RIGA TECHNICAL UNIVERSITY

Faculty of Information Technology and Computer Science

Institute of Applied Computer Systems

Department of Artificial Intelligence and System Engineering

Iurii Lepesevich

Academic Bachelor Study Program "Computer Systems"

Student ID 191ADB051

ANALYTICAL COMPARISON OF NORMALIZATION FUNCTIONS IN REGRESSION TASKS

Scientific adviser

Doctor of science, Researcher

Ēvalds, Urtāns

RIGA TECHNICAL UNIVERSITY

Faculty of Information Technology and Computer Science

Institute of Applied Computer Systems

Department of Artificial Intelligence and System Engineering

Work Performance and Assessment Sheet of the Bachelor Paper

The author of the graduation paper:

Student Iurii Lepesevich

_____ (signature, date)

The graduation paper has been approved for the defence:

Scientific adviser:

Doctor of Science, Researcher, Ēvalds, Urtāns

_____ (signature, date)

ABSTRACT IN ENGISH

ABSTRACT IN LATVIAN

TABLE OF CONTENTS

INTRODUCTION

Artificial Intelligence

Artificial Intelligence (AI) is a specific field in Computer Science that implies solving cognitive tasks that are created for human intelligence. Such tasks could be described as:

- Learning.
- Solving problems.
- Pattern recognition.

In modern world AI is widely used in different spheres. We can see development of this technology in such fields as healthcare, military, social field, etc.

For the last few years development in the discipline of statistical data has created such new domains as: Machine Learning (ML) and Deep Learning (DL). Those domains created background for further improvement of AI technology.

Deep Learning

Deep learning is a sub-field of AI. DL uses multilayer algorithms to deep analysis of inputted data. Such algorithms recognize patterns and find relationship between different features of data. If a huge dataset would be used, DL algorithm could recognize dependencies between individual variables.

As an example, image recognition could be given. DL is able to distinguish important parts of particular image and predict class of that image (such as is bird or fish is located on the image).

Deep learning models are widely used in such technologies as self-driving cars, cancer prediction, business production optimization,speech recognition, etc.

Aim

Aim of this paper is to perform analytical comparison of different normalization functions. It is required to create methodology so optimal normalization function would be applied in particular problem.

In this graduation paper – basic concepts of DL field will be described, several experiments will be performed and compared. Based on experiments results –

conclusion will be made and required guidelines for normalization function will be created.

Section Content

Firstly, theoretical background will be described. There will be information about basic concepts of DL field:

- Linear layers.
- Loss functions.
- Backpropagation algorithm.
- Optimization algorithms.
- Metrics (NRMSE, R-2 Score).
- Regression tasks.
- Importance of normalization functions.
- Learnable methods.
- Basic architecture.
- Datasets.

After theoretical part – experimental part will be shown. Experimental results will be verified in verification part.

1. DEEP MACHINE LEARNING

1.1. Basic architecture

1.1.1. Linear layer

1:5 Linear layer is a fundamental structure of DL model. It can take one or several arguments and as an input and produce one or several arguments as an output.Some layers are stateless, but more frequently layers have a state: the layer's weights, one or several tensors learned with stochastic gradient descent, which together contain the network's knowledge (Chollet, 2018).

A common view of linear function is next:

$$F(x) = w * x + b, \#(1.1)$$

where F(x) – output of the function that depends on argument x;

w–coefficient by which independent variable x is multiplied;

b – coefficient which is added toindependent variable x.

Every linear function has view as Formula (1.1). For example, if there is such equation as: $F(x) = \frac{-12*x+6}{2} + 5$ (you can see there are 2 w arguments and 2 b arguments) – it can be converted to such view: $F(x) = -6 * x + 8$.

Below you can see graph of Formula (1.1):

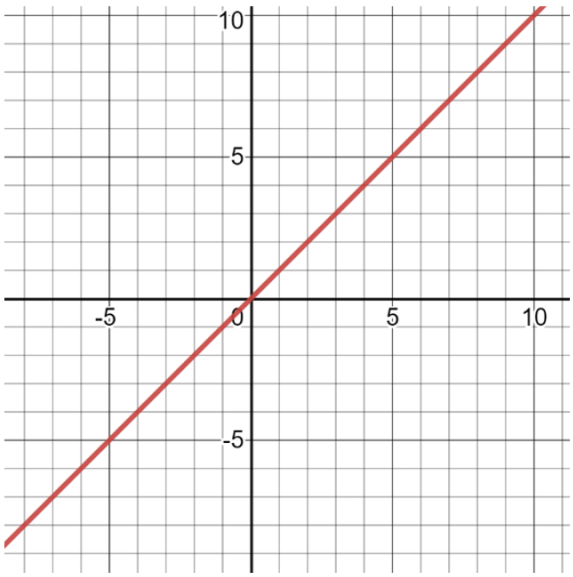

Fig. 1.1. Example of linear graph

As it is seen of Figure (1.1), every linear graph can be described as a

straightslope. w argument states the angle of the slope and b argument states how far the line is from x axes.

1.1.2. Activation function

As (Trask, 2019) states: "An activation function is a function applied to the neurons in a layer during prediction. Oversimplified, an activation function is any function that can take one number and return another number. But there are an infinite number of functions in the universe, and not all of them are useful as activation functions. There are several constraints on what makes a function an activation function. Using functions outside of these constraints is usually a bad idea, as you'll see."

Talking about constrains – several rules could be named. The first one is that activation function should be continuous and infinite in domain. There should not be a situation where you put a number in your functions but it can not produce an output for that value (Trask, 2019).

As an example, let's look at Figure (1.2) and Figure (1.3):

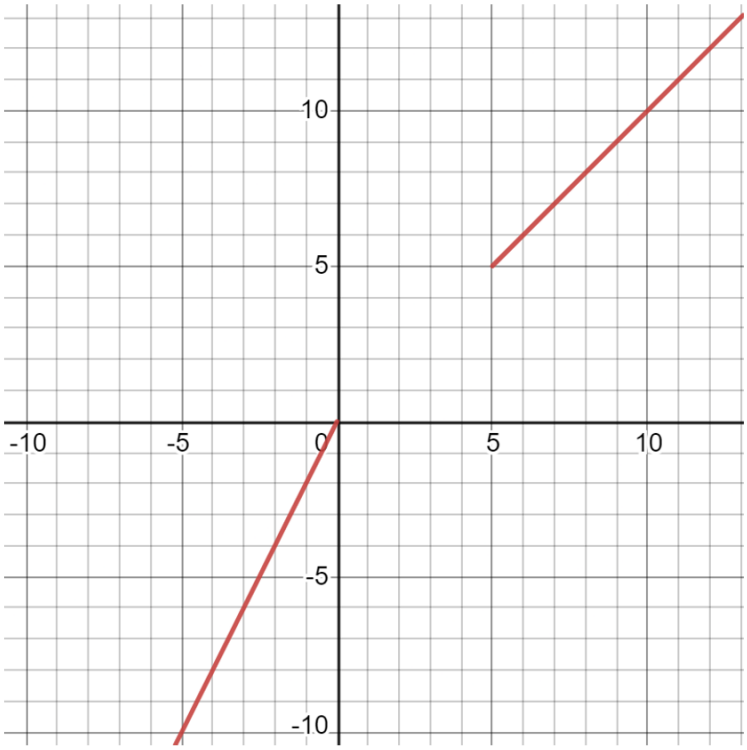

Fig. 1.2. Example of discontinuous function

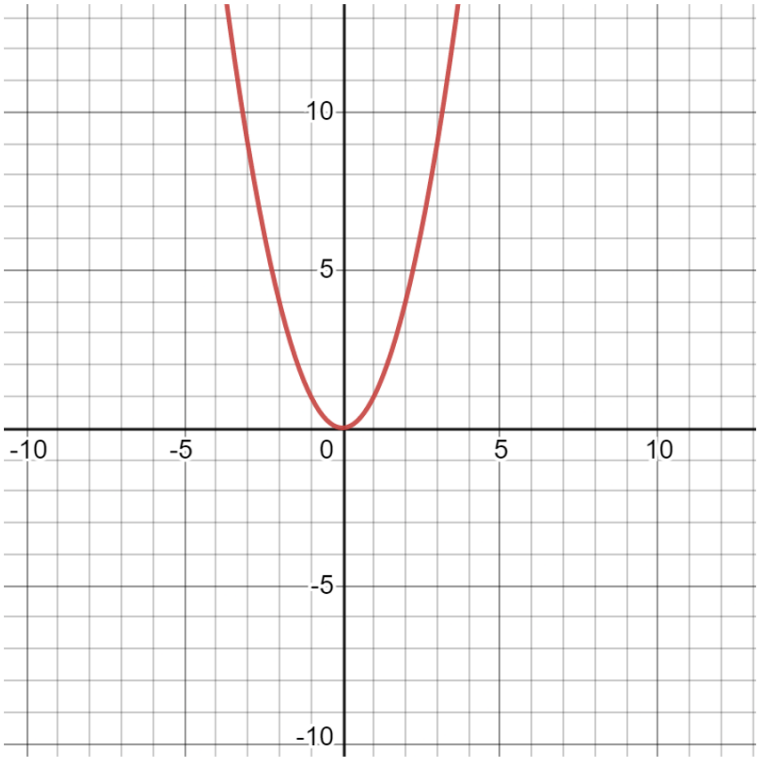

Fig. 1.3. Example of continuous function

As it is seen, on the first example, Figure (1.2) there are some values of x that has no values of y. This is a graph of an Formula (1.2):

$$f(x)\begin{cases} f(x) = x, & x > 5 \\ f(x) = 2x, & x < 0 \end{cases}, \#(1.2)$$

where $f(x)$ – the function of x value;

x – independent variable.

Discontinuous activation functions that have some 'empty' values of y would be a horrible example for a DL model (Trask, 2019).

