ARTIFICIAL NEURAL NETWORKS AND DEEP LEARNING. A SIMPLE OVERVIEW

Method · December 2019

CITATIONS
0

READS
3,822

1 author:

Paolo Dell'Aversana
Eni SpA - San Donato Milanese (MI)
168 PUBLICATIONS **1,054** CITATIONS

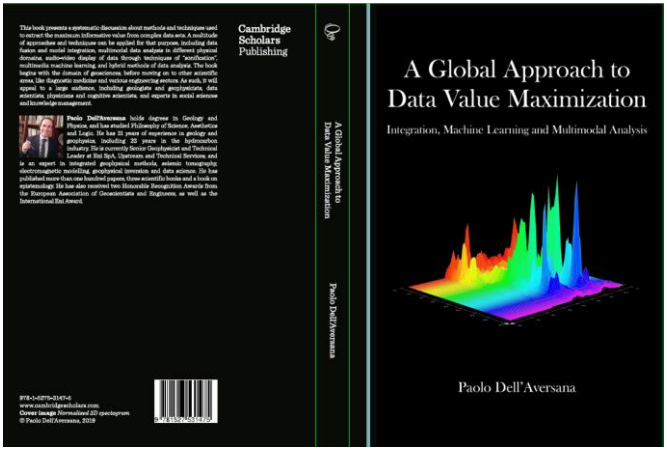
SEE PROFILE

ARTIFICIAL NEURAL NETWORKS AND DEEP LEARNING.

A SIMPLE OVERVIEW

PAOLO DELL'AVERSANA

In this overview, I summarize some aspects of Artificial Neural Networks (ANNs) and Deep Learning (DL). My goal is to provide the basic background for novices in Machine Learning and Artificial Intelligence. In the references, I recommend additional readings for who is interested in the details of this interesting subject. This brief manuscript is extrated from my book "A Global Approach to Data Value Maximization. Integration, Machine Learning and Multimodal Analysis" (Dell'Aversana, P., 2019 – Cambridge Scholars Publishing)



A brief history of ANNs: from the beginning to the first critiques

Artificial Neural Networks (ANNs) are inspired to biological neural networks forming the brain. However, while mathematical algorithms are well suited for linear programming, arithmetic and logic calculations, ANNs are more effective to solve problems related to pattern recognition and matching, clustering and classification.

The development of the first ANN was based on a very simple model of neural connections. The "Mark I Perceptron" machine, created by the neurobiologist Frank Rosenblatt, assumes that the artificial connections between neurons could change through a supervised learning process (Rosenblatt, 1957) that reduces the misfit between actual and expected output. The expected output comes from a training data set. That discrepancy is back-propagated through the entire network and allows updating the weights of the connections. In other words, the misfit between the actual and expected responses of the network represents the necessary information for improving the learning performance.

Figure A5-1 shows schematically how the perceptron receives the inputs x_1, x_2, …, x_m (the upper left box with symbol "1" represents a bias in the *input* data). These are combined with the *weights* w_i in order to compute the *Net input* function. In turn, this is passed on to the *activation function* (here: the unit step function) that produces a *binary output* -1 or +1 that corresponds (in this binary classification example) to the predicted class label of the sample. The output itself is used during the learning phase to calculate the error of the prediction. This is back-propagated in order to update the weights for reducing the misfit between the actual and the desired output.

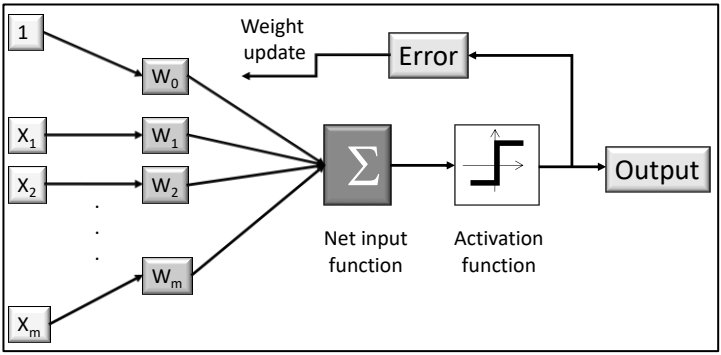


Figure A5-1. The Perceptron workflow. After Raschka and Mirjalili, 2017 (modified).

