Safety assertions in neural network classification

https://github.com/ericbill21/siemens

Eric Tillmann Bill*
RWTH Aachen University
Aachen, Germany
eric.bill@rwth-aachen.de

Robert Maximilian Giesler*
RWTH Aachen University
Aachen, Germany
robert.giesler@rwth-aachen.de

Felix Maximilian Knispel*
RWTH Aachen University
Aachen, Germany
felix.knispel@rwth-aachen.de

***Abstract*—Dependability and reliability of machine learning classification systems are getting increasingly important as machine learning techniques are being applied in more and more domains, including safety-critical areas. The task of the *Siemens AI Dependability Assessment Student's Challenge* is to provide a classifier for a given classification tasks as well as to determine an as-accurate-as-possible misclassification probability estimate.**

In this paper, we describe a machine learning model that has been optimized for the safest classification possible. A multilayer feed-forward neural network was trained on the three given datasets. We introduce a custom loss function to reduce the number of safety-critical misclassifications. Additionally, we detect predictions that are less certain than a determined threshold value and decide to resort to a safer classification in these instances.

Moreover, we describe the approach taken to determine a probability estimate for misclassifications: As training data quality is crucial to model performance and dependability, we develop several interpretations of training data and use these as indicators to estimate model accuracy.

For the three given datasets, our approach provides reliable (though not guaranteed) misclassification rates no larger than 6.89%, 2.34%, and 2.62%, respectively.

I. Introduction

A. Overview

Siemens Mobility's *AI Dependability Assessment Student's Challenge* consists of two main tasks:

1) developing a machine learning algorithm for a simple binary two-dimensional classification problem and, more importantly,
2) providing justifiable safety assertions for the decisions made by the algorithm and ideally presenting a reliable upper bound for the misclassification probability.

The motivation for this challenge stems from the controversy surrounding the deployment of machine learning algorithms in areas such as autonomous driving or unmanned aerial vehicles, where decisions critical to human safety must be made. The calculations made by machine learning algorithms are usually highly complex and unintuitive to human reasoning, which often leads to hesitance when applying machine learning solutions in safety-critical fields. It is important that we can provide justifiable safety assertions for the decisions made by machine learning algorithms and that we can sufficiently limit the probability of misclassification before implementing them in high-risk applications.

*Equal contribution

Potential safety-critical applications for machine learning algorithms which come to mind are often highly complicated and require the processing of multi-dimensional data. Although the complexity of such problems goes far beyond that of our two-dimensional classification problem, it is still meaningful to view the tasks at hand in this very basic context. For one, we cannot hope to solve these tasks for high-dimensional, safety-critical real-world examples if we cannot solve them for simple problems such as the one given here. Furthermore, solutions found here may be scaled up to prove effective for more complicated applications.

B. The Challenge

The challenge is defined by the following game:

"Player A (the Assessor) chooses a subset S of the unit square $I = [0, 1]^2$ and a probability distribution P on I.

Then A chooses a sample size n and generates n random variates x_i from P. Additionally labels I_i are assigned: $I_i = 1$ (red), if x_i is an element of S, and 0 (green), else – thus, the Assessor uses an indicator function I_S.

Player E (the Safety Expert) now has the task to provide a machine learning algorithm and safety arguments for this classification problem, including all assumptions and, possibly, safety-related application rules. In particular, she must provide a (non-trivial) upper bound for the probability, that the next random variate x_{n+1} is misclassified.

If a red data point is labeled green, then this decision is safety-critical, and an accident may happen. If a green data point is labeled red, then this may cause or some cost, but not directly a safety problem. Take traffic lights as an example…"

We as the challenge participants take the role of Player E while Siemens Mobility takes the role of Player A.

C. The Datasets

Siemens Mobility provides three datasets (A, B, and C) of varying sample size n, probability distribution P, and complexity of subset S, which each represent an independent instance of the proposed game. Going forward, we will refer to the attributes n, P, and S of a specific dataset X as n_X, P_X, and S_X. The datasets are provided as xls files containing the coordinates and the color labels (0 or 1) of all points, as can

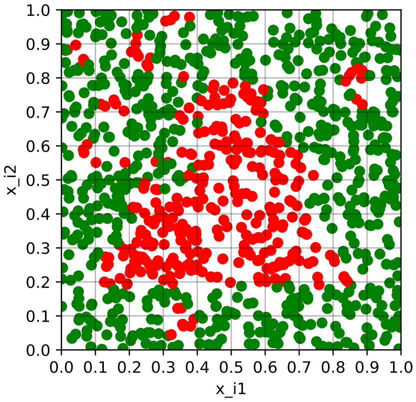

Fig. 1: Dataset A.

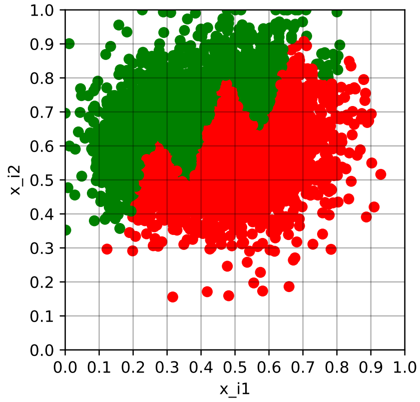

Fig. 2: Dataset B.

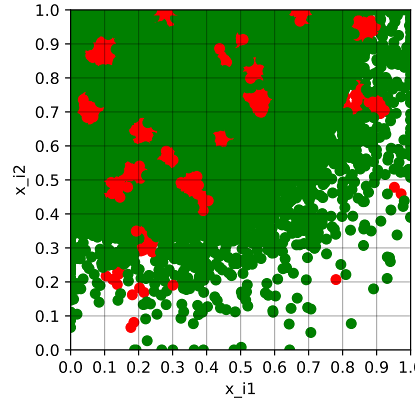

Fig. 3: Dataset C.

	x_i1	x_i2	l_i
x_1	0.78074177284725	0.412336851935834	0
x_2	0.623970211716369	0.94046398694627	0
x_3	0.89157349569723	0.435561570571736	0
x_4	0.590269535314292	0.58635878469795	1
...	...	...	...
x_997	0.917966741370037	0.700236862991005	0
x_998	0.416427534539253	0.444144908105955	1
x_999	0.475235156947747	0.267783672083169	1
x_1000	0.808225766057149	0.687669077888131	0

TABLE I: Extract of dataset A.

be seen for the example of dataset A in Table I. As the points of each dataset were randomly generated from a probability distribution P on the unit square $I = [0,1]^2$, the coordinates of the given points take real number values on a continuous scale between 0 and 1. There is a significant discrepancy between the sizes of the datasets: dataset A consists of 1000 points, dataset B of 5000 points, and dataset C of 50,000 points.

A substantial benefit of working with two-dimensional data is that it can be easily visualized and that it is easy to grasp and interpret for human readers, which is not usually the case when working with higher dimensional data. Visualizations of the three given datasets are found in Figure 1, Figure 2, Figure 3.

D. *Paper structure*

For the most part, the structure of this paper chronologically follows the line of approach we took in tackling this challenge. Section II focuses on the architecture of the machine learning model we designed, which acts as the base of our solution. We begin with an overview of the structure of our model and the process of finding suitable hyperparameters. A series of methods to specifically reduce the misclassification of red points are introduced later in section II.

In section III-C, we concentrate on the task of providing an estimate for the misclassification probability of our machine learning algorithm. We address the question of whether it is possible to prove that misclassification rates can be limited, and we explain the process of reaching reliable misclassification estimates on all datasets. Section III-C closes with suggestions on how statistical analysis of misclassification patterns may be used to further reduce the misclassification upper bounds in practice.

In section IV, we explain how our methods and approaches may be scaled to higher dimensions in order to accommodate complex real-world applications. We conclude in section V.

II. Classification Model

In this section, we go into detail about the machine learning algorithm we designed for the classification of the points in the given datasets. We especially focus on the methods applied to reduce the misclassification probability of red points, which, according to the challenge description, is substantially more critical that the misclassification probability of green points.

Motivated by the widespread success of artificial neural networks (ANNs) in classification tasks over the past decade (e.g. [1]–[3]) and previous experience of our team with feedforward ANNs, we decided to employ a feedforward artificial neural network as the classifier for this challenge.

We used GitHub and Google Colaboratory to share and jointly develop the code for our machine learning solution. All code for data analysis, data operations, visualizations, and the neural network itself was written in Python 3.7.10 in a Jupyter Notebook. The TensorFlow and Keras libraries were used for the implementation of the ANN. The complete code for our solution can be found in appendix **??**. Due to the limited computational power of the personal computers available to us, we additionally used Google Colaboratory Pro's and Datalore Pro's cloud computing services for many of our computationally intensive calculations.

A. *Model Architecture*

When constructing an artificial neural network for a classification task, several factors must be taken into consideration. The depth and width (number of layers and number of neurons per layer) of the network must be chosen such that the classification function separating the different classes of the given dataset may be adequately estimated by the network.

Generally speaking, increasing the depth and the width of a neural network is a straightforward way of increasing its classification performance. Larger networks, however, tend to face a higher risk of overfitting, especially when working with limited amounts of labeled training data [2]. This was especially important to keep in mind, as we were developing an ANN tasked with delivering promising classification results on three different datasets with vastly dissimilar amounts of labeled training points.

The neural network we designed for the proposed classification task consists of an input layer, four densely connected hidden layers, and a densely connected output layer. The number of neurons per layer, in order of input layer to output layer, are as follows:

$$2, 100, 70, 50, 10, 2$$

We use ReLU as the nonlinear activation function in all hidden layers. ReLU neurons are non-saturating and therefore substantially increase training speeds compared to saturating nonlinearities such as the tanh function [1], [4]. Our network receives the x_{i1} and x_{i2} coordinates of a datapoint as input and produces two output values. As is common practice in neural network classifiers, a softmax function is applied to the output to ensure that the output values sum up to 1. The two output values of the network represent the network's calculated probability that the input point is red, and the probability that the input point is green, respectively, as values between 0 and 1.

We found that our neural network architecture delivers impressive classification accuracy on all three datasets without running into noticeable issues with overfitting.

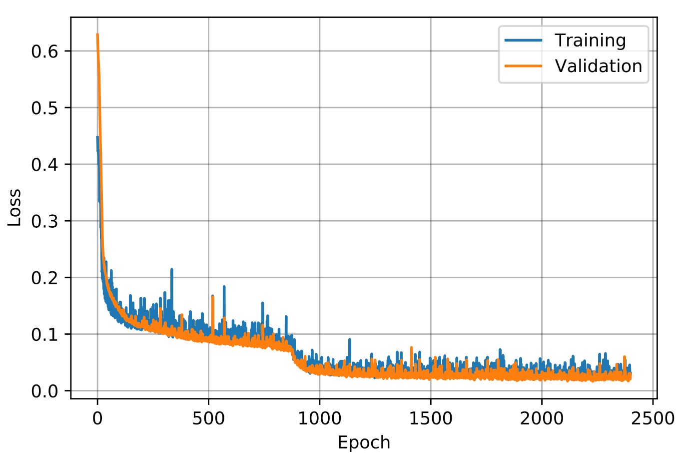

Fig. 4: Training and validation loss during training on dataset C.

Since no validation sets are provided for this challenge, we randomly extract 15-16% of the training points from each dataset before training (15% for A, 16% for B and C). Thereby we can train our model using 84-85% of the provided data and validate its accuracy using the remaining 15-16% of unseen points. Figure 4 shows the validation and training loss of an example training (fitting) run of our network on dataset C with respect to training epoch. It is evident that both training and validation loss follow a downward trend as the training process advances. Overfitting, which would be indicated by an upward trend in validation loss at some stage during training, does clearly not occur. Analogous plots for example training runs on datasets A and B can be found in appendix D.