Talking about an alternative – on the Figure (1.3), you can see function that is continuous. This is a graph of an Formula (1.3):

$$f(x) = x^2, \#(1.3)$$

Where $f(x)$ – the function of x value;

x – independent variable.

This activation function seems very nice as no 'empty' values are on it's domain and no problems would occur during learning algorithm.

2:4 Next important rule is that activation functions should be monotonic (never

changing direction) (Trask, 2019).

On Figures (1.3) and (1.4) examples of monotonic and non-monotonic functions could be seen:

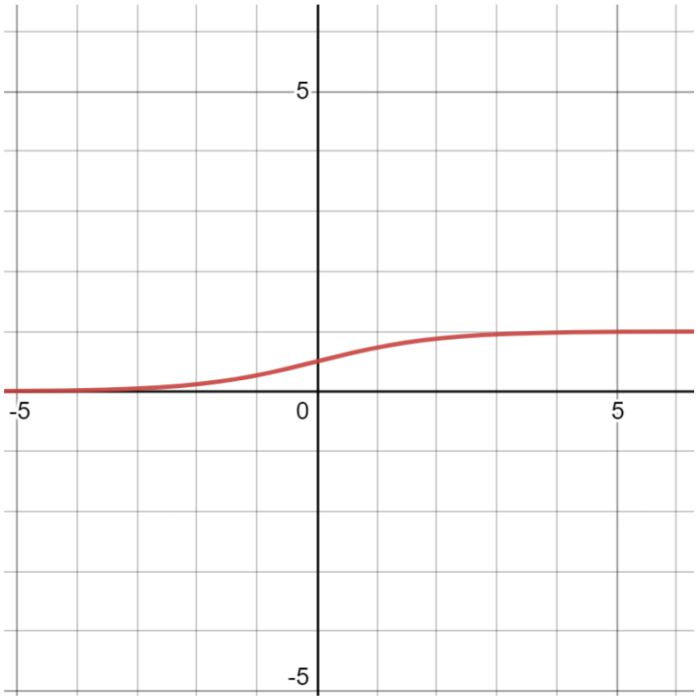

Fig. 1.4. Example of sigmoid function

Function on Figure (1.3) increases and decreases, in other words, it changes direction though it's domain. On the other side, Figure (1.4) shows sigmoid functions that is monotonic. It increases from 0 to 1 though the whole slope.

Difference between those 2 types of functions could also be described as non-monotonic function has one or several places where for 2 or more values of x there will be only 1 value of y (in Formula (1.2) – for $x = (-4, 4)$, $y = 2$). On the other hand, monotonic equation assumes that for 1 value of independent variable there can only be 1 value of dependent variable(Trask, 2019).

As (Trask, 2019) states: "This particular constraint isn't technically a requirement. Unlike functions that have missing values (noncontinuous), you can optimize functions that aren't monotonic".

Third rule is about activation function being non-linear. Explanation of this rule is described in (Trask, 2019): "In order to create it, you had to allow the neurons to selectively correlate to input neurons such that a very negative signal from one

input into a neuron could reduce how much it correlated to any input. As it turns out, this phenomenon is facilitated by any function that curves. Functions that look like straight lines, on the other hand, scale the weighted average coming in. Scaling something doesn't affect how correlated a neuron is to its various inputs. It makes the collective correlation that's represented louder or softer. But the activation doesn't allow one weight to affect how correlated the neuron is to the other weights. What you really want is selective correlation. Given a neuron with an activation function, you want one incoming signal to be able to increase or decrease how correlated the neuron is to all the other incoming signals. All curved lines do this".

Examples of such functions can be seen on Figures (1.1) and (1.4). Linear function that is based on Formula (1.1) just scales the weighted average, however, function illustrated on Figure (1.4) allows to see the correlation of the inputted signal to all other signals.

The last but not least, as we are talking about DL as n algorithm that is performed by computer and that takes huge dataset as an input, activation functions are considered to be low on computational resources. (Trask, 2019) states: "Many recent activation functions have become popular because they're so easy to compute at the expense of their expressiveness (relu is a great example of this)".

Relu equation is based on Formula (1.4) and can be seen on Figure (1.5):

$$f(x)\begin{cases} f(x) = 0, & x < 0 \\ f(x) = x, & x \geq 0 \end{cases}, \#(1.4)$$

where $f(x)$ – the function of x value;

x – independent variable.

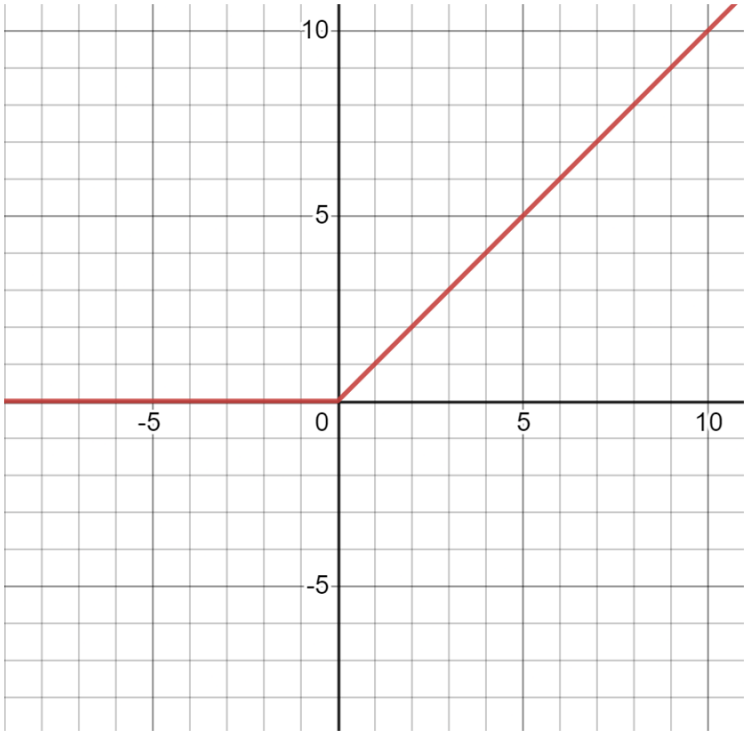

Fig. 1.5. Example of ReLU function

1.2. Normalization functions

To understand, why we need normalization – let's take an example of a model that predicts house prices. In such dataset – there will be a lot of features of a house, such as: number of floors, location, area, number of rooms, height of ceiling, etc. In such way values of those features would vary from ranges like (0 - 4) and (10000 - 1000000).

1:8

Book (Chollet, 2018) explains why it can be a problem: "In general, it isn't safe to feed into a neural network data that takes relatively large values (for example, multidigit integers, which are much larger than the initial values taken by the weights of a network) or data that is heterogeneous (for example, data where one feature is in the range 0–1 and another is in the range 100–200). Doing so can trigger large gradient updates that will prevent the network from converging. To make learning easier for your network, your data should have the following characteristics:

ϒ Take small values—Typically, most values should be in the 0–1 range.

ϒ Be homogenous—That is, all features should take values in roughly the same range."

Problem described above could be solved though applying normalization

function on your dataset. Usually Formula (1.5) is used. It makes all the values of dataset (values of features) to vary in the range from 0 to 1:

$$x_{new} = \frac{(x - x_{min})}{(x_{max} - x_{min})} \#(1.5)$$

where x_{new} – new value of the feature (after recalculation);

x – old value of the feature (before recalculation);

x_{min} – minimal value among values of particular feature;

x_{max} – maximal value among values of particular feature.

Another technic to which normalization is referred is "Batch Normalization".(Aggarwal, 2018) explains that method is such way: "Batch normalization is a recent method to address the vanishing and exploding gradient problems, which cause activation gradients in successive layers to either reduce or increase in magnitude. Another important problem in training deep networks is that of internal covariate shift. The problem is that the parameters change during training, and therefore the hidden variable activations change as well. In other words, the hidden inputs from early layers to later layers keep changing. Changing inputs from early layers to later layers causes slower convergence during training because the training data for later layers is not stable. Batch normalization is able to reduce this effect. In batch normalization, the idea is to add additional "normalization layers" between hidden layers that resist this type of behavior by creating features with somewhat similar variance".

Referring to what was preciously said – our DL model consist of different layers of linear and activation functions. Batch normalization usually applied before each linear function except the fist one. In such way, outputs from activation blocks are normalized and transferred to further linear layers.

Below you can see implementation of such model in Python:

```
class Model(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.layers = torch.nn.Sequential(
      torch.nn.Linear(in_features=10 + 2 * len(dataset_full.labels), out_features=24),
      torch.nn.ReLU(),
      BatchNormLast(in_features=24),
      torch.nn.Linear(in_features=24, out_features=16),
```

```
        torch.nn.ReLU(),
        BatchNormLast(in_features=16),
        torch.nn.Linear(in_features=16, out_features=1)
    )
```

On the example above, several layers are seen. Layers goes in such sequence: Linear – ReLU – Batch Normalization – Linear – ReLU – Batch Normalization – Linear. As it was already stated, each time values go through Linear and ReLU layer – before being transferred to next block of such layers, they are normalized with Batch Normalization layer.

More detailed explanation on how those layers work – paper (Buduma and Locascio, 2017) explains: "Normalization of image inputs helps out the training process by making it more robust to variations. Batch normalization takes this a step further by normalizing inputs to every layer in our neural network. Specifically, we modify the architecture of our network to include operations that:

1. Grab the vector of logits incoming to a layer before they pass through the nonli- nearity.

2. Normalize each component of the vector of logits across all examples of the mini- batch by subtracting the mean and dividing by the standard deviation (we keep track of the moments using an exponentially weighted moving average).

3. Given normalized inputs $\hat{x}$, use an affine transform to restore representational power with two vectors of (trainable) parameters: $\gamma\hat{x} + \beta$".

