes into account historical information • Weights are shared across time	• Computation being slow • Difficulty of accessing information from a long time ago • Cannot consider any future input for the current state

Table 3 []

Tensorflow

Machine learning is a complex web of mathematical calculations, data flows, system analyzes and other difficult representation of calculus, however, implementing models with existing libraries and frameworks is more convienient than calculate every step from beginning, one common used open source platform is Tensorflow created by Google. The process facilitation is uncomparible faster in terms of acquiring data, training models, serving predictions, and refining future results [https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html]. TensorFlow is an open source library which is being used for numerical computation and processes implementation and testing in machine learning. It creates complete loop between models input and computes output by using Python programming language to provide a convenient front-end API for building applications with the framework, while executing those applications in high-performance C++ language []. TensorFlow able to train and activate calculations in deep neural networks for digital classification, natural language processing and words predictions with embeddings, recurrent neural networks, image recognition solutions, sequence to sequence models for machine translation, natural language processing, and PDE (partial differential equation) based simulations [], image recognition and best of it, TensorFlow use same training models to provide production prediction at scale. Instead of dealing with the complex mathematical calculations of implementing algorithms, it gives possibility to fully focus on the overall problem, and not waste time for unnecessary details where Tensorflow can solved it immediately.

By understand Tensorflow functionality and how to derive to machine learning algorithm, tensor notion must be defined. Tensor in m-dimensional space is a mathematical object that has n indices and m^n components and obeys certain transformation rules, where each index of a tensor ranges over the number of dimensions of space [https://mathworld.wolfram.com/] in other words, multilear charts from a vector space into the real number is described as a tensor. Tensors are defined in mathematical concept as follows:

Scalars – single elements of a number fields (example no.1 shown in figure 1.). Expressed by the formula in equation 1, where ℝ is a real number, e_1 is base and c is a scalar.

$$f: \mathbb{R} \to \mathbb{R}, f(e_1) = c$$

Equation 30

Vectors – quanities with magnitude and directions (example no.2 shown in figure 1). Expressed by the formula in equation 2, where $\mathbb{R}$ is a real number, and $\mathbb{R}^n$ is an element size, e_1 is base, v_1 is a vector.

$$f : \mathbb{R}^n \rightarrow \mathbb{R}, f(e_1) = v_i$$

Equation 31

Matrices – collection of numbers in rows and columns (example no.3 shown in figure 1). Expressed by the formula in equation 2, where $\mathbb{R}^n, \mathbb{R}^m$ are element sizes, e_1, e_j are basis, A_{ij} is a scalar.

$$f : \mathbb{R}^n * \mathbb{R}^m \rightarrow \mathbb{R}, f\left(e_1, e_j\right) = A_{ij}$$

Equation 32

Multidimensional array of numbers – one-dimensional arrays presented in relational tables and matrices (example no.4 shown in figure 1). Expressed by the formula in equation 2, where $\mathbb{R}$ is a real number, $(V^* * V_i^*)$ is dual spaces (known as p copies) and $(V * V_i)$ and vector spaces (known as q copies) to derive multilinear maps in finite space.

$$f : (V^* * V_i^*) * (V * V_i) \rightarrow \mathbb{R}$$

Equation 33

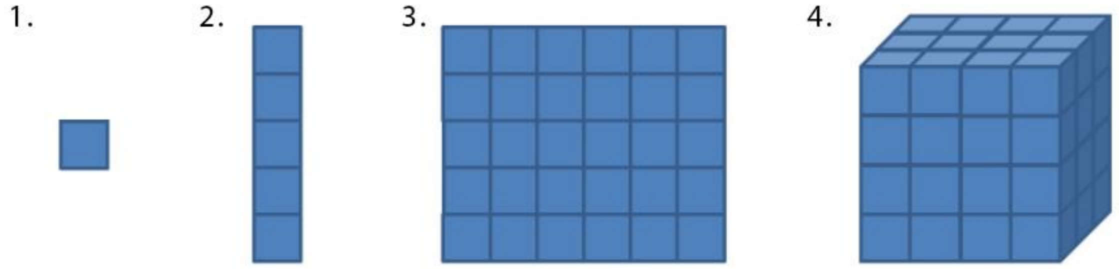
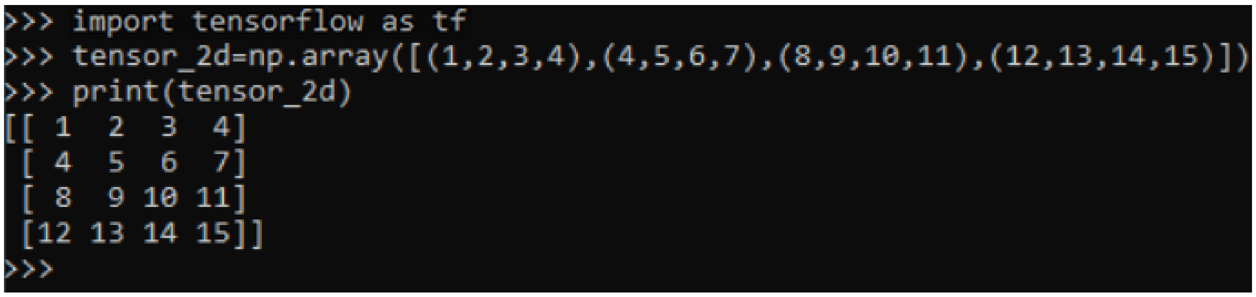