It is important to note that one trained instance of the network cannot possibly be used to accurately classify all three datasets. Instead, three instances of the model must be created and trained on one of the datasets each, leaving us with three separate classifiers.

Given that the main task of the challenge is not to provide a neural network architecture which is as accurate or efficient as possible, we spent comparatively little time optimizing and fine tuning the architecture of our model. Rigorously testing and optimizing both the performance and the efficiency of the neural network architecture may therefore further improve the results presented here, but lies beyond the scope of this project.

B. Training-related hyperparameters

After having defined the architecture of our neural network, we will now look at the training-related hyperparameters used for fitting our model.

Instead of using a classical stochastic gradient descent approach, we decided to use the Adaptive Moment Estimation (Adam) optimizer [5], which combines the two stochastic gradient descent algorithms of Adaptive Gradients (AdaGrad) and Root Mean Square Propagation (RMSProp). Due to its computational efficiency and often favorable performance compared to other stochastic optimization methods, Adam is currently recommended as the default stochastic gradient descent algorithm [4]. We found that using Adam's default parameters of $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ delivers good results when training our network on each of the three datasets.

The batch size (number of training samples propagated through in one forward pass) and the number of epochs (number of times the whole training set is shown to the network) have a strong influence on the classification performance of the network. To reduce the misclassification probability of our model as far as possible, we have to determine the optimal combination of batch size and number of epochs for training on each dataset. The optimum values of these parameters will vary from dataset to dataset due to the differences in the number of training samples and the complexity of the subsets S_A, S_B, and S_C.

We implemented a function named *averageEpochsBatchSize* to calculate the optimum batch size and number of epochs for each dataset. For a given dataset, *averageEpochsBatchSize* receives a list of possible batch sizes B, a list of possible epoch numbers E, and a number of iterations $n \in \mathbb{N}$. For every iteration $0 \leq i < n$, we extract a random balanced validation set from the given dataset and train our model once for each possible combination of batch size and epoch number $\{(b, e) \mid b \in B, e \in E\}$ using the remaining training points. Before every training run, the weights are reset to an initial value defined by the Glorot initialization algorithm. We then let our model classify the validation points to calculate the percentage of red points, the percentage of green points, and the total percentage of points from the validation set misclassified. After n iterations of extracting a random balanced validation set, training the model using all possible batch size and epoch number combinations, and classifying the validation points, the average red, green, and total misclassification percentages for every combination (b, e) are calculated and plotted in a three-dimensional space. Figure 18 shows the percentage of red points misclassified with respect to batch size and epoch number on dataset B, averaged over 50 training and validation runs for each (b, e) combination.

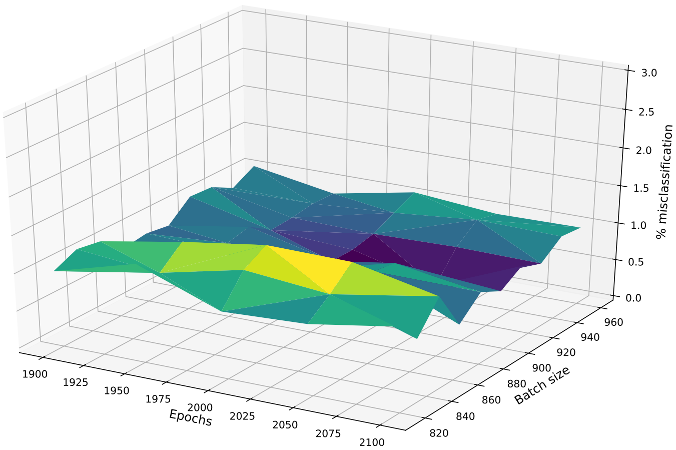

Fig. 5: Average red % misclassification with respect to batch size and epoch number on dataset B.
Parameters: $n = 50$, $B = \{816, \ldots, 960\}$ in increments of 16, $E = \{1900, \ldots, 2100\}$ in increments of 50.

Note that for this optimization procedure, we chose *balanced* validation sets, meaning that the points for the validation sets were randomly selected under the condition that each color must make up at least 40% of the validation points. In doing so, we avoid the risk of one color being underrepresented by chance, which could have resulted in a bias in our optimum batch sizes and epoch numbers.

Given the information that the misclassification of red points is severely more critical than the misclassification of green points, the optimum combination of batch size and epoch number is one which results in minimum average red percentage misclassification while maintaining acceptable green percentage misclassification.

Following this method, we conclude the following optimum batch sizes and epoch numbers for training our model on the three given datasets:

	batch size	**epoch number**
Dataset A	16	1450
Dataset B	912	2050
Dataset C	2000	2400

Due to our limited computational resources, we had to run *averageEpochBatchSize* multiple times for each dataset, slowly narrowing down the optimal points by first using large ranges of batch sizes and epoch numbers with large increments and later using smaller ranges with smaller increments.

Plots analogous to Figure 18 showing average total, red, and green percentage misclassification with respect to batch size and epoch number for all datasets can be found in appendix A.

C. Dataset balancing

Since the most critical part of machine learning is the learning part itself, we need to optimize training data in cases where safety-critical classes are greatly underrepresented. In the given datasets, the relations between green and red points are as follows:

Dataset A	32.60% red	67.40% green
Dataset B	52.52% red	47.48% green
Dataset C	9.59% red	90.41% green

A balanced distribution of both classes, as present in dataset B, is desirable as both classes are equally important - therefore, the network should see both classes appropriately often during training. In dataset C, for example, the safety-critical class red is heavily underrepresented. To account for such discrepancies in red or green samples, we duplicate data points of the less frequent color until both colors make up a minimum proportion of the training set defined by a threshold value $t_{Bal} \in [0, 0.5]$. During testing, we found that some threshold values lead to increased misclassification rates. Especially $t_{Bal} = 0.5$ led to a significant decrease in accuracy. As the goal of the balancing threshold is to increase accuracy, we only considered threshold values which reduce misclassification of the underrepresented class.

For each value $t_{Bal} \in \{0,\ 0.1,\ 0.2,\ 0.3,\ 0.4\}$, 50 different validation sets are extracted from dataset C. All validation sets are randomly chosen and consist of 40-60% green points. The remainder of dataset C is then balanced according to the threshold value t_{Bal} and used to train the model. Figure 6 shows the averaged misclassification on these different validation sets.

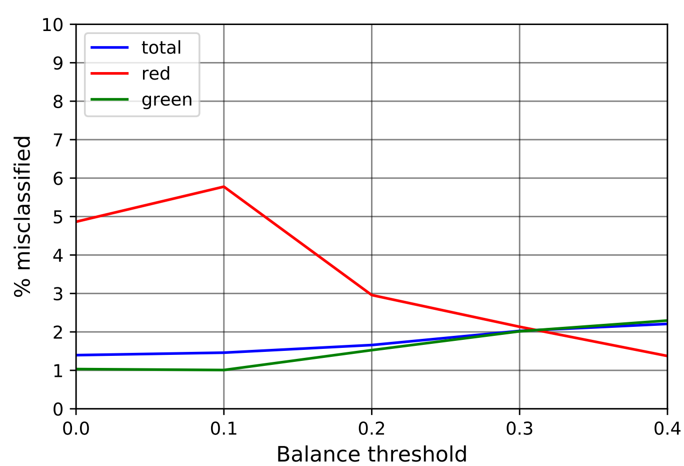

Fig. 6: Balance threshold effect on dataset C.
Parameters: $n = 50$, $t_{pen} = 0.2$, epochs $= 1800$, batch size $= 912$.

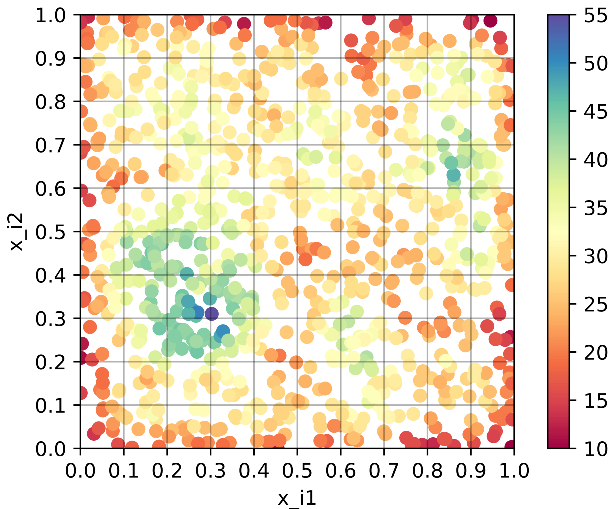

Fig. 7: Density plot showing the points of dataset A color-coded according to the number of other points in a radius of 0.1.

We found that for dataset C the optimal threshold value is 0.3, as red misclassification drops by 2.73% while green misclassification is increased by only 1%.

Finding such a threshold value for dataset A turned out to be more difficult: We observed that balancing the dataset with any threshold would result in either no difference at all or in a significant drop in accuracy. We assume that the small size of the dataset causes this effect. To understand this effect, we analyse dataset A further: For each data point, we calculate the number of "neighboring data points" in a radius of 0.1. We find that, on average, every point has 29.68 other points in its proximity and that the majority of points are equally spread over the unit square I (as can be seen in Figure 7). Therefore, duplicating only one point in a square of the grid in Figure 7, where red and green are equally represented, will almost certainly have a significant impact on the model's behavior. Considering these observations, we decided not to balance dataset A before training.

We conclude that balancing a given dataset can lead to an improvement in the training of the model. However, it should only be applied if the dataset is large enough and a safety-critical class is heavily underrepresented. Even then, extensive testing is necessary in order to avoid major distortions and to find the optimum threshold value t_{Bal}.

D. Penalty effect

Scenarios in which some mistakes are profoundly more impactful than others can be found in our everyday lives: Misclassifying a red traffic light as green can result in a serious traffic accident, while incorrectly classifying a green light as red usually only leads to furious honking of other drivers.

The setting of this challenge is quite similar. A misclassification of a point as red is assumed to be worse than a green misclassification. In order to achieve this effect, we introduced the *custom_penalty_loss* function which reduces the impact of green misclassifications, consequently making red misclassification seem more critical. This function receives the prediction output of the model as well as the ground truth values of the training samples. The model's output for a data point is represented by a tuple (p_G, p_R), where p_G is the probability that the point is green and p_R is the probability that it is red. Due to the use of the softmax function, $p_G + p_R = 1$ always holds.

If the model misclassifies a green point, the *custom_penalty_loss* function adds a constant penalty value to the p_G value and subtracts it from the p_R value. In cases where $p_G + Penalty > 1$ and $p_R - Penalty < 0$ applies, the loss function sets the tuple to $(1, 0)$. The result of this manipulation is the artificial reduction of the loss when misclassifying a green point. The custom loss function then uses an underlying standard loss function to calculate and return the loss of the manipulated predictions. Throughout all of our calculations, we used Sparse Categorical Crossentropy as the underlying loss function.

We call the impact of this adjustment the *Penalty effect*.

1) Approach to identify the optimal penalty: Picking a penalty value consists mainly of two criteria: 1) how many green misclassifications are we willing to accept and 2) how low do we want the red misclassification rate to be.

A penalty value of 0 results in no penalty effect at all, while a value of 1 results in all green misclassifications being fully ignored. A model which uses a penalty value of 1 rarely classifies any data points as green - making 1 obviously a very bad choice for a penalty value. To determine a better penalty value, we train the given model on different penalty values and use misclassification rates on a unseen validation sets as a measure to obtain an optimal penalty value.