1.3. Loss functions

Choose of loss function is a crucial moment in DL model. Point of loss function is to determine how far predicted value is from the real value. More precise wording is stated in (Chollet, 2018): "Loss function (objective function)—The quantity that will be minimized during training. It represents a measure of success for the task at hand".

Imagine, you have a goal to maximize average of well-being of all humans. With poor loss function your trained AI will choose to kill all humans, as in that case their average happiness will grow, however we understand how awful this solution is. We need to keep in mind that every AI model tries to lower loss functions with any cost (Chollet, 2018).

Table 1.1(Trask, 2019)

Correspondence of loss function to particular problem

Problem type	Last-layer activation	Loss function
Binary classification	sigmoid	binary_crossentropy Formula (1.6)
Multiclass, single-label classification	softmax	categorical_crossentropy Formula (1.7)
Multiclass, multilabel classification	sigmoid	binary_crossentropy Formula (1.6)
Regression to arbitrary values	None	MSE Formula (1.8)
Regression to values between 0 and 1	sigmoid	mse Formula (1.8) or binary_crossentropy Formula (1.6)

$$loss = \frac{\sum_{i=1}^{N} -(y_i * \log(p_i) + (1 - y_i) * \log(p_i))}{N} \#(1.6)$$

where $loss$ – loss that should be calculated via formula;

N – number of predictions;

p_i – predicted probability;

y_i – real probability of the class.

$$loss = -\sum_{i=1}^{N} -(y_i * \log(p_i)) \#(1.7)$$

where $loss$ – loss that should be calculated via formula;

N – number of predictions;

p_i – predicted probability;

y_i – real probability of the class.

$$loss = \frac{\sum_{i=1}^{N} (y_i - p_i)^2}{N} \#(1.8)$$

where $loss$ – loss that should be calculated via formula;

N – number of predictions;

p_i – predicted probability;

y_i – real probability of the class.

1.4. Optimization algorithm

Previously, it was described how data is transformed through different (linear and activation) layers. Below in Formula (1.9) you can see data transformation via linear layer:

$$output = W * input + b\#(1.9)$$

where $output$ – output of linear layer;

W – weight of the input;

b – bias;

input – data that is inputted in the model.

Expression (1.9) shows you very simple prediction of $output$ based on learnable parameter input. W and b are tensor or so called, learnable parameters of the layer.

Usually, values of those parameters are assigned randomly at the beginning of the learning process and are changed during this process based on algorithm. The idea is to adjust those values through training loop (Chollet, 2018).

(Chollet, 2018) describes training loop is such sequence:

- "Draw a batch of training samples x and corresponding targets y.
- Run the network on x (a step called the forward pass) to obtain predictions y_pred.
- Compute the loss of the network on the batch, a measure of the mismatch between y_pred and y.
- Update all weights of the network in a way that slightly reduces the loss on this batch".

Iterating through this loop will make you loss dramatically decrease. Now let's talk about each step. Step 1 is just about inputting and outputting values. Step 2 is about forward passing inputted x values. In this step your information (features) that you use are going though sequence of layers of you learning model. Step 3 involves usage of loss function (described in previous section). It calculates the distance of the output of the model (predicted values) and real values. The most interesting part is step 4, on this stage we need to somehow decide in which direction

and by how far to change the weights of features used. Different examples on how to do it could be given, such as we could 'freeze' all weight except one coefficient which is under consideration. Let's imagine that with this one coefficient equals 0.5 - loss of our computation is 10.After changing this coefficient value to 0.35 – loss changes to 15, however if we change it to 0.65 – the loss will be 5. From this iteration it seems that updating our coefficient with +0.15 is the right direction to minimize loss between real value and predicted one (Chollet, 2018).

Efficiency of this experiment (Chollet, 2018) describes in such way: "This approach would be horribly inefficient, because you'd need to compute two forward passes (which are expensive) for every individual coefficient (of which there are many, usually thousands and sometimes up to millions). A much better approach is to take advantage of the fact that all operations used in the network are differentiable and compute the gradient of the loss with regard to the network's coefficients. You can then move the coefficients in the opposite direction from the gradient, thus decreasing the loss".

1.4.1. The gradient

In (Chollet, 2018) we can fund such explanation of what is a gradient: "A gradient is the derivative of a tensor operation. It's the generalization of the concept of derivatives to functions of multidimensional inputs: that is, to functions that take tensors as inputs.Consider an input vector x, a matrix W, a target y, and a loss function loss. You can use W to compute a target candidate y_pred, and compute the loss, or mismatch, between the target candidate y_pred and the target y".

As an example let's imagine we have weight W with value W0. In that case, derivative of that weight in the point W0 would be gradient:

$$gradient(f(W0))\#(1.10)$$

where $gradient$ – gradient;

f – function that is the dependable variable of a gradient;

$W0$ – dependable variable of a function, value of weight that is under consideration.

where each coefficient of that gradient would describe direction and magnitude of the change of our loss function that will occur in case we adjust different values to W0 coefficient. Gradient described in Formula (1.10) is the gradient of the function:

$$f(W) = loss\#(1.11)$$

where f – function of gradient;

W – dependable variable, value of weight;

$loss$ – value of loss function.

in the point where weight = W0 (Chollet, 2018).

From math we know that derivative of a function $f(x)$ with just a single coefficient can describe the slope of that f function. In the same way, gradient in Formula (1.10) can describe curvature of function $f(W)$ around point W0 (Chollet, 2018).

Summing up the knowledge above, we can understand how minimizing the loss in our model could be performed. Just as we can reduce value of $f(x)$ by moving x in the opposite direction from the derivative, in the same way, value of $f(W)$ could be reduced via moving the W value from the opposite of it's gradient. As an example, such equation could be considered:

$$W1 = W0 - step * Formula(1.10)\#(1.12)$$

where $W1$ – updated value of weight that is under consideration;

$W0$ – value of weight that is under consideration before updating it;

$step$ – small scaling factor that represents how far we will go in the direction to update new weight;

$Formula(1.10)$ – formula of the gradient described in Formula (1.10), derivative of the weight $W0$(Chollet, 2018).

1.4.2. Stochastic gradient descent (SGD)

Having derivation function we could find the point on its gradient where derivative of the function is equal to 0. Point where derivative of a function equals to 0 – is the point of minimal value of the function itself. In the scope of a neural network – our task is to find combination of weights using which in forward pass, lead us to the minimum loss(Zhang *et al.*, 2021). This could be done solving the equation:

$$gradient\big(f(W)\big) = 0\#(1.13)$$

where $gradient$ – gradient;

f – function that is the dependable variable of a gradient;

W – dependable variable of a function.

Such way of reducing the loss could be applied on tasks small number of variables (somewhere around 2-4), however doing it in real tasks where datasets have thousands or millions of features – it would be really insufficient. Instead, we could use a four-step algorithm that was preciously described. As we are dealing with differentiable function – computation of its gradient could be performed to optimize step number 4 (updating the weight in the opposite direction of its derivative will lower the loss every iteration)(Chollet, 2018):

- Draw a batch of training samples x and corresponding targets y.
- Run the network on x to obtain predictions y_pred.
- Compute the loss of the network on the batch, a measure of the mismatch between y_pred and y.
- Compute the gradient of the loss with regard to the network's parameters (a backward pass).
- Move the parameters a little in the opposite direction from the gradient—for example W -= step * gradient—thus reducing the loss on the batch a bit.

Described algorithm above is an example of a SGD algorithm. Below you can see graphical representation of it.

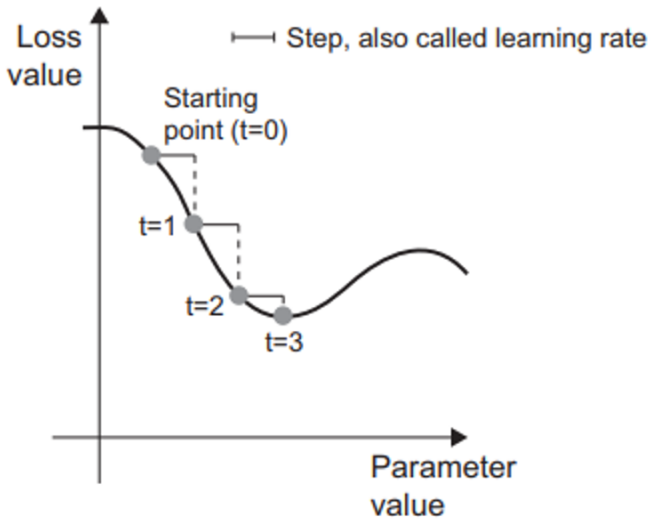

Fig. 1.6. Graphical representation of SGD algorithm(Chollet, 2018).

It was already described, why we need step variable in Formula (1.12), however a note about this variable should be added. If too small value would be

picked – model could be stuck in local minimum (Figure 1.7) or it would spend too much iterations and learning process will last for too long time. If large value would be chosen –updated weights could be found in completely random positions of the curve and end up not minimizing the loss, but even increasing (Chollet, 2018).

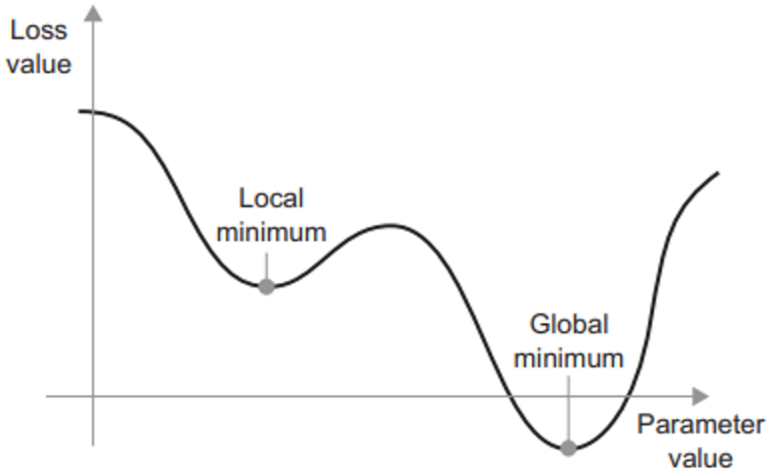

Fig. 1.7. Representation of local and global minimum (Chollet, 2018).