About in the same period John von Neuman (1958) introduced the idea to explain the behavior of neurons in terms of digital units: every neuron can be "On" or "Off. In case of activation, it transmits signals to other neurons forming a neural aggregate. Following Neuman's approach, neurons' activity would be fully described by a binary behavior: the activity of an entire neural population can be formalized through an ensemble of logic binary functions working in parallel.

As we will see, a step forwards was obtained in 1960 by Bernard Widrow and Marcian Edward Hoff at Stanford University. They created two neural network systems: ADALINE (Adaptive Linear Neuron) and MADALINE (Multiple Adaptive Linear Neuron). These were able to produce the first practical results in the field of signal processing and noise suppression in telephone communications using the concepts of ANNs.

Alternate successes and failures characterized the history of the Artificial Neural Networks until 1985. In that year the American Institute of Physics started the Annual Conferences of Neural Networks for Computing. Two years later, the conference was organized in San Diego by the Institute of Electrical and Electronic Engineers and was attended by almost two thousand participants. However, despite the initial successes, part of the scientific community expressed high criticism versus the fundamentals of ANNs (Fodor and Pylyshyn, 1988). I have summarized that critical view in my book Cognition in Geosciences (2013, paragraph 1.2.6: Critiques to Connectionisms). It starts "… from two fundamentally different visions of intelligence expressed by the so-called "Computationists" and "Connectionists". The crucial difference between classical computational systems and neural networks is that the first type of system is intrinsically sequential, whereas neural networks operate with a parallel architecture. This difference has a strong impact on the conception of intelligence and is the basis for the two opposite approaches to the study of the human mind. In the connectionist systems, the intelligence itself stays in the weights of the connections between the units forming the network. It means that the system can learn by iterative update of the strength of the connections. On the other hand, the Computationists state that the human mind works by applying continuous computations and the brain behaves as a calculator. Jerry Fodor was one of the most critical opponents to connectionism. In his 1975 book, "The Language of Thought", he criticizes the basis of the whole approach to Artificial Intelligence and not only the neural networks. This approach based on modelling and simulation is wrong, according to Fodor, in the same way as it was inappropriate to study the physical word by attempting to reproduce

it. Classical physics, for instance, is not the tool to build a machine similar to the real world. Although the critiques of Fodor and other Computationists are useful for highlighting the limitations of connectionism and of neural networks, the idea that the human mind works like a serial computation system is misleading and inappropriate…" (Dell'Aversana, 2013).

One of the hardest critiques to ANNs came from Marvin Minsky, founder of the MIT AI Lab, and Seymour Papert, director of the lab at the time. They performed a rigorous analysis of the limitations of Perceptrons and published their results in a seminal book (Minsky & Papert, 1969). In their book, the authors highlighted the intrinsic limits of a Perceptron, such as its incapability to learn the simple Boolean function XOR because it is not linearly separable. This publication is widely believed to have triggered and supported a long period of disillusionment about ANNs that causes a freeze to funding and publications.

The Perceptron's algorithm

In order to understand how the Perceptron evolved over the years, let us transform the workflow of figure A5-1 into a formal algorithm. We can suppose that we have to solve a problem of binary classification. For instance, our objective can be to classify a set of medical observations into "diagnosis 1" and "diagnosis 2", or briefly Class 1 and Class 2. We define the *activation function* $\Phi(z)$ that takes a linear combination z of our vector of input values $\mathbf{X} = [x_1, x_2, \ldots, x_m]^T$ and corresponding vector of weights $\mathbf{W} = [w_1, w_2, \ldots, w_m]^T$.

The linear combination z is the so-called *net input*:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_m x_m \quad . \tag{A5-1}$$

In figure 1, the activation function is simply the *unit step function*. If the activation function is greater than a certain threshold,Θ, then the activation function gives in output 1. Otherwise, it gives -1.

$$\Phi(z) = \begin{cases} 1 \; if \; z \geq \Theta \\ -1 \; otherwise \end{cases} . \tag{A5-2}$$

In the simplistic approach of the initial Perceptron, the basic idea is that a neuron has just a binary choice: either it fires or it does not. Consequently, the Perceptron's algorithm can be summarized in the following sequence of steps:

1) Initialize the weights to 0 or to small random numbers.
2) Prepare a training (labelled) data set and use it as input for the Perceptron.
3) Compute the output value, y_{out} using the unit step function.
4) Compare the output y_{out} with the desired output y_{true} (the true class label for the selected training sample).
5) Use the error (the difference between y_{true} and y_{out}) for updating the weights.
6) Iterate until the error is below a certain desired threshold.