We start by selecting n different validations sets, each 15-16% the size of the dataset. All sets are randomly chosen and consist of 40-60% green points. For each penalty value we remove the validation set from the dataset, initialize the model weights, and train the model using the remaining training points. This process is repeated for each of the n validation sets. The total, red, and green misclassification values are then averaged for each penalty value.

Such a penalty value analysis for dataset B can be found in Figure 8, using 20 different penalty values and $n = 50$. As hypothesized earlier, penalty values near 1 lead to near-zero red misclassification at the cost of skyrocketing green misclassification. The constant decline in red misclassification rates as the penalty value increases suggests a direct relation between penalty value and the number of red misclassifications. This behavior confirms that the implementation of a custom penalty loss function leads to the wanted results.

Interestingly, green misclassification rates show a similar linear relation on the interval $[0, 0.5]$. Once we increase the penalty value above 0.5, however, we see a rapid increase in green misclassification of at least 4.7%. We are not willing to accept such an increase in green misclassification in return for very little further decreases of red misclassifications. However, we find that these misclassification rates do not necessarily follow a particular pattern, making cost estimation difficult

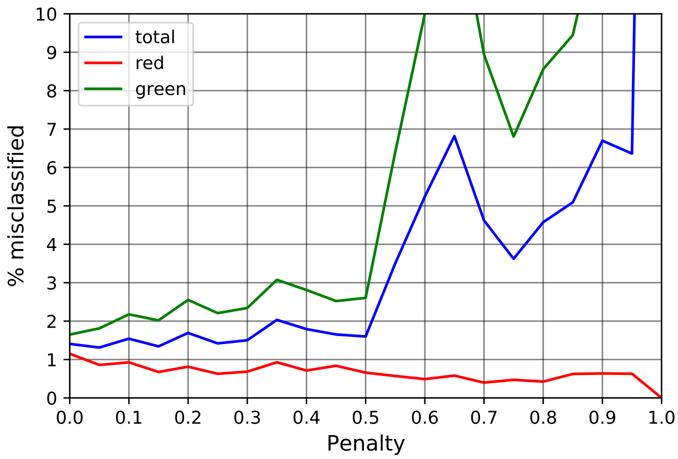

Fig. 8: Penalty effect on dataset B.
Parameters: $n = 50$, epochs $= 1800$, batch-size $= 912$.

without extensive testing. Based on our testing, we recommend a penalty value between 0 and 0.5.

It is to be noted that a similar penalty effect was observed on all three datasets, further underlining the effectiveness of the custom penalty loss function.

After performing extensive testing on each dataset, we agreed on the following penalty values:

Dataset A	t_{pen} = 0.4
Dataset B	t_{pen} = 0.25
Dataset C	t_{pen} = 0.2

2) Visualization: To understand how our model classifies data, we can visualize its predictions: By generating predictions for a grid of 1000×1000 data points, we obtain a predicted color as well as the model's certainty in its prediction. Those results are then color-coded and plotted, as can be seen in Figure 9 for the color red. In this specific figure, the model was trained without the penalty function. In comparison to the exact same model trained with a penalty value of $t_{pen} = 0.25$, we can easily see how the area of greater uncertainties is greatly reduced due to the penalty effect (Figure 10). Moreover, we can observe that areas where we have fewer data points (like in the top right corner) the model is highly certain of these data points in this area being red. In contrast, the model without the penalty effect considered this area as green - on might hypothesize that this is a consequence of the ratio of points in this corner: 4 green points and 2 red points can be found in this area in dataset B. This green majority might lead to the classification of this area as green. However, 6 data points hardly provide reliable information about this area. As red misclassifications are to be avoided, this penalty-effect-induced behavior is beneficial to the safety-critical nature of our task.

E. Custom Training Approaches

To further optimize the penalty effect, we conducted two different experiments during the training stage:

- Increasing-penalty training
- Decreasing-penalty training

The *increasing-penalty training* method starts training the model with a penalty value of 0 on the first epochs and then sequentially increases the penalty value until the final, previously-determined penalty value is reached. Over the last epochs, we train the model with this final penalty value. *Decreasing-penalty training* changes the penalty effect in the opposite direction during training.

For example, in Figure 12 on dataset B, we increased the penalty value every 50 epochs until we reach 1300 epochs. From there on we train with the full penalty of 0.25 for the last 500 epochs, resulting in 1800 epochs in total. Furthermore, we tested the rate of misclassification on 50 different validation sets and averaged them, similar to the penalty effect approach. The Figure 12 clearly indicates that *increasing-penalty training* as well as *decreasing-penalty training* perform worse. Hence, we will not further investigate these custom approaches.

F. Threshold effect

As we've seen previously, our model is highly certain of its predictions in most areas. Along the "border" between red and green areas however, predictions are less confident - as can be seen in Figure 11. Since guessing that a data point is red is safer in such uncertain scenarios, we introduce a threshold function that acts as a simple linear threshold neuron at the end of the model for predictions only. Therefore, the model will only predict a data point as green if the model's certainty in its decision is equal to or above the certainty threshold. Otherwise, it swaps the predictions.

After extensive testing, similar to the findings of the penalty values, we agreed on the following certainty threshold values for each dataset:

Dataset A	t_{cert} = 0.9
Dataset B	t_{cert} = 0.9
Dataset C	t_{cert} = 0.95

Even though certainty values below 0.9 seem to be far from "purely guessing" a class, they are not sufficient enough to be classified as green. This is due to the overall high certainty of the model on each dataset, as seen in the case of dataset B where there are only minor differences between Figure 10 and Figure 11.

Analysis of optimal certainty threshold values on dataset C can be found in Figure 13: increasing t_{cert} values leads to slowly rising green misclassification rates.

However, $t_{cert} = 1$ of course leads to catastrophic green classification rates. Consequently selecting a value of 0.95 for dataset C is optimal, since red misclassification rates further minimized by $\approx 1,73$, while the costs for deteriorating green classification performance are deemed justifiable.

III. Misclassification upper bound

In this section, we describe how we assess the safety and dependability of our model's predictions.

We've implemented methods such as the penalty effect and the certainty threshold which reduce the number of red points

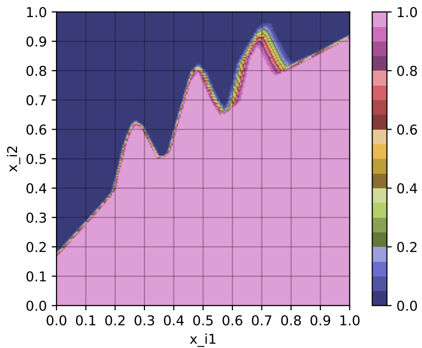

Fig. 9: Certainty for red with $t_{pen} = 0$ and $t_{cert} = 0$.

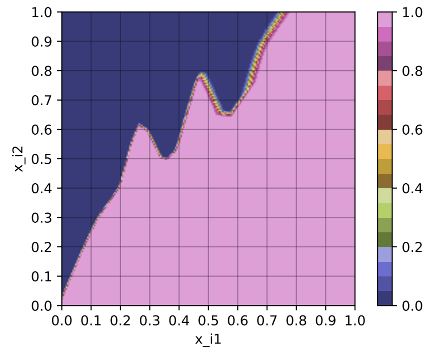

Fig. 10: Certainty for red with $t_{pen} = 0.25$ and $t_{cert} = 0$.

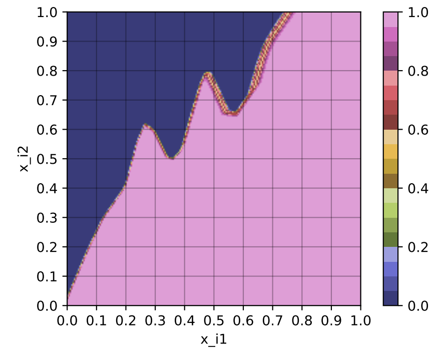

Fig. 11: Certainty for red with $t_{pen} = 0.25$ and $t_{cert} = 0.9$.

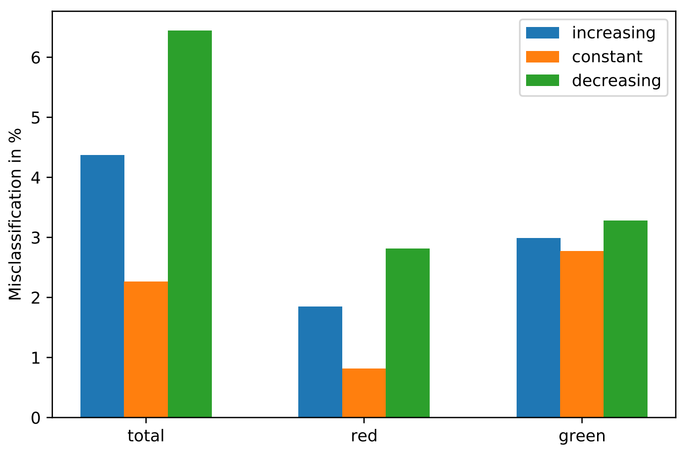

Fig. 12: Average misclassification with different training approaches on dataset B.
Parameters: $n = 50$, $t_{pen} = 0.25$, epochs $= 1800$, batch size $= 912$.

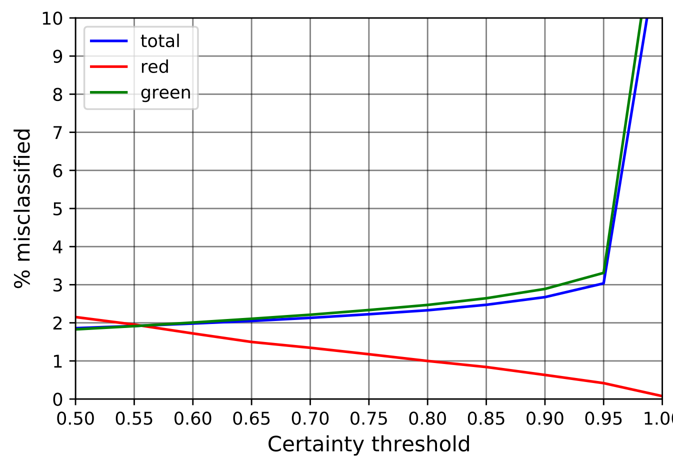

Fig. 13: Optimal threshold-prediction value for dataset C
Parameters: $n = 50$, $t_{bal} = 0.3$, $t_{pen} = 0.2$, epochs $= 2000$, batch size $= 1800$

being misclassified and therefore increase the safety of our predictions. On these grounds, we now try to determine a *reliable* misclassification probability. Such an assessment is crucial, as we operate in a safety-critical area and want to provide the user with safety assertions.

The question arises, whether it is possible to determine a *guaranteed* maximum misclassification probability or to even prove that misclassification is impossible. We believe that, when given only the information provided in this challenge, such a guaranteed boundary does not exist. The primary reason for this lies in the continuous nature of the assumed two-dimensional space I.

It would be possible to provide a guaranteed misclassification upper bound if we were able identify areas of I which *definitely* only contain red points or which *definitely* only contain green points. When observing the three datasets, it may seem clear that certain areas contain only red points or only green points, and that we should be able to guarantee 0% misclassification here if our certainty maps show that our network indeed only predicts the “correct” color in those areas. Although this may be true for the few points which we were given as a training set, we simply have no reliable information about the vast majority of points in I, as I is a continuous space which contains infinitely many points.