As it is seen on Figure (1.7), loss function has several points where its value could be described as minimum. Local minimum represents point of the curve where loss decreases, but not getting to the minimal value of the whole curve. With small step value, model could decide that this local minimum is the only minimum in our function and get stuck in it, however, we want our model to get to the real minimum, named global minimum.

Such problem could be solved via using some technics, (Chollet, 2018) describes how it could be performed: "You can avoid such issues by using momentum, which draws inspiration from physics. A useful mental image here is to think of the optimization process as a small ball rolling down the loss curve. If it has enough momentum, the ball won't get stuck in a ravine and will end up at the global minimum. Momentum is implemented by moving the ball at each step based not only on the current slope value (current acceleration) but also on the current velocity (resulting from past acceleration). In practice, this means updating the parameter W based not only on the current gradient value but also on the previous parameter update, such as in this naive implementation".

Here you can see a small algorithm in programming language that realizes that

method (adopted from (Chollet, 2018)):

```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
  w, loss, gradient = get_current_parameters()
  velocity = past_velocity * momentum + learning_rate * gradient
  w = w + momentum * velocity - learning_rate * gradient
  past_velocity = velocity
  update_parameter(w)
```

1.5. Backpropagation algorithm

With the knowledge background of optimization algorithm, we can proceed further to backpropagation algorithm. This technic is based on widely known chain rule of derivation. The situation is that, in neural network, there are a lot of layers, and they are connected with each other in a certain sequence. To compute gradient of a whole model –chain rule is applied(Chollet, 2018; Vasilev *et al.*, 2019).

Tern of backpropagation is referred to the backward phase that is used in every multi-layer neural network. There are 2 phases(Aggarwal, 2018):

- Forward phase: we input values of the features of a dataset that are going to be used in computation of the output. After values are inputted – they go through defined layers (like linear or activation) and used for calculations with respect to their weight. We try to minimize loss between outputted value and the real value that we have. Next step would be to calculate derivative of the loss function with respect to previously used weight.
- Backward phase: idea of this phase is to calculate gradient of loss function with respect to weights of features to reduce this loss function. Calculation of those weights is performed via applying chain rule of differential calculus. Term back appears because, using chain rule we start calculation from the bottom layers of the model and sequentially listing up to the top layers.

Let's consider an example (Chollet, 2018):

$$f1(W1, W2, W3) = f2(W1, f3(W2, f4(W3)))\#(1.14)$$

where $f1$ – main function of the model that includes all the layers of the model (3

layers in our case);

$f2$ – inner function of the model, that takes $W1$ and $f3$as inputs (it is the top layer);

$f3$ – inner function of the model, that takes $W2$and $f4$as inputs (it is an output for $f2$);

$f4$ – inner function of the model, that takes $W3$ inputs (it is an output for $f3$);

$W1, W2, W3$ – weights of the features from dataset.

Using Function (1.14), we would need to apply chain rule in order to calculate its derivative. We would start from deriving $f4$, then $f3$ and up with computing $f2$. A more formulaic form of this calculation is represented in Function (1.15)(Petersen and Pedersen, 2007):

$$f(g(x)) = f'\big(g(x)\big) * g'(x) \#(1.15)$$

where f – main function that takes $(g(x))$ as an input;

g – inner function that takes x as an input;

x – argument of g function.

Today's usage of backpropagation algorithm is thoroughly described in (Chollet, 2018): "Nowadays, and for years to come, people will implement networks in modern frameworks that are capable of symbolic differentiation, such as TensorFlow. This means that, given a chain of operations with a known derivative, they can compute a gradient function for the chain (by applying the chain rule) that maps network parameter values to gradient values. When you have access to such a function, the backward pass is reduced to a call to this gradient function. Thanks to symbolic differentiation, you'll never have to implement the Backpropagation algorithm by hand. For this reason, we won't waste your time and your focus on deriving the exact formulation of the Backpropagation algorithm in these pages. All you need is a good understanding of how gradient-based optimization works".

1.6. R-Squared

R-squared is a type of metric that is used in popular DL model to determine variation of dependent variables by the independent variables of the dataset. Book (Chicco, Warrens and Jurman, 2021) gives such explanation: "The coefficient of determination (Wright, 1921) can be interpreted as the proportion of the variance in the dependent variable that is predictable from the independent variables".

R-squared is based on Formula (1.16):

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(X_i - Y_i)^2}{\sum_{i=1}^{N}(\bar{Y} - Y_i)^2} \#(1.16)$$

where R^2 – R-squared;

N – number of samples;

X – independent variable of a dataset;

Y_i – dependent variable of a dataset (true value of the dataset);

$\bar{Y}$– mean of real values.

As it was already described, by the output of the Formula (1.16), we can understand how teachable dataset is. Values of output could vary from $-\infty$ to +1, where $-\infty$ is the worst possible value, while 1 is the best (Chicco, Warrens and Jurman, 2021).

To understand how that metric could be used, let's consider an example where we have 2 regression models with output that scales from 0 to 10 in one case and from 0 to 100 in another. With such metrics as MSE or MAE, it would be impossible to compare those 2 models, however, using R-squared we would get coefficient in the range $(-\infty, 1]$, which will show us the predictive performance of those datasets (Chicco, Warrens and Jurman, 2021).

1.7. Regression tasks

Now, with theoretical background of deep learning models, we can dive into regression tasks. There are 2 main types of DL problems(Chollet, 2018; Russell and Norvig, 2021):

- Classification tasks: idea of classification task is that you model outputs probability for every class of your dataset. As an example, image classification can be given. Pixels are taken as feature and type of image as an output. You input set of images in required format and model predicts what kind of classes are shown on the image (model can predict either dog, cat or elephant is located on the image). In such tasks, output vary in the range 0 – 1, where 0 means that that exact class in not found or determined on the image and 1 means that probability of that class is 100%.
- Regression tasks: in such tasks, model also, teaches based on the

features inputted and real values of the dataset, however, output values can vary in any range. As an example, prediction of house prices could be named. You input dataset to your model with features of houses, such as: number of floors, area of the house, distance between house and center of the city. Those features can, also, have not scalar values, but categorical, such as: location or type of house. In such case those categories will be transformed into scalar values so they can be used by the model (examples of such values will be given in the practical section). After all house knowledge is inputted, model starts to learn and after assigning faithful values to the weight of the features, it starts to can predict the price.

1.7.1. Inputs – Outputs

First stage of building the model–implementation of dataset based on which learning would be performed. Usually, values of dataset can vary in a huge range and it can slow down the learning. Solution of this problem is described in (Chollet, 2018): "It would be problematic to feed into a neural network values that all take wildly different ranges. The network might be able to automatically adapt to such heterogeneous data, but it would definitely make learning more difficult. A widespread best practice to deal with such data is to do feature-wise normalization: for each feature in the input data (a column in the input data matrix), you subtract the mean of the feature and divide by the standard deviation, so that the feature is centered around 0 and has a unit standard deviation".

Book (Chollet, 2018), also, gives example of how such normalization could be performed:

```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std
test_data -= mean
test_data /= std
```

1.7.2. Example of architectures

After successful implementation of the dataset, we can proceed to building the

logical unit of the model. As it was previously mentioned, model consist of different layers of functions, such as linear or activation (sigmoid, ReLU, Tanh, etc…). Depending on size of dataset, complexity of the model and many other circumstances, different layers and their sequence will be built.

Usually, last layer of a regression model is a linear layer. With that, model can output values in a various range, which is a crucial factor of a regression neural network. For example, if sigmoid function would be chosen, for the last layer, output could only vary in a range from 1 to 0 (which is acceptable for classification model, but not for regression)(Chollet, 2018).