These steps can be formalized as following.

The output value is the class label predicted by the unit step function. The simultaneous update of each weight w_j in the weight vector **w** can be formally written as:

$$w_j = w_j + \Delta w_j. \quad \text{(A5-3)}$$

The update Δw_j is

$$\Delta w_j = \eta\left(y_{true}^{(i)} - y_{out}^{(i)}\right)x_j^{(i)}. \quad \text{(A5-4)}$$

The formula (A5-4) is commonly indicated as the *Perceptron learning rule*. It allows updating the weights in such a way to decrease the error. However, the convergence is only guaranteed (the weights will stop updating after a finite number of iterations) if the two classes are linearly separable and if the "learning rate" η (a constant value between 0.0 and 1.0) is sufficiently small.

Linear separability means that classes of patterns of data can be separated with a single decision surface. This is easy to visualize in two dimensions. For instance, let us think one set of data represented by circles in the plane and another set represented by triangles. These two sets are linearly separable if there exists at least one line in the plane with all of the circles on one side of the line and all the triangles points on the other side. In higher-dimensional Euclidean spaces the same problems is generalized if the line is replaced by hyperplane. Its intrinsic limitation to linearly separable classes represented the main reason why the Perceptron met severe critiques.

Improving the Perceptron

Bernard Widrow (1960) and his doctoral student Tedd Hoff proposed an improvement of the Perceptron idea. They introduced another type of single-layer neural network: ADAptive LInear NEuron (Adaline). It is based on the key concepts of defining and minimizing cost functions. Differently from Rosenblatt's perceptron, in Adaline's approach the weights are updated based on a linear activation function, $\varphi(z)$, rather than a unit step function. That activation function is simply the identity function of the net input, so that $\varphi(\boldsymbol{w}^T\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$. It is followed by the application of a Quantizer for producing a binary output (1, -1). Figure A5-2 shows a scheme of the Adaline workflow.

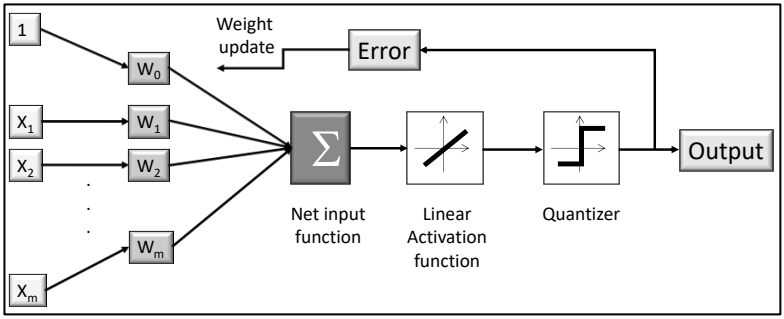


Figure A5-2. Scheme of the Adaline workflow. After Raschka and Mirjalili, 2017 (modified).

The weights' update is realized by minimizing a cost function, $J(\boldsymbol{w})$, defined by the Sum of Squared Errors (SSE) between the calculated outcome and the true class label.

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_i \left(y_{true}^{(i)} - \varphi\left(z^{(i)}\right)\right)^2 . \qquad \text{(A5-5)}$$

The linear activation function $\varphi(z)$ used in the Adaline approach introduces the simple, but important benefit to be differentiable (in contrast to the unit step function). Thus, we can compute the gradient of the cost function by calculating the partial derivatives of the cost function with respect to each weight. It means that the weight update is calculated based on all samples in the training set (instead of updating the weights incrementally after each sample).

An additional advantage is that the linear activation function allows the cost function (A5-5) being convex. Thus, we can use "simple" optimization algorithms, such as the gradient descent[1], to minimize it, possibly finding global cost minimum[2].

The update of the weight vector **w** is:

$$\Delta \mathbf{w} = -\eta \, \Delta \mathrm{J}(\mathbf{w}) \quad . \tag{A5-6}$$

The update of the generic weight w_j is.

$$\Delta w_j = \; -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y_{true}^{(i)} - \; \varphi\left(z^{(i)}\right) \right) x_j^{(i)} \quad . \tag{A5-7}$$

Logistic regression

A further improvement of the Perceptron's idea is Logistic regression. This classification model performs very well on linearly separable classes. Furthermore, it can be extended to multiclass classification. The idea of *Logistic regression* starts form the simple concept of *odd ratio*. This is the the odds in favour of a particular event

$$odd\ ratio = \frac{occurring}{not\ occurring} = \frac{p}{1-p} \quad . \tag{A5-8}$$

The *Logit function* is the log of the odd ratio.

$$logit(p) = \log \frac{p}{(1-p)} \, . \tag{A5-9}$$

Logit function transforms input in the range [0, 1] into real values. These values can be used to express a linear relationship between feature values and the log-odds:

$$logit\big(p(y = 1|x)\big) = \sum_{i=1}^{m} w_i x_i = \boldsymbol{w^T x}. \tag{A5-10}$$

[1] In the gradient descent method, one takes the update steps proportional to the negative of the gradient (or approximate gradient) of the cost function at the current point.