Figure 16

http://www.big-data.tips/what-is-a-tensor

To represent all given data, TensorFlow uses structure of tensor data, tensors are passing through and between operations for the computation graph. Tensor data structure in TensorFlow support a variety of element types, including signed and unsigned integers ranging in size from 8 bits to 64 bits, IEEE float and double types, a complex number type, and a string type (an arbitrary byte array) []. For instance, if the array of the numbers are defined, and importing tensorflow, numbers array can easily be presented as matrix and be the input for further calculations (as shown in the fig. 2).



```
>>> import tensorflow as tf
>>> tensor_2d=np.array([(1,2,3,4),(4,5,6,7),(8,9,10,11),(12,13,14,15)])
>>> print(tensor_2d)
[[ 1  2  3  4]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
>>>
```

Figure 17

The Tensorflow‘s platfrom provides ability to define functions for tensors and automatically derive derivatives - Tensorflow is based on automatic differentation, where, by specifying graph in its operations, Tensorflow is automatically running the chain rule of calculus along the graph, with all setup of the derivatives of each specified operation (equation 5). In computer programming tensors are used as higher dimentional arrays to define abundance of data in the array of numbers.

$$\nabla_A f(A) \in \mathbb{R}^{m*n} = \begin{bmatrix} \frac{\partial f(A)}{\partial A_{11}} & \frac{\partial f(A)}{\partial A_{12}} & \cdots & \frac{\partial f(A)}{\partial A_{1n}} \\ \frac{\partial f(A)}{\partial A_{21}} & \frac{\partial f(A)}{\partial A_{22}} & \cdots & \frac{\partial f(A)}{\partial A_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f(A)}{\partial A_{m1}} & \frac{\partial f(A)}{\partial A_{m2}} & \cdots & \frac{\partial f(A)}{\partial A_{mm}} \end{bmatrix}$$

Equation 34 https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788390392/1/ch01lvl1sec9/calculus

The one more application of TensorFlow platform is to provide directed graph as the numeric computation. In the figure X is shown the computation graph of $\mathrm{h} = \mathrm{ReLU(Wx + b)}$, *where the graph is presented as system, in which the inputs of the data x and the output h, variables of weight W, bias b and the functions (ReLU, MatMul, Add).* This is important element in neural networks, which conducts an linear transformation of the input data and then feed to a linearity (rectified linear activation function in this case) [https://becominghuman.ai/an-introduction-to-tensorflow-f4f31e3ea1c0].

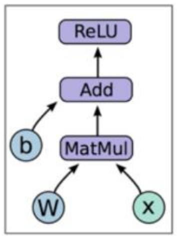
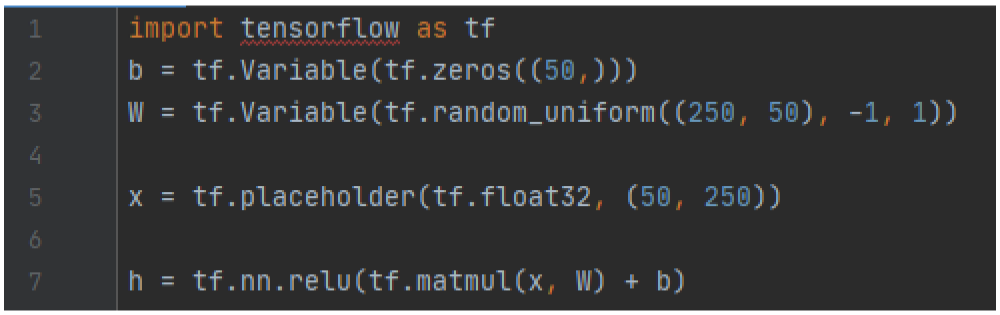


Figure 18

To implement the code on Tensorflow, variables must be expressly initialized. When variable is created, tensor is initializing process as the Variable() constructor. The initializer can be constants, sequences and random values []. In the code the bias vector b is set as constant and stays zero, the weights of the matrix is W and it initializes by random uniform. The shape of the tensor must be specified and shape is automatically converted as variable of the shape. In this instance, b is the first tensor with shape (50, 0) and W is the second tensor with shape (250, 50).



```
import tensorflow as tf
b = tf.Variable(tf.zeros((50,)))
W = tf.Variable(tf.random_uniform((250, 50), -1, 1))

x = tf.placeholder(tf.float32, (50, 250))

h = tf.nn.relu(tf.matmul(x, W) + b)
```

Figure 19

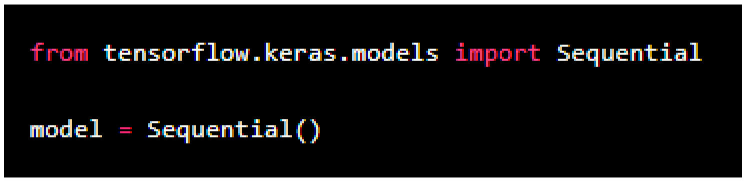
Further possibilities and performance of TensorFlow's calculations and implementation in real functional model will be explained in the corrosion detection model together with convolutional neural network.