Example of such architecture is given in scientific paper (Chollet, 2018). This is an example of Python code using opensource library for deep learning models "keras":

```
from keras import models
from keras import layers
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
                           input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

1.7.3. Loss functions – MSE, MAE, Huber Loss

In the previous sub-section you can find background knowledge of loss functions. It was already said, that loss function allows you to determine distance between real values and predicted ones. Here briefly will be described 3 main types of such loss function that are used in regression models.

Mean average error (MAE) is one of the most common and simple functions to calculate loss of the model. MAE is based on Formula (1.17). First step is to find distance between one particular output and real value connected with features of this output. Then all of those distances are summed up and divided by the number of outputs(Chai and Draxler, 2014):

$$loss = \frac{\sum_{i=1}^{N} |y_i - \widehat{y_i}|}{N} \#(1.17)$$

where $loss$ – value that represents loss that calculates via formula;

N – number of observations;

y_i – real value;

$\widehat{y_i}$ – output of the model;

i – number of iteration.

Another common function that is used to determine loss, is the mean square error (MSE). First step is the same as in MAE, we calculate distance, however, after that, result of first calculation is squared. After that program must sum all of those squares and divide by the number of observations. Formula (1.18) represents that equation(Chai and Draxler, 2014):

$$loss = \frac{\sum_{i=1}^{N} (y_i - \widehat{y_i})^2}{N} \#(1.18)$$

where $loss$ – value that represents loss that calculates via formula;

N – number of observations;

y_i – real value;

$\widehat{y_i}$ – output of the model;

i – number of iteration.

Last example of a loss function in this section is Huber loss. This function is considered as a balance between MSE and MAE. Calculation of loss could be performed using Formula (1.19)(Meyer, 2021):

$$loss = \begin{cases} \frac{1}{2} * (y_i - \widehat{y_i}) & , for (y_i - \widehat{y_i}) \leq delta \\ delta * (y_i - \widehat{y_i}) - \frac{delta^2}{2}, & otherwise \end{cases} \#(1.19)$$

where $loss$ – value that represents loss that calculates via formula;

$delta$ – parameter defined by user;

y_i – real value;

$\widehat{y_i}$ – output of the model.

1.8. Importance of normalization functions

2. METHODOLOGY

2.1. Datasets

In the related work several datasets will be implemented to compare usage of normalization functions under different circumstances. In this section such dataset are described.

2.1.1. Dataset - Weather in Szeged

The first dataset is “Weather in Szeged”. It consists of 96543 samples of data with 12 features. Each row has next features:

- Formatted Date: In this column date and time of weather observation is recorded. Data is stored in scalar type. Example: “2006-04-01 00:00:00.000 +0200”.
- Summary: Overall information about forecast. Data is stored in text format. Example: “Partly Cloudy”.
- Precip type: Type of precipitation during particular time. Data is stored in textformat. Example: “rain”. Figure (2.1).
- Temperature: This column represents the temperature in Celsius. Data is stored in scalar type. Example: “9.47222222222222”. Figure (2.2).
- Apparent Temperature: This feature represents apparent temperature of particular time. Data is stored in scalar type. Example: “7.38888888888888”. Figure (2.3).
- Humidity: Here humidity of the air is observed. Data is stored in scalar type. Example: “0.89”. Figure (2.4).
- Wind Speed: Information about speed of the wind is stored (in kilometers per hour). Data is stored in scalar type. Example: “14.1197”. Figure (2.5).
- Wind Bearing: Wind bearing is observed in this column (in degrees). Data stored in scalar type. Example: “251”. Figure (2.6).
- Visibility: This feature represents visibility in kilometers. Data is stored in scalar type. Example: “15.8263”. Figure (2.7).

This dataset is based on weather observations during period from 2006 – 2016 in Szeged.

Below you can see histograms for most of the features to understand the statistics of this dataset:

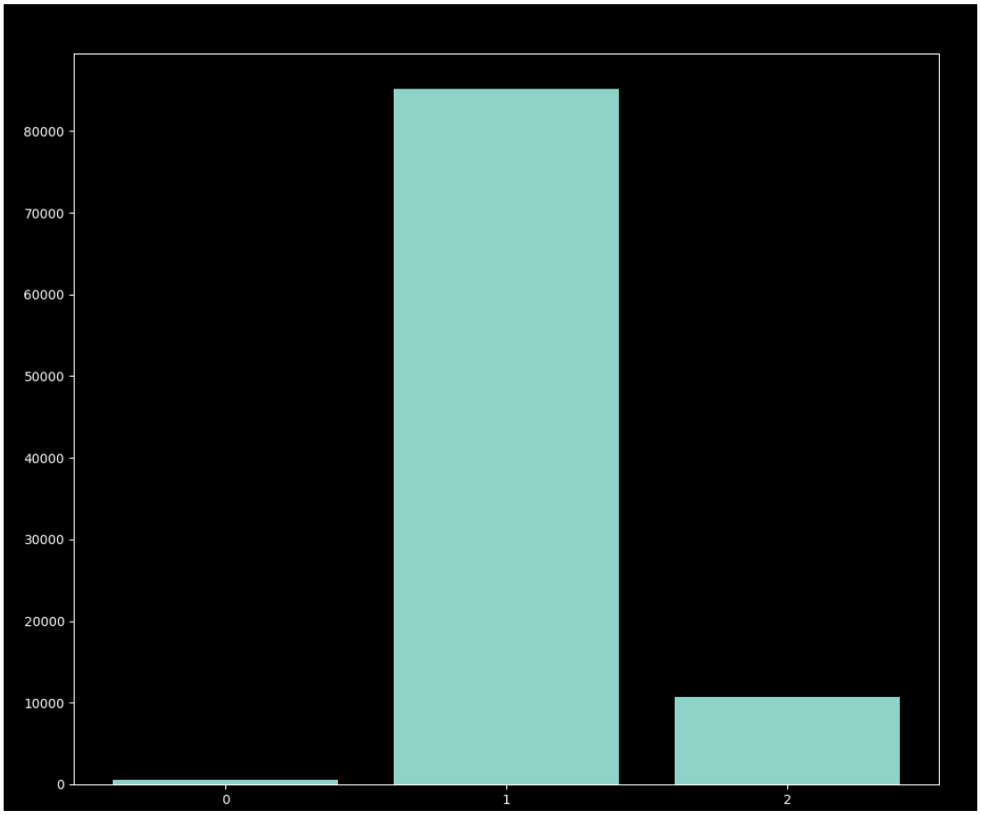

Fig. 2.1. Histogram forfeature “Precip type”

On Figure (2.1), you can see histogram for the feature “Precip type” which shown how many number of different types of precipitation were stored in current dataset. Each number on x axes represents each type of precipitation:

- 0 – No precipitation was recorded,
- 1 – Rain,
- 2 – Snow.