Assume we have two points $p, q \in S$ with $p \neq q$, meaning that p and q are red data points at different locations. No matter how small the distance between the two points, we can never assume that for an unknown point r, located in the middle of p and q, $r \in S$ is valid - it is well possible that Player A has chosen the subset S in such a way that it contains the arbitrarily close together points p and q, but not the arbitrarily small space between p and q in which r lies.

In practice, we can only work with the data points that we have to make *assumptions* about the data points that we don't have. This is precisely what our neural network does as it learns the patterns of the training set, estimates what the subset S probably looks like, and then uses this information to make assumptions about the validation set during prediction. Although it is impossible to guarantee the accuracy of our model, we can provide reliable misclassification probabilities by analysing the density of the datasets and the misclassification patterns of our network.

A. Probability distribution

When evaluating how certain we can be in the predictions of our model, the density of points in different areas of the provided datasets is an important factor to take into consideration. Areas of a dataset X which have a relatively high density of points provide us with much more information about the subset S_X than areas which have a low density of points - if we have no or very few given data points in a region, we cannot make reliable predictions about the color of new data points placed in that area. Higher trust can therefore be placed in the predictions of our network in high density areas than in low density areas. We have to incorporate this information when calculating the misclassification probability of our network.

For this sake, we split our datasets into grids where every square represents a subset of I and has a density value given by the number of points present in that square. The number of squares per grid varies from dataset to dataset due to the differences in the total number of points per dataset. The grid "resolution" was chosen so that for each dataset every grid square contains at least 10 points on average. Thereby, we receive a meaningful distribution of densities across the grid squares of each dataset. The grid sizes of the three datasets were chosen as follows:

	grid size
Dataset A	10×10
Dataset B	20×20
Dataset C	50×50

Note that the choice of grid sizes does not affect our final misclassification probability upper bounds, but is more of relevance for the readability and interpretability of the misclassification patterns presented in the next section.

The density distribution of the training points in a dataset X is directly proportional to the probability distribution P_X of that dataset by a constant factor of $\frac{1}{n_X}$. We know that data points are generated by drawing variates from the probability distribution P_X. As the already known training points were drawn from the same probability distribution, the density of training points can be seen as a "likelihood-of-appearance-map" for the next data points to be generated.

Drawing a density grid of a dataset X and then dividing all grid values by n_X therefore serves us as an estimation for the probability distribution P_X and as a measure for the density distribution of the training points in dataset X. Figure 14 shows the probability distribution map of dataset B, calculated as explained above. The color coding of each square shows the probability that the next randomly generated variate from P_B will land in that square, as a value between 0 and 1. The probability distribution maps of datasets A and C can be found in appendix E.

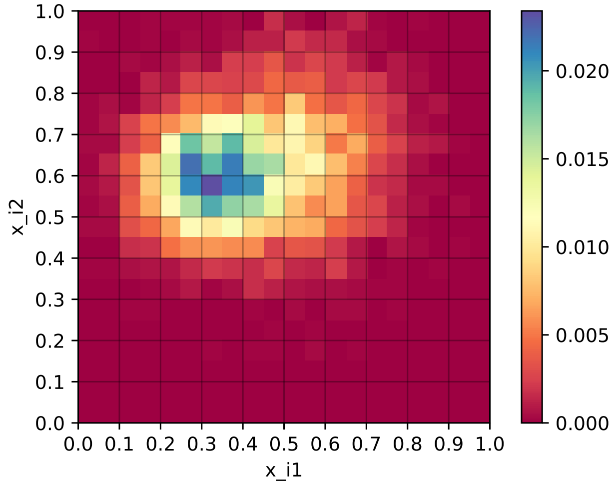

Fig. 14: The distribution map of dataset B.

B. Misclassification distribution

When classifying validation sets randomly generated from a probability distribution P_X for a dataset X, the misclassified points are generally not evenly distributed across the unit square I. Rather, there are areas where very few misclassifications occur and areas where misclassifications occur more frequently. The regions of high misclassification probability tend to be "border regions" between S and $I \backslash S$, especially when the borders are "blurry", i.e. there are few data points in the border region. This issue is especially prominent in dataset A, as can be seen in Figure 1, as dataset A has very few data points in total.

Determining the misclassification probability of validation points in different areas of a datasets can help us calculate a reliable total misclassification upper bound for that dataset. As in section III-A, we split our datasets into grids. To calculate representative misclassification probabilities per grid square, we performed multiple training and validation runs of our network on each dataset.

In each run, we first extract a random validation set from the given dataset and then train our (Glorot weight initialized) network on the remaining training set using the optimum training hyperparameters and penalty effect values calculated in section II. We then classify the validation set and for each square of the dataset grid save the proportion of validation points chosen from that square which were misclassified. We repeat this process n times and then calculate the average misclassification probability per square over the n runs. For every grid square, this leaves us with the probability that a newly generated point will be misclassified if it is located in that square. Additionally, we calculated the misclassification probability per grid square specifically for red points, as the misclassification probability of red points is of supreme interest.

The following number of training and validation runs were completed for each dataset to calculate the misclassification probability per grid square:

	n
Dataset A	2000
Dataset B	2000
Dataset C	1000

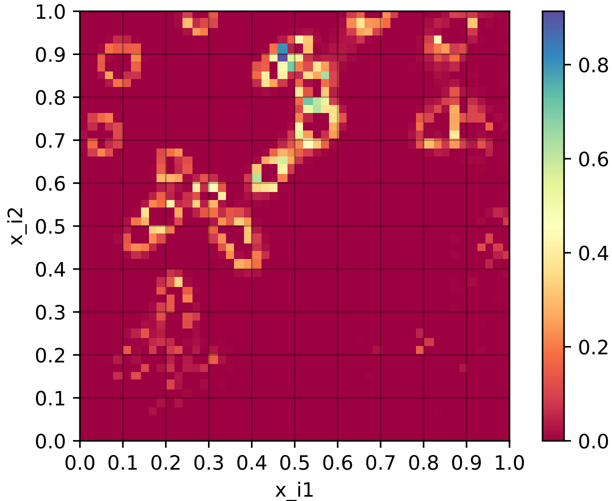

Fig. 15: Total misclassification probability per square in dataset C.

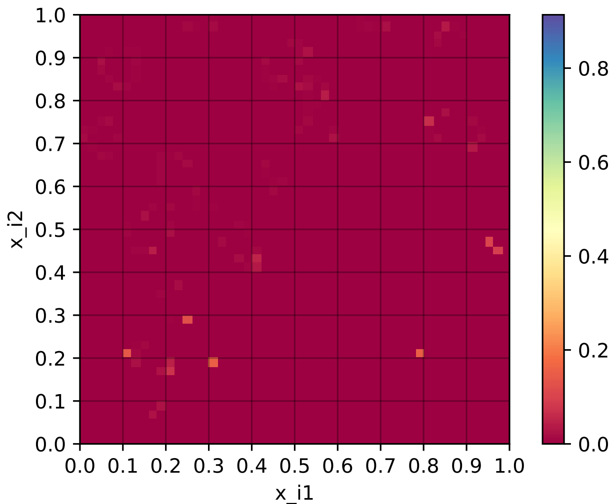

Fig. 16: Red misclassification probability per square in dataset C.

As we selected a random validation set from the given datasets in every run, and the datasets were randomly generated from the probability distributions P_X, it is safe to say that our results are representative of any validation set randomly generated from P_X.

Figure 15 and Figure 16 show the total and red misclassification probabilities per grid square in dataset C, as values between 0 and 1, respectively. Analogous plots can be found for datasets A and B in appendix F.

Note that for the training and validation runs on dataset C, we used the training hyperparameters batch size $= 1800$ and epoch number $= 2000$ instead of the calculated optimum parameters of batch size $= 2000$ and epoch number $= 2400$ due to time and computational power constraints. Despite not being optimal, our tests showed that the values used also produced very good results.

C. *Misclassification probability upper bound*

We can now calculate realistic and reliable misclassification probability upper bounds for both total and red misclassification by combining the probability that a new variate generated from the probability distribution P_X for a dataset X falls into a certain grid square and the probability that a point located in that square is misclassified. The prior information is encoded in the probability distribution maps computed in section III-A, and the latter information is encoded in the misclassification probability per grid square maps computed in section III-B.

We multiply the values of all grid squares in the probability distribution map of a dataset with the values of all corresponding grid squares in the misclassification probability map of that dataset. The result is the so-called *weighted misclassification probability map* where each grid square contains information about how likely it is that the next randomly generated data point lands in this square and is misclassified. In the case of using a red misclassification probability map, each grid square of the resulting weighted misclassification probability map contains information about how likely it is that the next randomly generated red data point lands in this square and is misclassified. The total and red weighted misclassification probability maps for all datasets can be found in appendix G.

By summing up the values of all grid squares, we arrive at a single average misclassification probability percentage. Doing this for total and red weighted misclassification probability maps for all datasets results in the following average total and red misclassification probability percentages:

	total	**red**
Dataset A	6.89%	2.16%
Dataset B	2.34%	0.19%
Dataset C	2.62%	0.10%

Note that these values are the average misclassification probability values over the 2000 (1000 for dataset C) training and validation runs for each dataset, meaning that about half of our runs fulfill these misclassification probabilities and about half of our runs do not fulfill them.

Individually looking at each of the 2000 (1000 for dataset C) training and validation runs lets us conclude that the following misclassification probability upper bounds are fulfilled in **99.9%** of all cases:

	total	**red**
Dataset A	15.23%	7.40%
Dataset B	8.63%	5.67%
Dataset C	5.45%	2.00%

D. *Further improving misclassification upper bounds*

The results presented in Section III-C clearly show that there is an inverse correlation between the size of a dataset and the misclassification probability when classifying new points in that dataset. These findings support our suggestion that a higher density of known data points results in greater classification confidence, as more information about the classification function is available to the network and the network can therefore predict the subset S_X of a dataset X more accurately.

Substantially expanding the sizes of the training datasets would most likely result in a great performance increase of the neural network across all datasets. This is a crucial step in lowering the given realistic upper bounds to a level which makes the model suitable for classification tasks in safety-critical areas.

The *misclassification probability per square* maps introduced here can further be employed to identify weak points in the training data. Areas which show high misclassification probabilities probably suffer from a lack of sufficient training data required to allow the model to learn the complex classification function at that point. The operators of the safety-critical system can use this information to selectively collect more training data in these particularly "dangerous" areas and thereby increase the model's accuracy and lower the total misclassification probability upper bounds.

IV. Scalability

Although the problem set by Siemens Mobility is certainly a simple and abstract one, the methods and approaches presented in this paper can easily be scaled in order to be applicable to complex real-world problems.

For one, neural networks can work well with higher-dimensional data and can be extended by dimensionality-reducing components such as autoencoders to improve performance when working with more than two dimensions.

Our idea of the penalty effect is scalable as well: manipulating loss functions to penalize different forms of misclassification to differing extents on higher-dimensional data poses no big problem with backpropagation-based neural network training.

Introducing certainty thresholds which only allow specific classifications to be made if a defined certainty threshold is reached can easily be implemented in neural networks which work with more than two output neurons. Different certainty thresholds for different classes and certainty thresholds which require specific value combinations from multiple output neurons are also feasible. A prerequisite for implementing a certainty threshold is that there must always be a form of safest classification which can be resorted to if none of the required thresholds are fulfilled. In this challenge, the safest classification which we default to if the certainty threshold is not fulfilled is red. In more complex real-world applications, the safest classification to resort to may dynamically adapt to the current state of the system. Take the example of a self-driving car. When the car is approaching a pedestrian crossing and the main machine learning algorithm of the car is not sure whether a pedestrian is currently crossing the street or not, you would want the car to default to a full emergency break, just to be on the safe side.