[2] Of course, in more complex minimization problems, we cannot exclude to find a local minimum, rather than a global one.

In the above formula, $\big(p(y=1|x)\big)$ is the conditional probability that a particular sample belongs to class 1 given its features vector **x**.

Now we must remember that we are interested in predicting the probability that a certain sample belongs to a particular class. As we said above, the Logit function does the opposite: it transforms probabilities into real values. Thus, we want to use the inverse of the Logit function. Indeed, the Logit function of a variable *z* is invertible and its inverse function is

$$\phi(z) = \frac{1}{(1+e^{-z})} = \frac{e^{z}}{(1+e^{z})} \ . \qquad \text{(A5-11)}$$

Here, the variable *z* is the net input that is the linear combination of weights and sample features.

In summary, we use the logistic function as a new type of continuous activation function. It allows taking real number values as input and transforming them to values in the range [0, 1] with an intercept at 0.5 (figure A5-3).

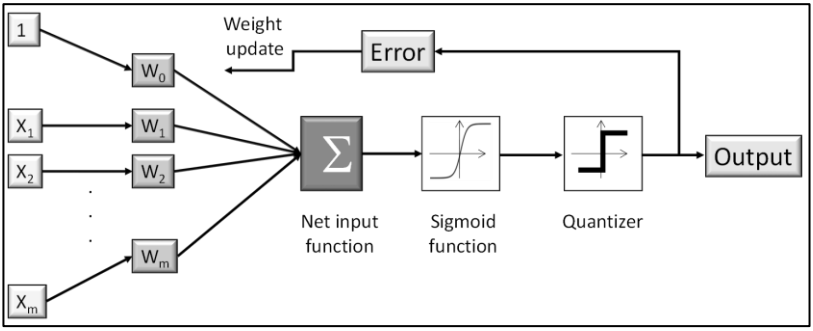


Figure A5-3. Scheme of the Adaline workflow using the sigmoid (Logistic) function. After Raschka and Mirjalili, 2017 (modified). In the figure, the Sigmoid function is just a special case of the logistic function.

Now it is possible to update the weights for the Logistic regression (using, for instance, the Sigmoid Activation function) by minimizing an objective function as we did in the case of the Linear Activation function.

As explained above, the weights' update is realized by minimizing a cost function, $J(\boldsymbol{w})$, defined by the Sum of Squared Errors (SSE) expressed by (A5-5). We remark that we are assuming that the individual samples in

our dataset are independent of one another. With that assumption, we want to maximize the following likelihood function L:

$$L(\boldsymbol{w}) = P(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{w}) = \prod_{i=1}^{n} P\left(y_{true}^{(i)}|x^{(i)};\boldsymbol{w}\right) = \\ = \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1-\phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}. \quad \text{(A5-12)}$$

It is easier to maximize the "natural log-likelihood function" of the (A5-12) (Raschka and Mirjalili, 2017).

$$log\ L(\boldsymbol{w}) = \sum_{i=1}^{n} log\left(\phi\left(z^{(i)}\right)\right) + \left(1-y_{true}^{(i)}\right) log\left(1-\phi\left(z^{(i)}\right)\right). \quad \text{(A5-13)}$$

For this purpose, we can use, for instance, a standard optimization algorithm such as the "Gradient ascent".

Multi-layer Artificial Neural Networks and Deep Learning

After the first implementation of the Rosenblatt's perceptron in the 1950s, a significant part of the scientific community progressively lose interest in neural networks. The main reason was that no one had a good solution for training a neural network with multiple layers in order to solve complex classification or recognition problems. Nowadays, this problem has been solved, thanks to new algorithms and to the improved computation capacities of modern systems. Neural networks composed by many hidden layers (Deep Neural Networks", or briefly DNN) can be successfully trained to solve difficult problems in both academic and industrial sectors. Consequently, ANN have reconquered large renewed interest.