It is clearly seen that rain is the dominant precipitation among dataset. To be exact, there are 85224 times rain was recorded, 10712 – snow and 517 times no precipitation was found (all together 96543 records).

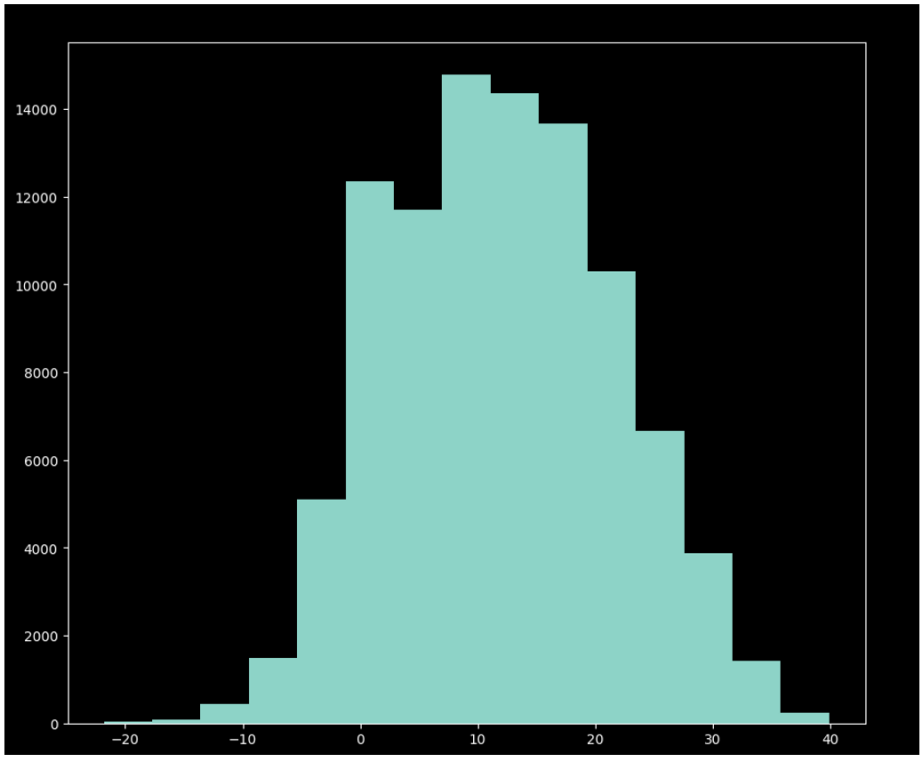

Fig. 2.2. Histogram for feature “Temperature”

Figure (2.2) allows us to notice normal or Gaussian distribution of the temperature samples. X axes represents temperature of a row. We can see that most of values of that feature are concentrated in the middle of a range. The lowest temperature recorded is -21.8 C, while maximum is 39.9 C.

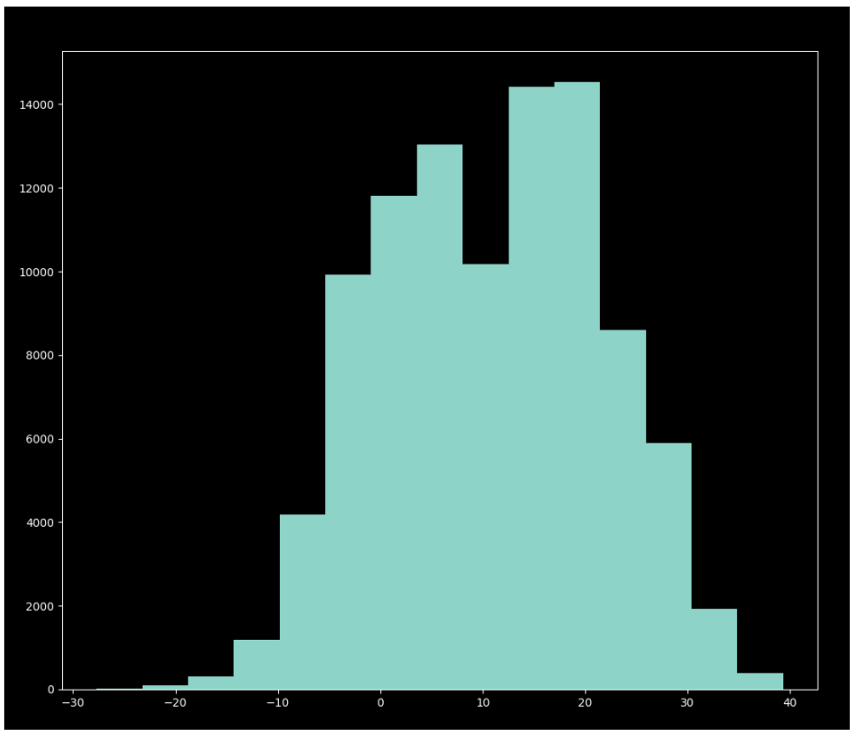

Fig. 2.3. Histogram for feature “Apparent temperature”

This histogram (Figure 2.3), as previous, shows us normal distribution of the values of current feature. Most of the values are located in the range from -5 C to 25 C. Minimal apparent temperature recoded is -27.7 C, while maximum is 39.3 C.

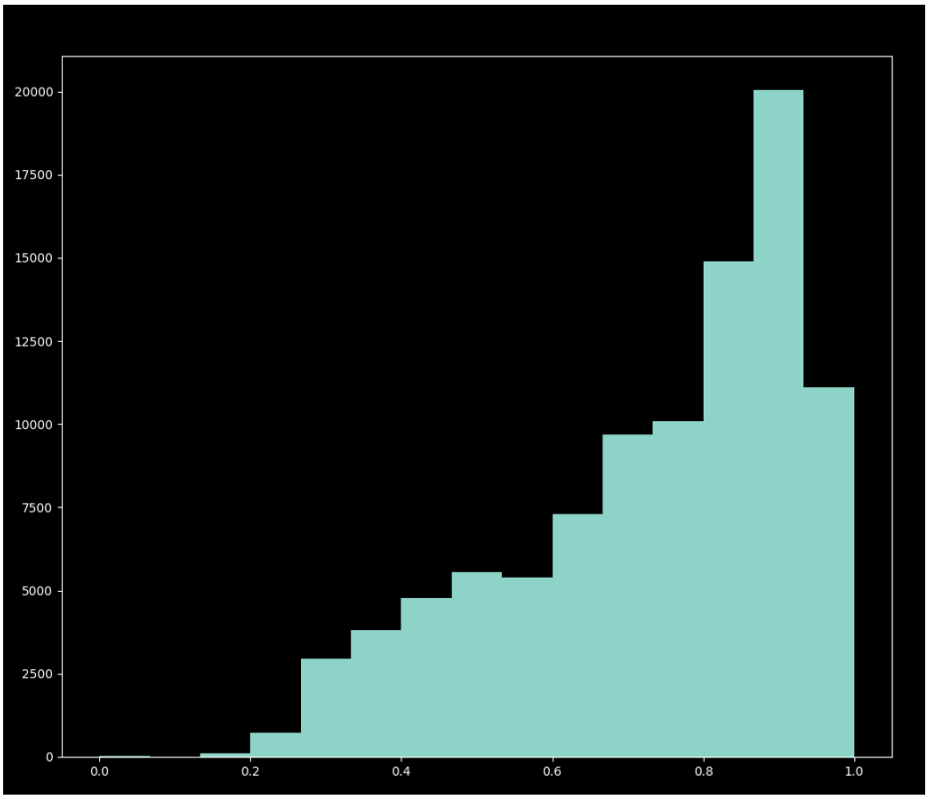

Fig. 2.4. Histogram for feature “Humidity”

On Figure (2,4) histogram for feature “Humidity” is located. Unlike Figure (2.2) and Figure (2.3) no Gaussian distribution is seen. As values grow their number of occurrences increases. Most of occurrences are seen at the highest numbers of humidity. In such sequence, most of the samples are located in the range from 0.7 to 1. Minimal value of humidity recorded is 0 and the maxim is 1.

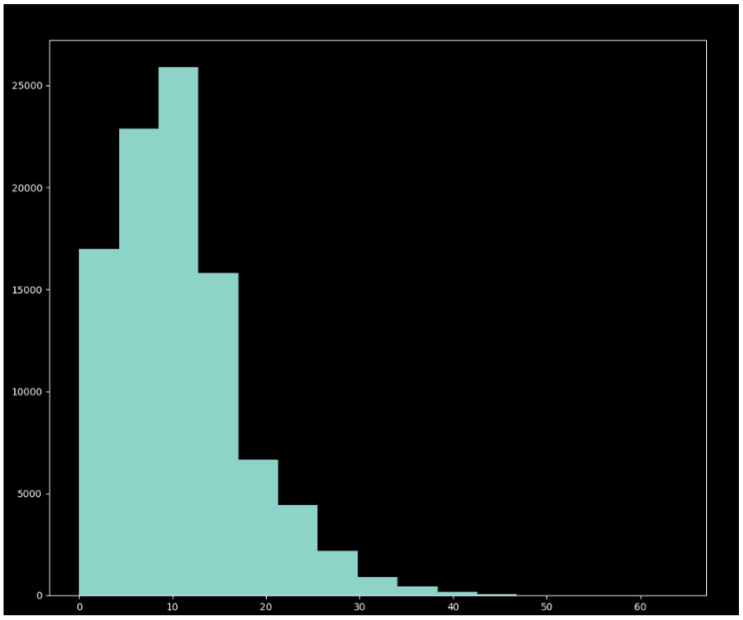

Fig. 2.5. Histogram for feature “Wind Speed”

Looking on Figure (2.5), it is clearly seen situation that is opposite to what is shown on Figure (2.4). As values of feature decreases – number of their occurrences increases. This kind of distribution is called exponential distribution. With that logic, most of the values of wind speed are distributed across the range from 0 km/h to 15km/h. Minimal wind speed that was stored in the dataset is 0 km/h, while maximum is 63.9 km/h.

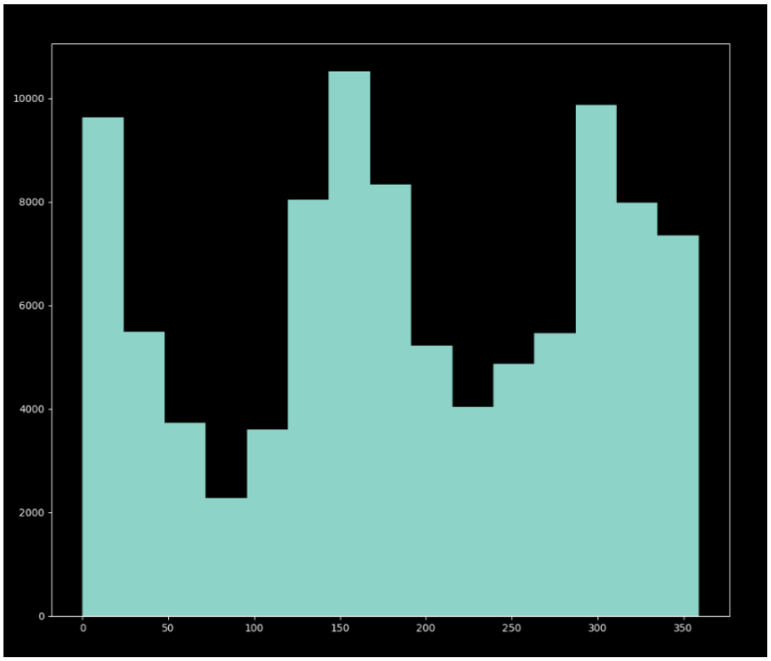

Fig. 2.6. Histogram for feature “Wind Bearing"

Looking at Figure (2.6) we can observe that most of the features are located on the extrema and at the middle part of the histogram. Biggest angle of wind bearing recorded is 0 degree, while maximum is 359 degree. This distribution is logical as we are talking about degrees, which are applied to a circle. It could, also, be described as: most of the samples are located in the places where sinus of their degree would be equal to 0 and vice versa, as sinus of their degree approaches 1 – number of occurrences decrease.

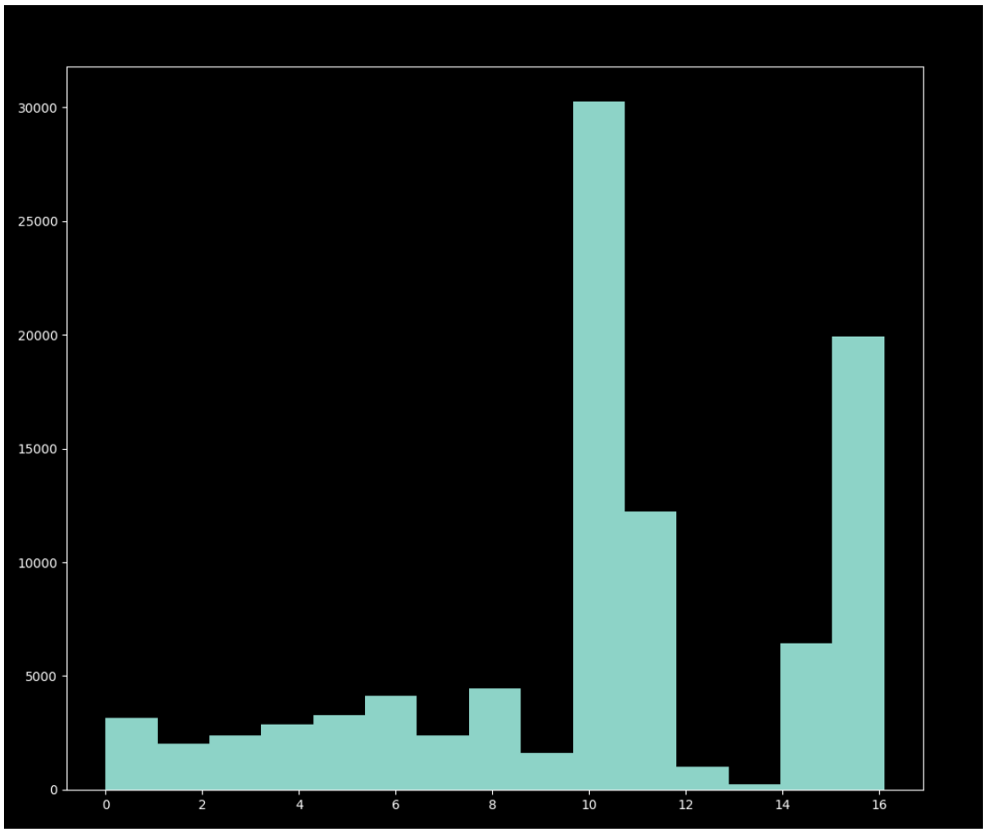

Fig. 2.7. Histogram for feature “Visibility"

Figure (2.7) allows us to understand normal visibility in the region when data was recorded. Most of the time, visibility is approximately 10 kilometers. Minimal value that was recorded is 0 km, while maximum is 16.1 km.

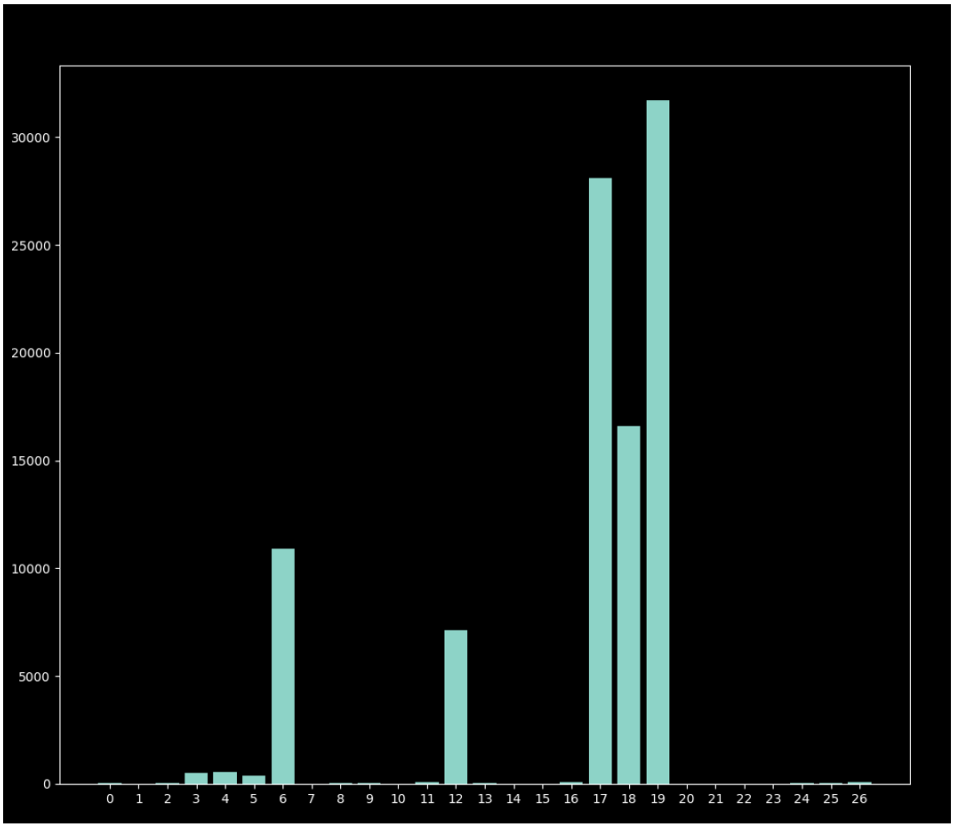

Fig. 2.8. Histogram for feature "Summary"

As on Figure (2.1), Figure (2.8) shows us distribution of text format data. That kind of data usage was described in previous section. To use text data in learning process – we need to represent it in a way of scalar values. For each value there is one concrete class:

- 0 –Breezy,
- 1 – Breezy and dry,
- 2 – Breezy and Foggy,
- 3 – Breezy and Mostly Cloudy,
- 4 – Breezy and Overcast,
- 5 – Breezy and Partly Cloudy,
- 6 – Clear,
- 7 – Dangerously Windy and Partly Cloudy,
- 8 – Drizzle,
- 9 – Dry,
- 10 – Dry and Mostly Cloudy,
- 11 – Dry and Partly Cloudy,

- 12 – Foggy,
- 13 – Humid and Mostly Cloudy,
- 14 – Humid and Overcast,
- 15 –Humid and Partly Cloudy,
- 16 – Light Rain,
- 17 – Mostly Cloudy,
- 18 – Overcast,
- 19 – Partly Cloudy,
- 20 – Rain,
- 21 – Windy,
- 22 – Windy and Dry,
- 23 – Windy and Foggy,
- 24 – Windy and Mostly Cloudy,
- 25 - Windy and Overcast,
- 26 - Windy and Partly Cloudy.

It is seen that mostly, dataset is filled with values of: “Clear”, “Foggy”, “Mostly Cloudy”, “Overcast”, “Partly Cloudy”.

Plotting of histograms was performed using python programming language with opensource library for plotting “matplotlib”. Below, you can see code which was written for plotting:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
X = np.array(pd.read_csv("./weatherHistory.csv"))
for scalar_feature in range(3, 6):
    self.X = X[:, scalar_feature:9]
    plt.hist(self.X[:, 5], bins=15)
    plt.show()
for text_feature in range(1, 3):
    self.Y = np.array((X[:, text_feature])).astype(np.str_)
    unique, counts = np.unique(self.Y, return_counts=True)
    y_hist = dict(zip(unique, counts))
    names = list(range(0, len(y_hist)))
    values = list(y_hist.values())
```

```
plt.bar(range(len(y_hist)), values, tick_label=names)
plt.show()
```