Finally, the process of dividing the possible range of values into subareas and calculating the misclassification probability per subarea is scalable to higher-dimensional data. The same goes for calculating the probability distribution of a given dataset. Visualization of these measurements will be impossible when working beyond three dimensions, but the mathematical operations are feasible beyond any dimensional limit.

V. Conclusion

We propose a fully-connected feedforward neural network architecture as a classifier for the provided datasets. Our model achieves an average total accuracy of 93.11%, 97.66%, and 97.38% and an average red accuracy of 97.84%, 99.81%, and 99.90% on datasets A, B, and C, respectively. Additional optimization of the network architecture could be undertaken to further increase classification performance and reduce training times. Our contribution to the Siemens Mobility AI Dependability Assessment Challenge is two-fold:

For one, we customize the classification model to increase the safety of its predictions. Under the assumption that red misclassifications have significantly worse consequences than green misclassifications, we tune the model to minimize red misclassifications. This makes our model inherently more safe than other classification models. Using a highly optimized neural network architecture with lower general misclassification rates would certainly further improve the safety of the model.

While misclassification rates determined during model validation do provide some idea of the safety of a model, they are highly dependent on the data that the model was *tested* on. To reach more dependable misclassification probability estimates, we focus on the knowledge that we have about the datasets. We determine areas in which our model struggles to correctly classify data points and calculate the probabilities that the next data point generated will fall into these areas. This allows us to provide reliable (though not guaranteed) upper bounds for total and red misclassification probability. In 99.9% of cases, our network fulfills the upper bounds of 15.23%, 8.63%, and 5.45% for total misclassification probability and the upper bounds of 7.40%, 5.67%, and 2.00% for red misclassification probability on datasets A, B, and C, respectively.

References

[1] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol. 25, 01 2012.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014.

[3] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016.

[4] F.-F. Li, R. Krishna, and D. Xu, "CS231n: Convolutional Neural Networks for Visual Recognition," http://cs231n.stanford.edu/, 2020.

[5] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

Appendix A
Optimal batch size and epoch numbers

Parameters \ Datasets	A	B	C
epochs	1450	2050	2000
batch size	16	912	2400
t_{bal}	-	-	0.3
t_{pen}	0.4	0.25	0.2
t_{cert}	0.9	0.9	0.95

TABLE II: Overview of the optimal parameters for training

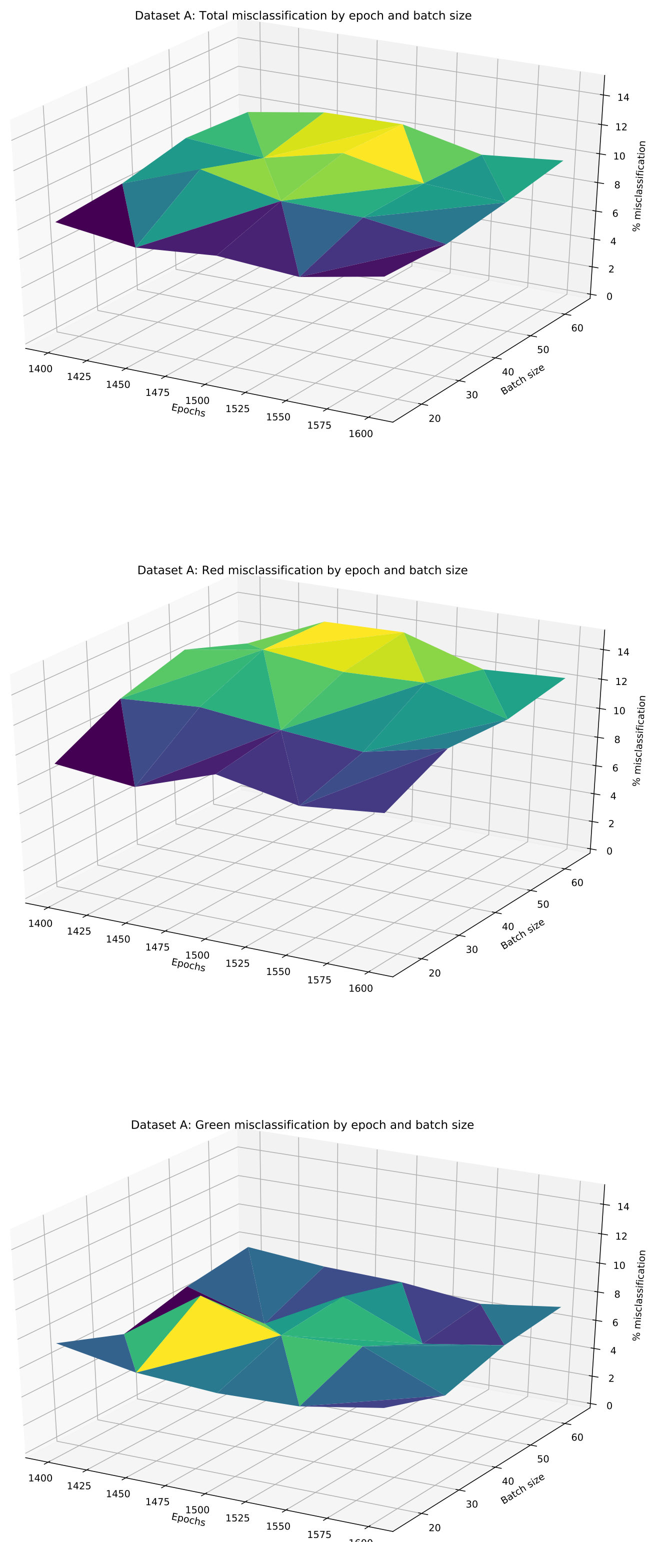

Fig. 17: Average % misclassification with respect to batch size and epoch number on dataset A.
Parameters: $n = 10$, $B = \{16, \ldots, 64\}$ in increments of 16, $E = \{1400, \ldots, 1600\}$ in increments of 50.

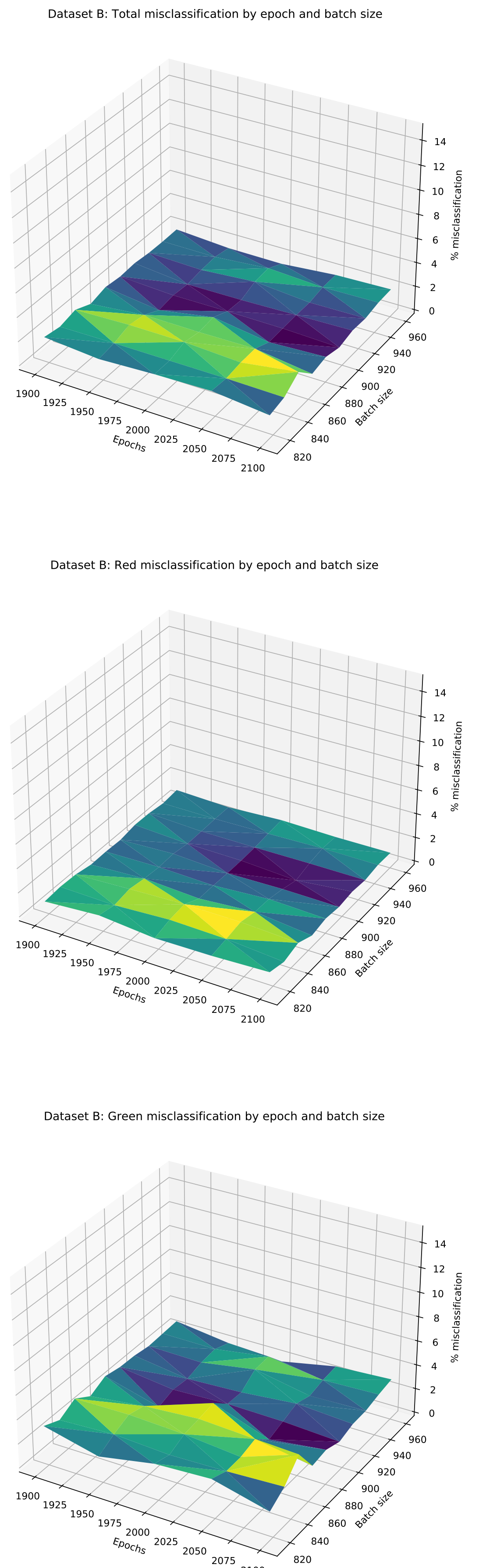

Fig. 18: Average % misclassification with respect to batch size and epoch number on dataset B.
Parameters: $n = 50$, $B = \{816, \ldots, 960\}$ in increments of 16, $E = \{1900, \ldots, 2100\}$ in increments of 50.

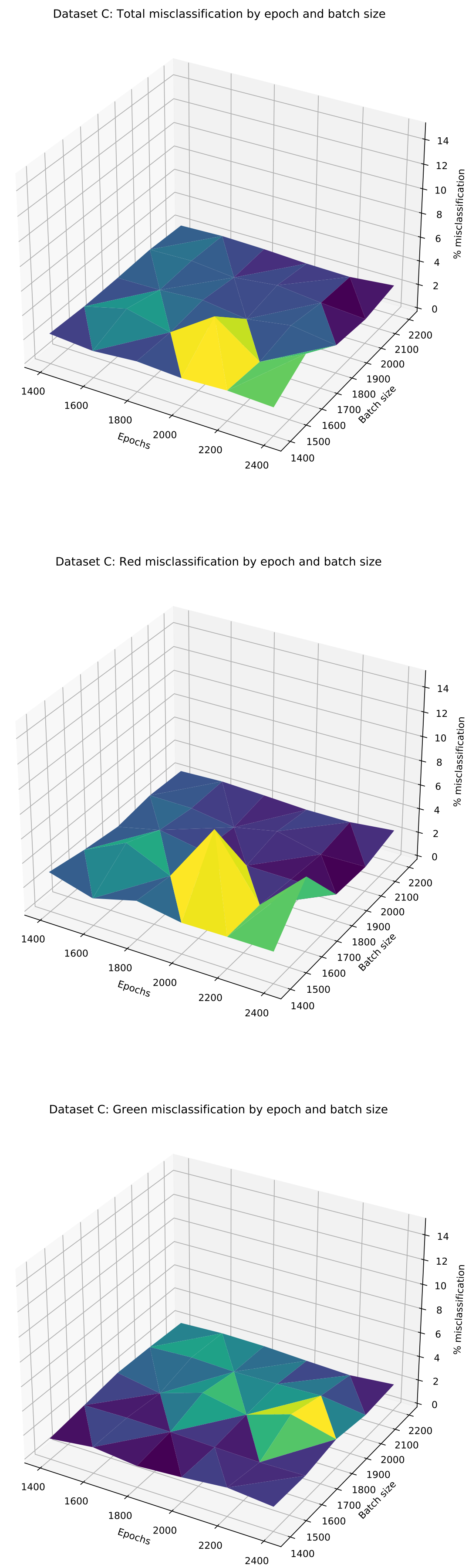

Fig. 19: Average % misclassification with respect to batch size and epoch number on dataset C.
Parameters: $n = 50$, $B = \{1400, \ldots, 2220\}$ in increments of 200, $E = \{1400, \ldots, 2400\}$ in increments of 200.