The expression of "Deep Learning" is commonly used when talking about multilayer ANN. Current applications of DNN concern image and speech recognition, text recognition in images for real-time translation, drug discovery and toxicity prediction and many other applications in health industry. Furthermore, there is a growing number of applications in hydrocarbon industry, including exploration, reservoir characterization, monitoring during oil production and oil field development.

The basic idea of DNN is partially inspired to the hierarchical models of our visual system. This consists of neural networks distributed with a layered topology. The complex pathways in our brain goes from the retinas, to the visual cortex, to the occipital cortex, and finally they reach high-level associative areas. Along these paths, the receptive fields at one level of the hierarchy are constructed by combining inputs from units at a lower level.

Indeed, one of the strength points of DNN is their hierarchical organization. That layered organization allows sharing and reusing information. It is possible to select specific features and reject useless details.

The first type of multi-layer ANN architecture is the multi-layer perceptron (MLP). Three layers form the simplest type: one input layer, one hidden layer, and one output layer. Each unit in the hidden unit is connected to all units in the input layers, and the output layer is fully connected to the hidden layer.

In case there is more than one hidden layer, such a network is also called a Deep Artificial Neural Network. The learning workflow of MLP is schematically the following:

1) Forward propagation of the input patterns (training data vector) from the input layer through the network to generate an output.
2) Error calculation by comparing network output with desired output.
3) Error back-propagation, derivative calculation with respect to each weight in the entire network, and model update.

We iterate the above steps many times (epochs) until we obtain a proper convergence (error reduction below the desired threshold).

Nowadays we commonly use DNN's including a number of layers ranging from 7 to 50. Deeper networks (more than 100 layers) allows slightly better performances, but sacrificing computational effectiveness. The number of layers represents just one of the "hyper-parameters" of a DNN. The complexity of a network is given also by the number of neurons, connections and weighs. Every individual weight represents a parameter to be learn. Of course, the complexity of the training depends on the number of those weights.

There are various types of DNN's. Among the networks addressed to supervised learning, one of the most used is Convolutional Neural Network (briefly, ConvNet), introduced by LeCun et al. (1998). These became popular in computer vision due to their good performance in solving complex problems of image classification (Simard et al, 2003). The main differences with respect to the Multilayer Perceptron are the following (Figure A5-4):

1) *Local processing*: the neurons belonging to two successive layers are connected only locally. That type of local link allows reducing the number of connections and, consequentially, the computation complexity.
2) *Shared weights*: the weights are shared in groups, allowing significant reduction of the number of weighs.

3) *Alternation of the layers* for *feature extraction* and for *pooling*.

Let us explain better the three above concepts with the help of figure A5-4. In neural networks, each neuron receives input from a certain number of locations in the previous layer. We talk about "fully connected layer" when each neuron receives input from every element of the previous layer. The basic idea of "convolutional layer" is that neurons receive input from only a restricted subarea of the previous layer. That input area of a neuron is called its "receptive field". In other words, in a fully connected layer, the receptive field consists of the entire previous layer. Instead, in a convolutional layer, the receptive field is smaller than the entire previous layer. This concept emulates the functioning of the real response of individual neurons that is confined to their own visual stimuli[3].

This is an effective strategy for reducing the computation efforts. In fact, connecting all the neurons of two consecutive layers would not be practical. For instance, let us suppose that we have a small image. If in our ANN, we have the first neural layer of size 100 x 100, for having fully connected layers, we need 10000 weights for each neuron in the second layer. We need to set a more effective strategy for lowering the number of variables and the computational efforts of our ANN. For that purpose, we can apply an operation of *convolution*. This allows reducing the number of free parameter (the weights to be determined). In fact, each *convolutional neuron* processes data only for its "receptive field". This step is performed by connecting the input layer to a feature map. These receptive fields represent "overlapping windows" that we slide over the pixels of an input image (figure A5-5).

The second key concept is "pooling". This is a sort of down sampling of an image. The role of pooling layer is to reduce the resolution of the feature map but saving the features of the map required for classification through translational and rotational invariants. As schematically shown in Figure A5-4, pooling layers combine the outputs of neuron clusters at one layer into a single neuron in the next layer. Examples of pooling are "max pooling" and "average pooling". These use, respectively, the maximum and the average value from each of a cluster of neurons at the prior layer.

[3] The neurophysiologists Hubel and Wiesel in the 1950s and 1960s demonstrated that cat and monkey visual cortices contain neurons that individually respond to small regions of the visual field.