2.1.2. Datasets –CalCOFI

CalCOFI is a dataset of a 60 years of oceanographic observation by California Cooperative Fisheries Investigations. It consists of 864864 samples with 74 rows of features each.There is data of:

- temperature,
- salinity,
- oxygen,
- phosphate,
- silicate,
- nitrate and nitrite,
- chlorophyll,
- transmissometer
- PAR,
- C14 primary productivity,
- phytoplankton biodiversity,
- zooplankton biomass,
- zooplankton biodiversity.

Even though, having such big data, not all of the rows are filled fully, however, this problem can be solved during the stage of implementation of the dataset.

Description of such big data would take too much space; only statistics of most usable and important features are listed below:

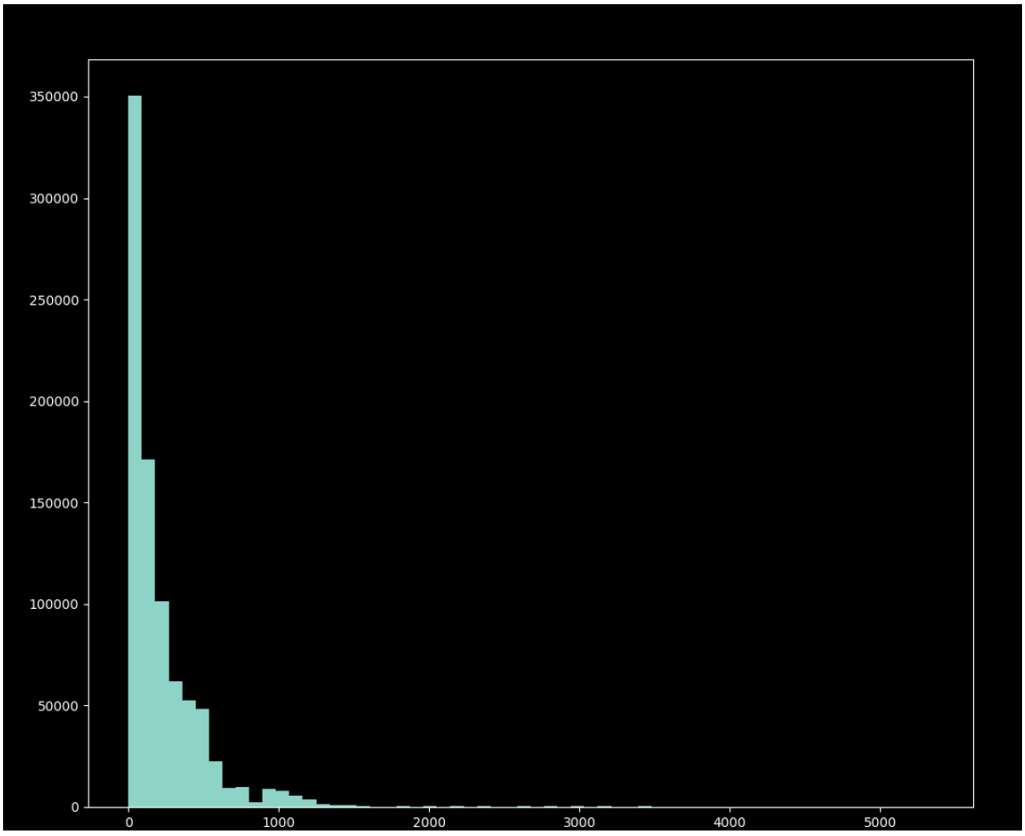

Fig. 2.9. Histogram for feature “Depthm”

Histogram located on Figure (2.9) shows us distribution of values of feature “Depthm”. This feature represents depth (in meters) that was recorded. All of those values are of scalar type. We can see that most of the occurrences are of a depth not more than 400 meters. As value of feature increases – number of samples – decreases.

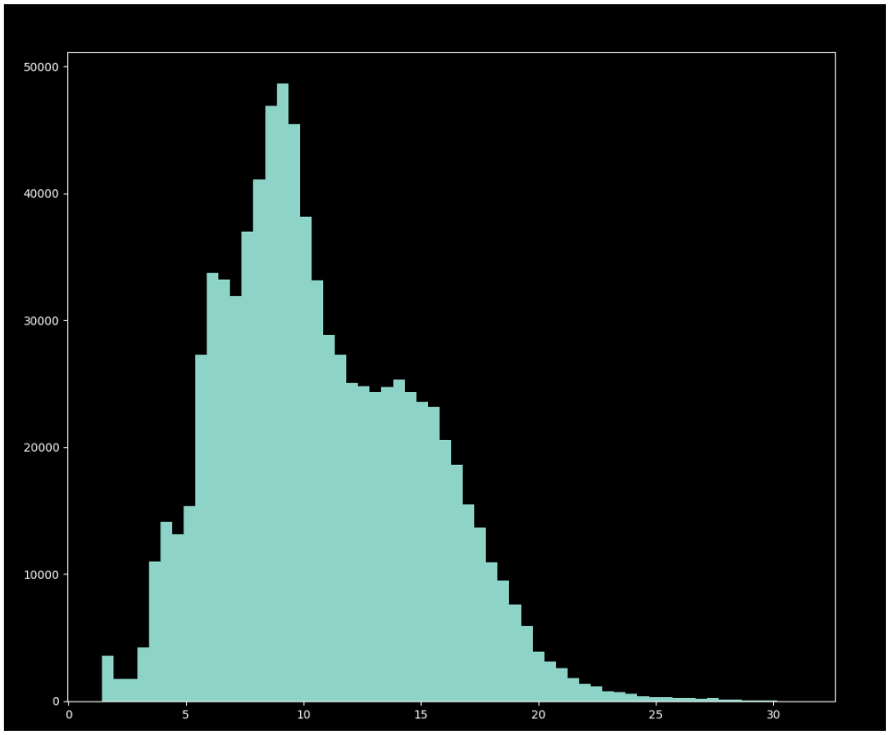

Fig. 2.10. Histogram for feature "T_degC"

Figure (2.10) shown values of feature "T_degC". Most of the values are located in the range from 3 to 17.