Appendix B
Certainty threshold values

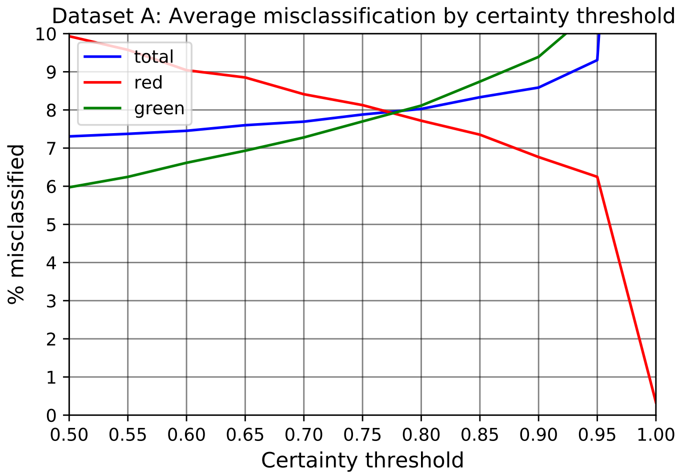

Fig. 20: Average % misclassification with respect to the certainty threshold value on dataset A.

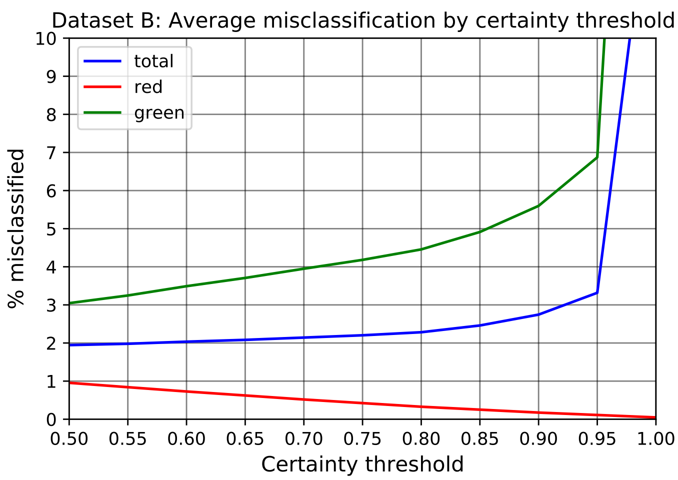

Fig. 21: Average % misclassification with respect to the certainty threshold value on dataset B.

Appendix C
Penalty values

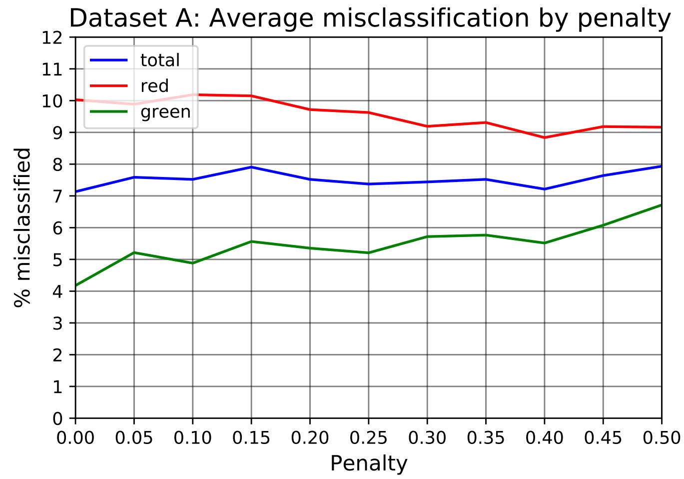

Fig. 22: Average % misclassification with respect to the penalty value on dataset A.

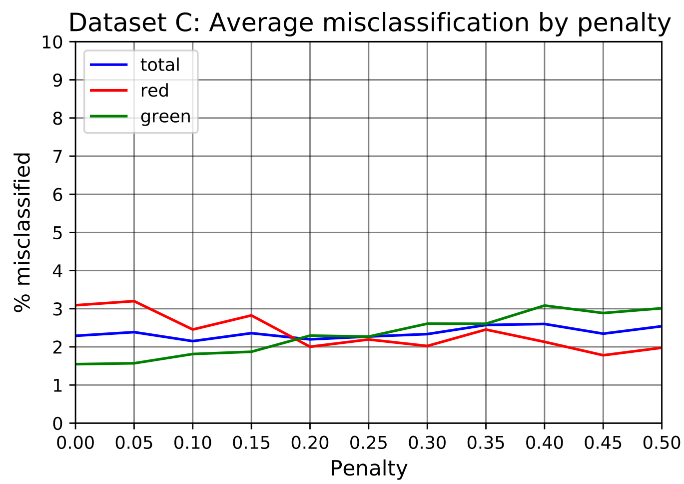

Fig. 23: Average % misclassification with respect to a penalty value from 0 to 0.5 on dataset C. A somewhat linear relation between penalty value and red/green misclassification rates become visible.

Appendix D
Loss plots

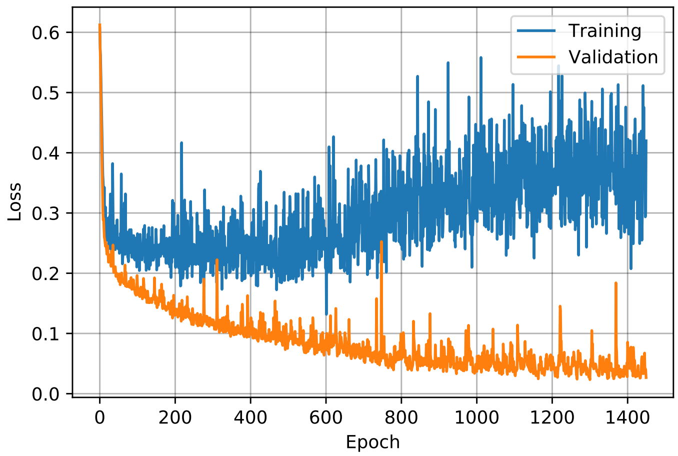

Fig. 24: Loss plot for a training and validation run on dataset A.

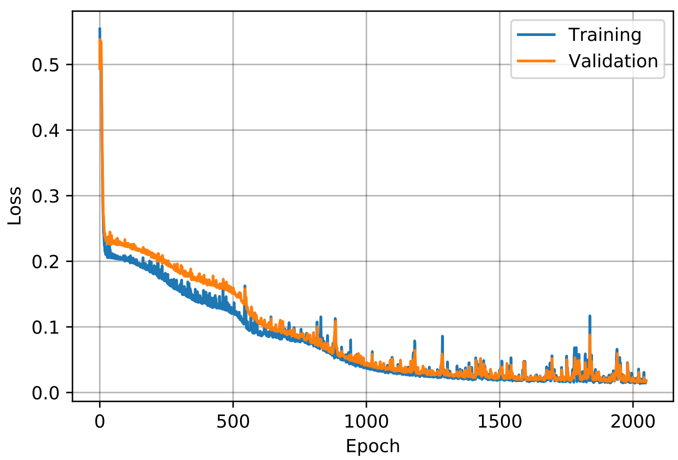

Fig. 25: Loss plot for a training and validation run on dataset B.

Appendix E
Data density

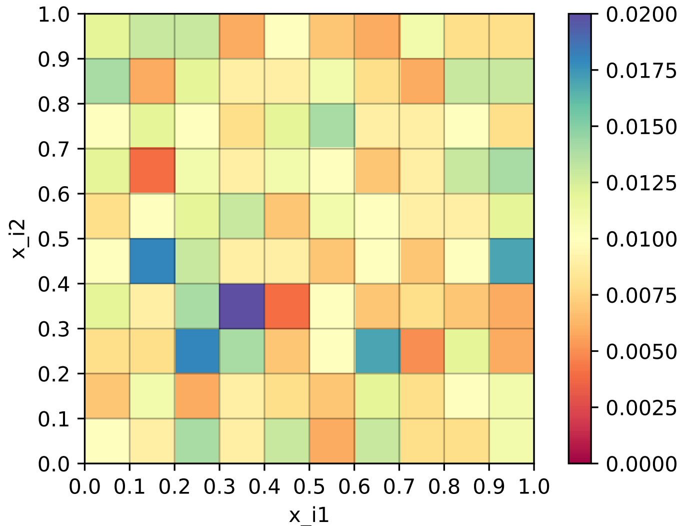

Fig. 26: Data distribution map for dataset A.

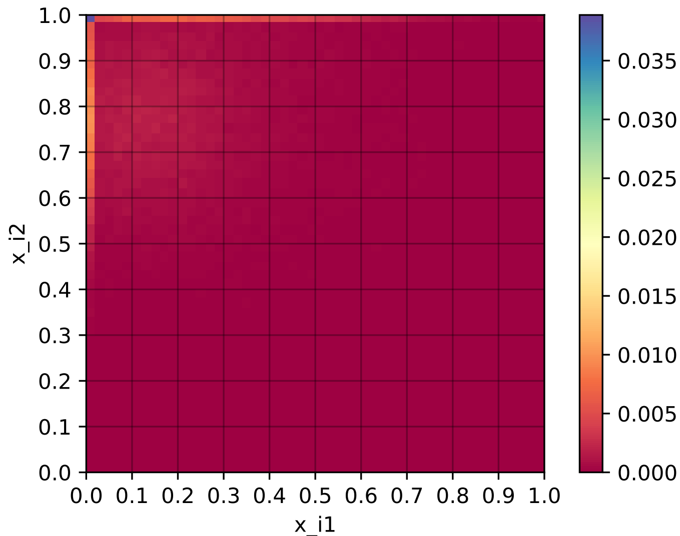

Fig. 27: Data distribution map for dataset C.

Appendix F
Misclassifications per square

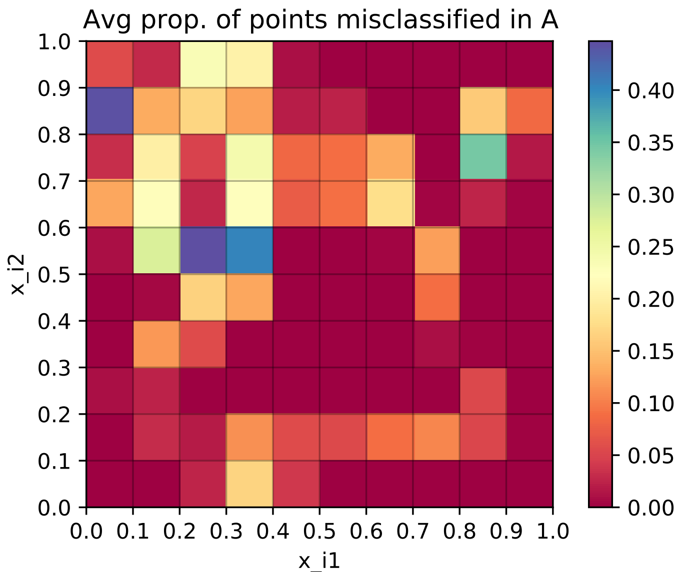

Fig. 28: Average rate of misclassifications per square on dataset A on 2000 validation runs.

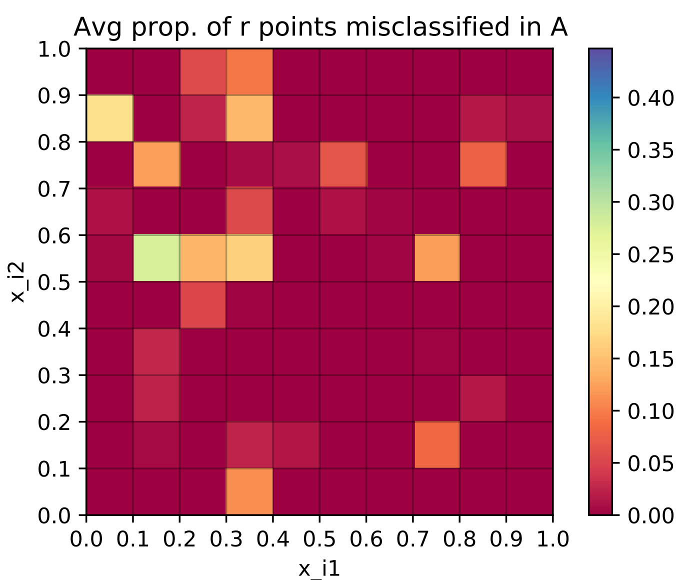

Fig. 29: Average rate of misclassifications of red data points per square on dataset A on 2000 validation runs.