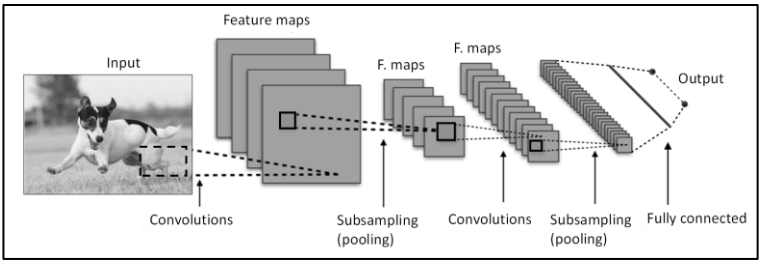


Figure A5-4. Scheme of Convolutional Neural Network.

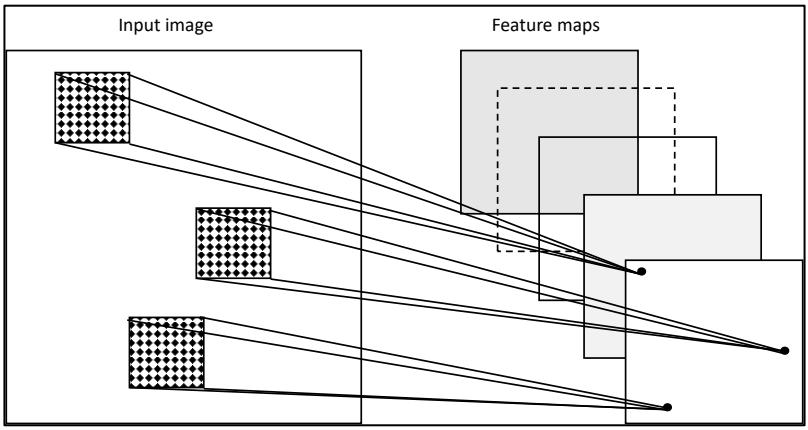


Figure A5-5. Example of convolutional layer.

An additional important concept of CNN Is that the receptive fields that map the features to the units in the next layer share the same weights. The reason is that a feature detector that was useful in one part of the image can be reused effectively in a different part. That approach allows reducing the number of unknown parameters that the machine needs to learn.

Finally, as shown in figure A5-4, after the sequence of convolutional and pooling layers, the last part of the CNN workflow is to create a fully connected output. This last step is performed by a multi-layer perceptron.

To conclude, I would like to suggest some references of books and open-source libraries of codes dedicated to neural network implementation, with particular reference to parallel architecture.

In Python language, a useful library is Theano (http://deeplearning.net/software/theano/) that allows building and training Deep Neural Networks.

Furthermore, a collection of tutorials about Theano is available at http://deeplearning.net/software/theano/tutorial/index.html#tutorial.

Finally, Raschka and Mirjalili (2017) provide a detailed discussion of how Parallelizing Neural Networks and training with Theano. These authors describe how to write and optimize Python machine learning codes, how to select the activation functions for artificial neural networks and how to use specific libraries for Deep Learning (such as Keras).

References

1. Dell'Aversana, P., 2013. Cognition in geosciences: the feeding loop between geo-disciplines, cognitive sciences and epistemology, EAGE Publications, Elsevier.
2. Fodor, J. and Pylyshyn, Z., 1988. Connectionism and cognitive architecture: A critical analysis. Cognition, 28, 3-71.
3. LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P., 1998. Gradient-based Learning Applied to Document Recognition. Proceedings of the IEEE, 86(11):2278-2324, November 1998.
4. Marvin Minsky, M. & Papert, S., 1969. Perceptrons. An Introduction to Computational Geometry. M.I.T. Press, Cambridge, Mass., 1969.
5. Raschka, S. and Mirjalili, V., 2017. Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow, 2nd Edition.
6. Rosenblatt, F., 1957. The Perceptron, a Perceiving and Recognizing Automaton. Cornell Aeronautical Laboratory.
7. Simard, P. Y., Steinkraus, D. and. Platt, J. C, 2003. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. IEEE, 2003, p.958.
8. von Neumann, J., 1958. The Computer and the Brain. New Haven/London: Yale University Press.
9. Widrow, B., 1960. Adaptive "Adaline" neuron using chemical "memistors". Number Technical Report 1553-2. Stanford Electron. Labs., Stanford, CA, October 1960.