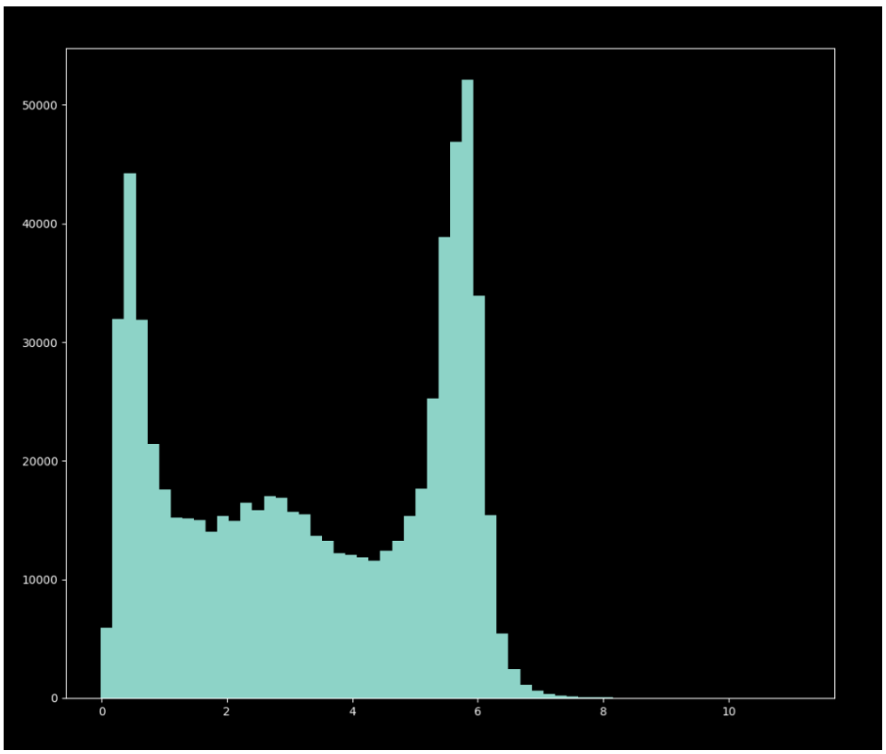

Fig. 2.11. Histogram for feature "STheta"

Figure (2.11) shown values of feature “STheta”. We can see 2 point of largest cluster of samples around values 0.4 and 5.5. Most of other values are located between those 2 clusters.

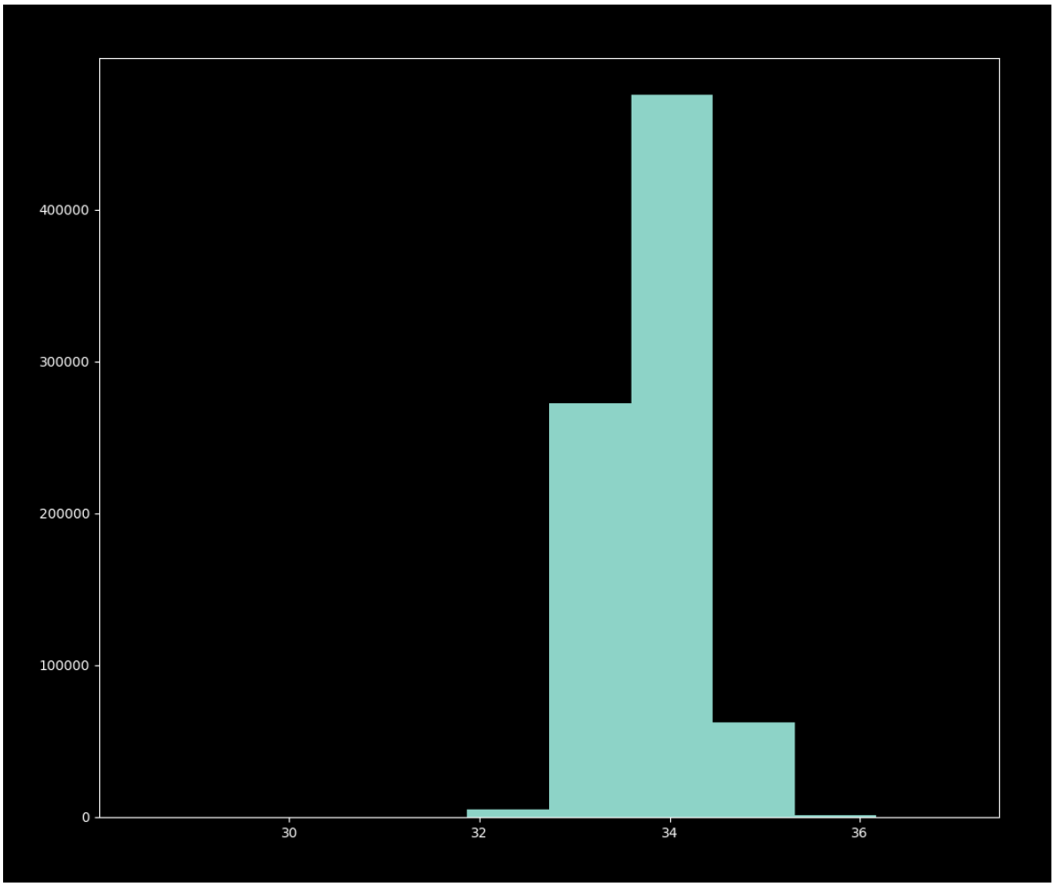

Fig. 2.12. Histogram for feature “Salnty”

On Figure (2.12) values of feature “Salnty” are shown. It is seen that most of values are equal to 34. Other don’t deviate from it very much. 99% of them are in the range from 33 to 35.

2.2. Normalization functions

2.3. Architecture

2.4. Metrics

2.4.1. R-Squared

R-squared is a type of metric that is used in popular DL model to determine variation of dependent variables by the independent variables of the dataset. Book (Chicco, Warrens and Jurman, 2021) gives such explanation: “The coefficient of determination (Wright, 1921) can be interpreted as the proportion of the variance in the dependent variable that is predictable from the independent variables”.

R-squared is based on Formula (2.1):

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(X_i - Y_i)^2}{\sum_{i=1}^{N}(\bar{Y} - Y_i)^2} \#(2.1)$$

where R^2 – R-squared;

N – number of samples;

X – independent variable of a dataset;

Y_i – dependent variable of a dataset (true value of the dataset);

$\bar{Y}$– mean of real values.

As it was already described, by the output of the Formula (2.1), we can understand how teachable dataset is. Values of output could vary from $-\infty$ to +1, where $-\infty$ is the worst possible value, while 1 is the best (Chicco, Warrens and Jurman, 2021).

To understand how that metric could be used, let’s consider an example where we have 2 regression models with output that scales from 0 to 10 in one case and from 0 to 100 in another. With such metrics as MSE or MAE, it would be impossible to compare those 2 models, however, using R-squared we would get coefficient in the range $(-\infty, 1]$, which will show us the predictive performance of those datasets (Chicco, Warrens and Jurman, 2021).

3. REFERENCES

Aggarwal, C.C. (2018) *Neural networks and deep learning: a textbook*. Cham, Switzerland: Springer. doi:10.1007/978-3-319-94463-0.

Buduma, N. and Locascio, N. (2017) *Fundamentals of deep learning: designing next-generation machine intelligence algorithms*. First edition. Sebastopol, CA: O'Reilly Media.

Chai, T. and Draxler, R.R. (2014) 'Root mean square error (RMSE) or mean absolute error (MAE)? – Arguments against avoiding RMSE in the literature', *Geoscientific Model Development*, 7(3), pp. 1247–1250. doi:10.5194/gmd-7-1247-2014.

Chicco, D., Warrens, M.J. and Jurman, G. (2021) 'The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation', *PeerJ. Computer Science*, 7, p. e623. doi:10.7717/peerj-cs.623.

Chollet, F. (2018) *Deep learning with Python*. Shelter Island, New York: Manning Publications Co.

Meyer, G.P. (2021) 'An Alternative Probabilistic Interpretation of the Huber Loss', in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Nashville, TN, USA: IEEE, pp. 5257–5265. doi:10.1109/CVPR46437.2021.00522.

Petersen, K.B. and Pedersen, M.S. (2007) 'The Matrix Cookbook'.

Russell, S.J. and Norvig, P. (2021) *Artificial intelligence: a modern approach*. Fourth edition. Hoboken: Pearson (Pearson series in artificial intelligence).

Trask, A.W. (2019) *Grokking deep learning*. Shelter Island: Manning.

Vasilev, I. *et al.* (2019) *Python deep learning: exploring deep learning techniques and neural network architectures with PyTorch, Keras, and TensorFlow*. Second edition. Birmingham Mumbai: Packt Publishing Limited.

Zhang, A. *et al.* (2021) 'Dive into Deep Learning', *arXiv preprint arXiv:2106.11342* [Preprint].