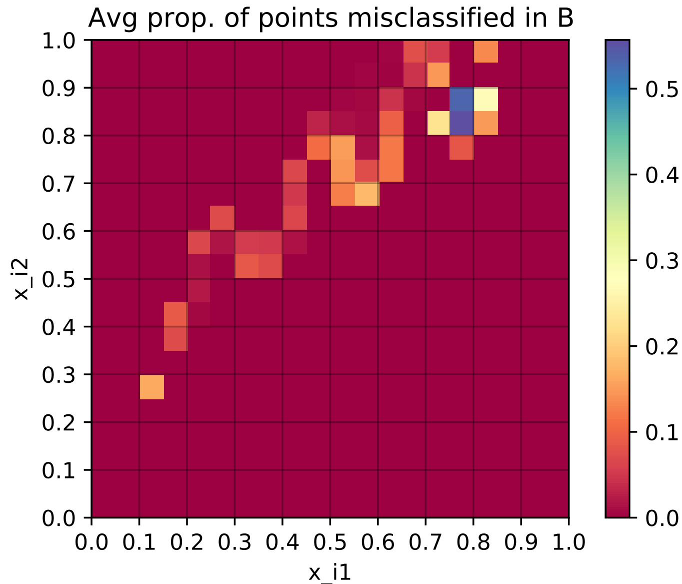

Fig. 30: Average rate of misclassifications per square on dataset B on 2000 validation runs.

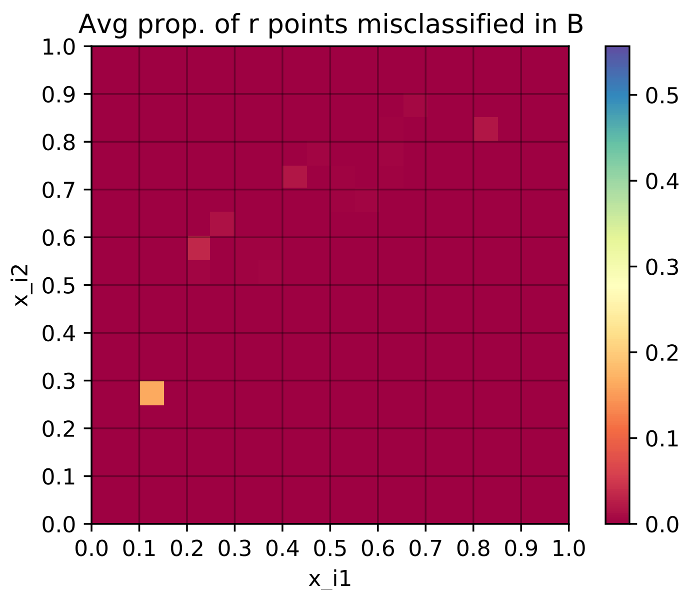

Fig. 31: Average rate of misclassifications of red data points per square on dataset B on 2000 validation runs.

Appendix G
Weighted misclassifications per square

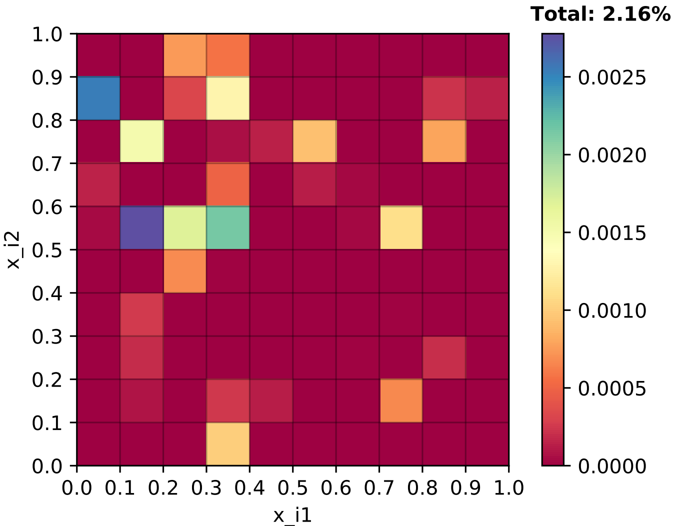

Fig. 32: Average rate of misclassifications of red data points per square, combined with the likelihood of a new data point being drawn in this square for dataset A.

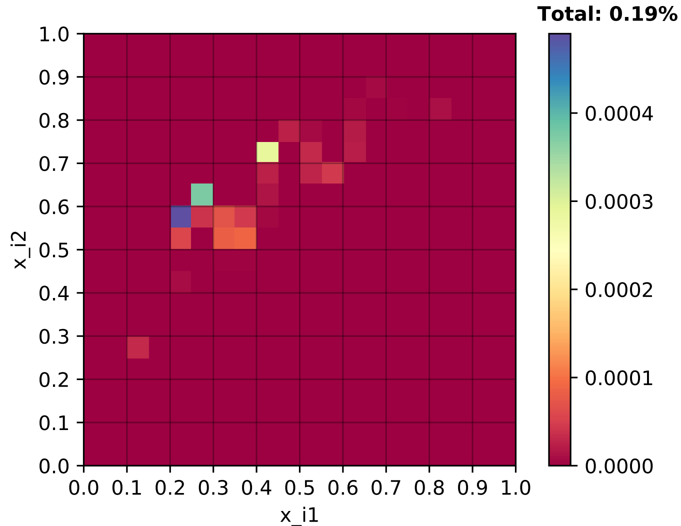

Fig. 34: Average rate of misclassifications of red data points per square, combined with the likelihood of a new data point being drawn in this square for dataset B.

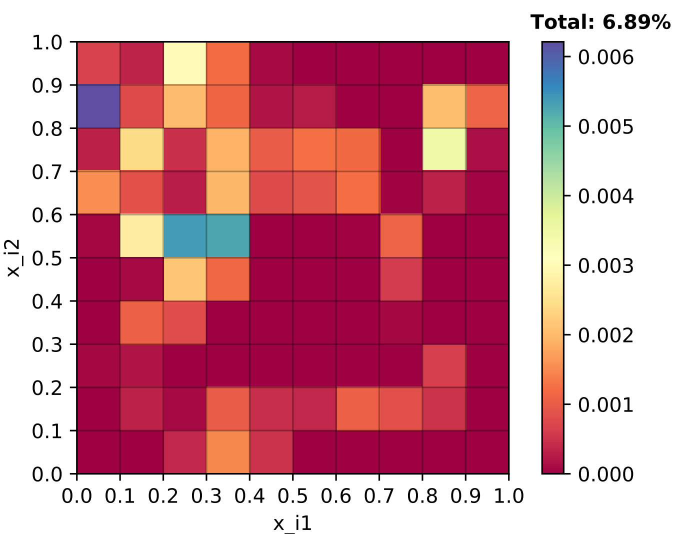

Fig. 33: Average rate of misclassifications per square, combined with the likelihood of a new data point being drawn in this square for dataset A.

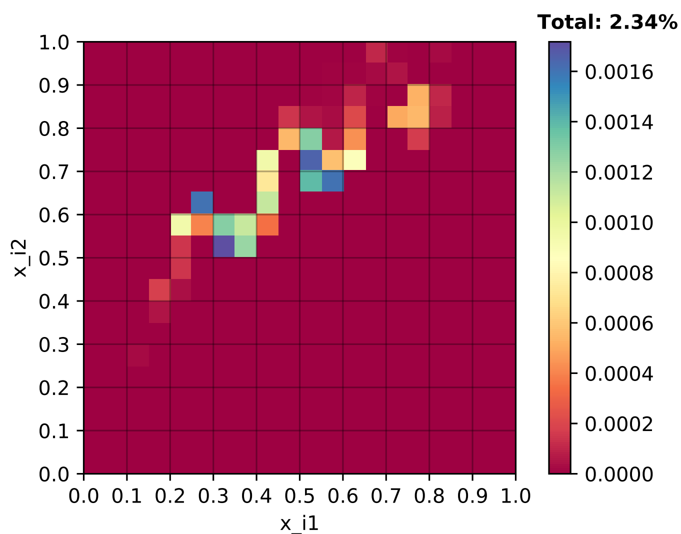

Fig. 35: Average rate of misclassifications per square, combined with the likelihood of a new data point being drawn in this square for dataset B.

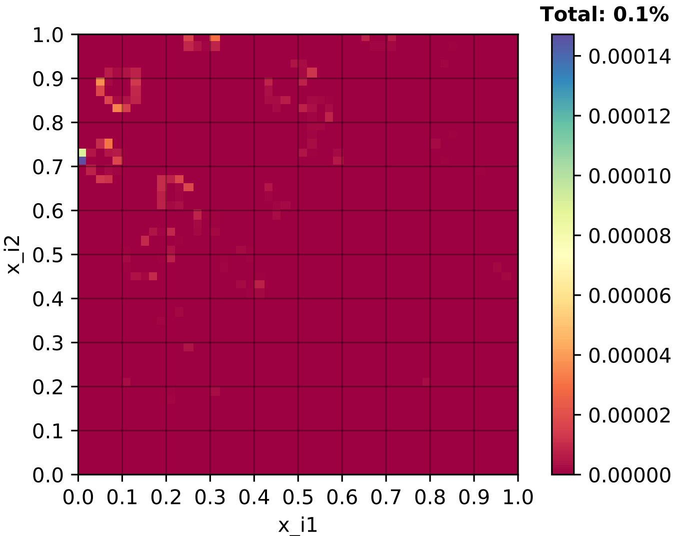

Fig. 36: Average rate of misclassifications of red data points per square, combined with the likelihood of a new data point being drawn in this square for dataset C.

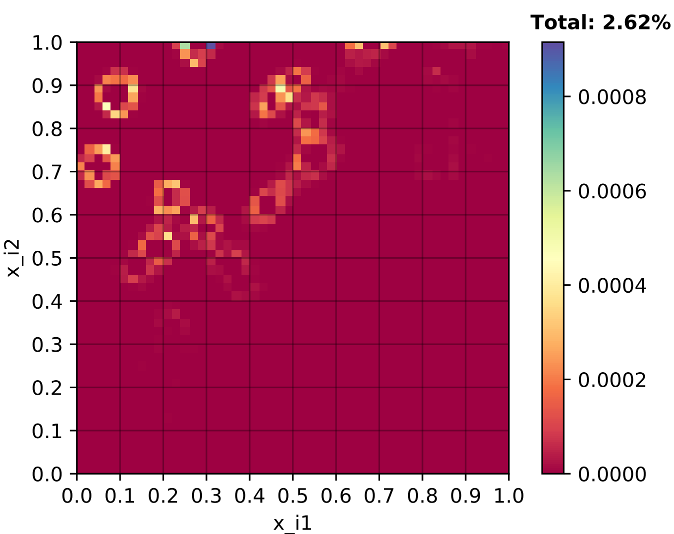

Fig. 37: Average rate of misclassifications per square, combined with the likelihood of a new data point being drawn in this square for dataset C.