Keras is advanced neural network library which is built on top of TensorFlow and has interface with neural connection in Python programming language. By using Tensorflow with Keras (it is API (application programming interface) for TensorFlow) it derives an approachable, convenient and productive interface for solving deep learning problems, and giving high-level feedback. Keras gives possibility to build neural networks in more convenient way without calculating algebra of tensors in mathematical aspects, numerical calculations and without methods of process optimization. It empowers of the extensibility and capacity to fully run Keras on TPU or on large clusters of GPUs [], moreover, Keras allows to process and run exported models by using mobile devices or browser. Two main factors of Keras usability are distinguished as written:

- Keras is an API created for better understanding for developer. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error. []

- This ease of use does not come at the cost of reduced flexibility: because Keras integrates deeply with low-level TensorFlow functionality, it enables to develop highly hackable workflows where any piece of functionality can be customized. []

The key of appropriate calculations in Keras are models and layers. The most usable and uncomplicated model is known as the Sequential model which is based on the layers of linear stack. Keras functional API is used for difficult and multiplex architectures, which empowers to construct the arbitrary graphs of layers, or write models entirely from scratch via subclasssing (shown in figure 5). [].

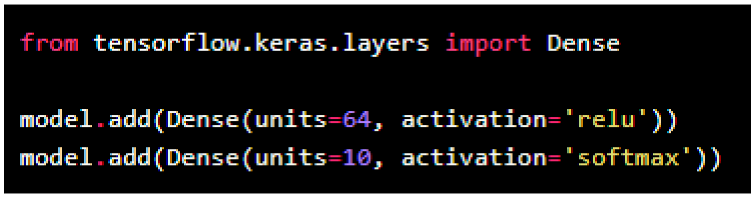


```
from tensorflow.keras.models import Sequential

model = Sequential()
```

Figure 20

The process of layers stacking and selection of activation function by using method .add():

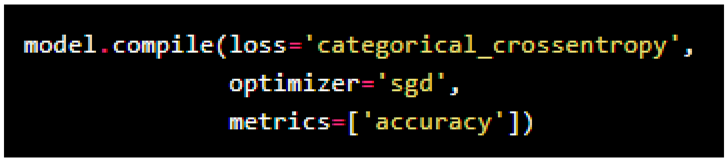


```
from tensorflow.keras.layers import Dense

model.add(Dense(units=64, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
```

Figure 21

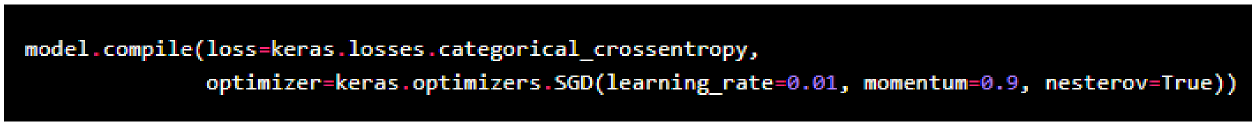
Learning process implementation by using compile(), which is used for loss function of the crossentropy, selection of optimizers and metrics:



```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

Figure 22

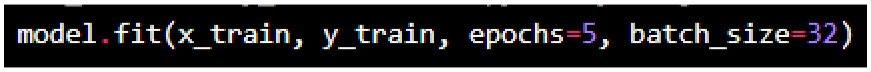
The configuration process of the optimizer (the concept of optimizers is defined as algorithms or methods used to change the attributes of the neural network such as weights and learning rate to reduce the losses. The main key of using optimizers is to solve optimization problems by minimizing the function []). In this way Keras is helping to fully control optimizer’s change of learning rate and momentum to find the solution to solve any problem, where in the process, source code is more extensible than subclassing []:



```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9, nesterov=True))
```

Figure 23

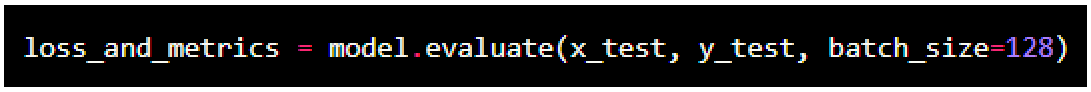
The iteration process of trained data in batches by epochs (the concept of epoch is summarized as the full circle of the entire dataset when it is passing forward and backward neural network once, and divisions of the batches are used for reason of the size of epoch which is just too large to be fed into computer at once []):



```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Figure 24

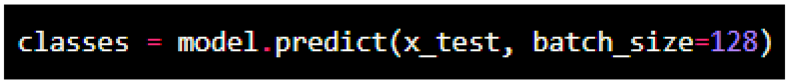
The evaluation of model loss and metrics:



```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

Figure 25

The generation process of predictions over new data:



```
classes = model.predict(x_test, batch_size=128)
```

Figure 26