Safety considerations in neural network classification

Source code

Eric Tillmann Bill
RWTH Aachen University
Aachen, Germany
eric.bill@rwth-aachen.de

Robert Maximilian Giesler
RWTH Aachen University
Aachen, Germany
robert.giesler@rwth-aachen.de

Felix Maximilian Knispel
RWTH Aachen University
Aachen, Germany
felix.knispel@rwth-aachen.de

May 2, 2021

1 Imports, Config & GPU Info

[ ]:
```
# Tensorflow and Keras
import tensorflow as tf
from tensorflow import keras
from IPython.display import clear_output

# Arithmetic Operations
import pandas as pd
import numpy as np
import random
import math

# Data visualization
from matplotlib import pyplot as plt

# Progress calculation
import sys
import time
from datetime import date
```

2 Global Constants and Variables

[ ]:
```
#@title Global Constants

# Dictionaries
COLORS = {0 : 'green', 1 : 'red', 'green' : 0, 'red' : 1}
SOURCES = {'A' : 'https://drive.google.com/file/d/1hAzAKZNpmSclSI7HnV_cRjpMS4Kh5r1q/
↪view?usp=sharing', 'B' : 'https://drive.google.com/file/d/
↪12VlecL-5iYs-BFpnTOba1x65jWofBX1P/view?usp=sharing', 'C' : 'https://drive.google.com/
↪file/d/1-ZORuJIi1cZcqrrmV6TqTO01PwI2OiBY/view?usp=sharing'}
```

```
SOURCE_SIZE = {'A': 1000,'B' : 5000, 'C' : 50000}

CURRENT_SET = 'B'

# Balancing dataset to threshold
THRESHOLD_DATA = 0.3

# Threshold for balanced validation set
THRESHOLD_VAL = 0.4

# Minimum certainty required to predict green
MISCLASS_THRESHOLDS = {'A' : 0.9, 'B' : 0.9, 'C' : 0.95}
MIN_GREEN_CERT = MISCLASS_THRESHOLDS[CURRENT_SET]

# Randomly selecting validation points
VAL_INDICES = random.sample(range(SOURCE_SIZE[CURRENT_SET]), int(0.
 ↪16*SOURCE_SIZE[CURRENT_SET]))

# Penalties applied to false green classifications in custom loss function
PENALTIES = {'A' : 0.4, 'B' : 0.1, 'C' : 0.2}
PENALTY = PENALTIES[CURRENT_SET]
```

[ ]:
```
#@title Global Variables

# Initialize current dataset as empty dataframe
DATASET = pd.DataFrame()
# Dataset previously used
PREV_SET = None

# Time prediciton
PREV_TIME = 0
PB_START_TIME = 0

# Random number seed
random.seed(time.time())
```

3 Functions

[ ]:
```
#@title Data Operations

def getDataSet(dataset=None):
  """Returns pandas.DataFrame of dataset.

  Args:
    dataset: char, optional
      The dataset to return. 'A', 'B', or 'C'.
  """
  global DATASET
  global PREV_SET

  if dataset == None:
    dataset = CURRENT_SET
```

```
  try:
    if DATASET.empty:
      path = 'https://drive.google.com/uc?export=download&id='+SOURCES[dataset].
↪split('/')[-2]
      DATASET = pd.read_excel(path)
      PREV_SET = dataset

    if dataset != PREV_SET:
      path = 'https://drive.google.com/uc?export=download&id='+SOURCES[dataset].
↪split('/')[-2]
      DATASET = pd.read_excel(path)
      PREV_SET = dataset

  except:
    print('Exception in getDataSet occured')
    print('Going to sleep for 2 minutes and trying again')
    time.sleep(120)
    DATASET = getDataSet(dataSet)

  return DATASET.copy()


def separateValidationSet(dataSet, validationIndices):
  """Separates a subset of points from dataSet as validation points.

  Validation points are extracted and deleted from dataSet to be used for
  validation later on.

  Args:
    dataSet: pandas.DataFrame
      Dataframe with columns 'x_i1', 'x_i2', 'l_i1'. Dataset which the
      validation points are extracted from.
    validationIndices: 1-D list of ints
      The elements corresponding to these indices are extracted from dataSet.

  Returns:
    2-tuple of the form (valSet_points, valSet_labels), where valSet_points
    is a np.array of shape (x,2) and valSet_labels is a np.array of shape (x,1).
  """
  # Checking for the right type
  if not isinstance(dataSet, pd.DataFrame):
    raise TypeError(f'dataSet is of type: {type(dataSet)}, but should be \
      {pd.DataFrame}')

  # Checking for the right shape
  if len(np.array(validationIndices).shape) != 1:
    raise TypeError(f'The shape of the parameter validationIndices is: \
      {np.array(validationIndices).shape}, but it should be 1 dimensional')

  valSet_points = dataSet[['x_i1','x_i2']].loc[validationIndices]
  valSet_labels = dataSet['l_i'].loc[validationIndices]
```

```
    # Saving the validation points
    valSet_points = np.array(valSet_points)
    valSet_labels = np.array(valSet_labels).astype('float')

    # Removing the validation point
    dataSet.drop(index=validationIndices, inplace=True)
    dataSet.reset_index(inplace=True)

    return (valSet_points, valSet_labels)


def timeCalc():
    """Calculates time between previous call and current call.

    Returns:
      Time difference in minutes as float.
    """
    global PREV_TIME
    if PREV_TIME == 0:
      PREV_TIME = time.time()
      return 0

    res = (time.time() - PREV_TIME) / 60
    PREV_TIME = time.time()
    return res


def balanceDataset(dataSet, threshold, verbose=1):
    """Artificially balances dataSet by duplicating red or green points.

    Args:
      dataSet: pandas.DataFrame
        Dataframe with columns 'x_i1', 'x_i2', 'l_i1'. The datset to be balanced.
      threshold: float between 0 and 0.5
        The function duplicates red or green points until the fraction of points
        of the less frequent color is at least equal to the threshold.
      verbose: int, optional
        If set to 0 the function does not print to the console, otherwise it prints
        the results.

    Returns:
      pandas.DataFrame with columns 'x_i1', 'x_i2', 'l_i1'.
    """
    total_number_of_points = dataSet.shape[0]
    number_of_green_points = dataSet.loc[dataSet["l_i"] == 0].shape[0]
    number_of_red_points = dataSet.loc[dataSet["l_i"] == 1].shape[0]

    amount = 0

    if number_of_red_points / total_number_of_points < threshold:
      amount = int( (threshold * total_number_of_points - number_of_red_points) // (1 -
  ↪threshold) )
      red_points = dataSet.loc[dataSet['l_i'] == 1] #Getting all red points
```

```
    chosen_points = red_points.sample(amount, replace=True) #Selecting a random subset␣
↪of red points
    dataSet = dataSet.append(chosen_points, ignore_index=True) #appending the subset

  if number_of_green_points / total_number_of_points < threshold:
    amount = int( (threshold * total_number_of_points - number_of_green_points) // (1␣
↪- threshold) )
    green_points = dataSet.loc[dataSet['l_i'] == 0] #Getting all green points
    chosen_points = green_points.sample(amount, replace=True) #Selecting a random␣
↪subset of green points
    dataSet = dataSet.append(chosen_points, ignore_index=True) #appending green subset

  dataSet = dataSet[['x_i1','x_i2','l_i']]

  total_number_of_points = dataSet.shape[0]
  number_of_green_points = dataSet.loc[dataSet["l_i"] == 0].shape[0]
  number_of_red_points = dataSet.loc[dataSet["l_i"] == 1].shape[0]

  if verbose > 0:
    print(f'Artificially exended by {amount} points')
    print(f'Relation is now: {round(number_of_green_points / total_number_of_points,␣
↪2)}',
            f'green : {round(number_of_red_points / total_number_of_points, 2)} red ')

  return dataSet


def getBalancedValSetIndices(dataSet, size, threshold):
  """Get indices of validation points such that neither color is represented
    less than (threshold*100)% of the validation set.

  Args:
    dataSet: pandas.DataFrame
      Dataframe with columns 'x_i1', 'x_i2', 'l_i1'. Dataset from
      which the validation points are to be chosen.
    size: int
      Size of the validation set.
    threshold: float between 0 and 1
      Fraction of validation points which each color must at least
      represent.

  Returns:
    1-D array of ints (indices).

  Raises:
    ValueError: If the requested size of the validation set is not feasible.
  """
  random.seed(time.time())

  # Amount of points for each color
  amount_g = int(random.randint(size*threshold, size*(1-threshold)))
  amount_r = size - amount_g
```

```
  # Indices of each points with the specific color
  indices_g = np.where(dataSet['l_i'] == 0)[0]
  indices_r = np.where(dataSet['l_i'] == 1)[0]

  # Check if possible
  if indices_g.shape[0] + indices_r.shape[0] < size:
    raise ValueError('The requested size of the validation set is not feasible')

  if indices_r.shape[0] < amount_r:
    indices_g += amount_r - indices_r.shape[0]

  if indices_g.shape[0] < amount_g:
    indices_r += amount_g - indices_g.shape[0]

  # Randomly selceting a subset for each color
  indices_g = np.random.choice(indices_g, amount_g)
  indices_r = np.random.choice(indices_r, amount_r)

  # Concatenate and shuffle the chosen subsets
  indices = np.concatenate([indices_g, indices_r])
  np.random.shuffle(indices)

  return indices


def thresholdPredict(data, model, threshold):
  """Generates output predictions for the input samples. Points are only
    predicted as green if the model's certainty for green is >= threshold. All
    other points are predicted red.

  Args:
    data: array-like, tensors, tf.data dataset...
      Input samples.
    model: keras.model
      The model to perform the predictions.
    threshold: float between 0.5 and 1
      The minimum certainty required for the network to predict a point as green.

  Returns:
    Numpy array(s) of predictions.
  """
  prediction = model.predict(data)

  for i in range(len(prediction)):
    if prediction[i,0] >= 0.5 and prediction[i,0] < threshold:
      temp = prediction[i,0]
      prediction[i,0] = prediction[i,1]
      prediction[i,1] = temp

  return prediction
```

[ ]:

```
#@title Visualization

def printProgressBar(iteration, total, prefix = '', suffix = '', decimals = 1,
```

```
                    length = 100, fill = '#'):
  """Prints a progress bar.

  Args:
    iteration: int
      Current progress step as. (iteration/total progress).
    total: int
      Total progress steps until completion.
    prefix: str, optional
      Printed infront of the progress bar.
    suffix: str, optional
      Printed behind ETA.
    decimals: int, optional
      Number of decimal places of percentage progress.
    length: int, optional
      Length of the progress bar in characters.
    fill: char, optional
      Filler of the progress bar.
  """
  # Preparing strings
  percentage_progress = (100*(iteration/float(total)))
  percent = ("{0:." + str(decimals) + "f}").format(percentage_progress)
  filledLength = int(length * iteration // total)
  bar = fill * filledLength + '-' * (length - filledLength)

  # Bob's alternative time calculation
  if iteration == 0:
    global PB_START_TIME
    PB_START_TIME = time.time()
    time_so_far = 0
    time_remaining = 0
  else:
    time_so_far = time.time() - PB_START_TIME
    time_remaining = time_so_far/percentage_progress * (100-percentage_progress)

  sys.stdout.write(f'\r{prefix} |{bar}| {percent}% | ETA: {round((time_remaining/60),␣
↪2)} minutes | {suffix}')
  sys.stdout.flush()

  # Erease progress bar on complete
  if iteration == total:
    global PREV_TIME
    PREV_TIME = 0
    sys.stdout.write('\r')
    sys.stdout.flush()


def makePlot(dataSet=CURRENT_SET, correct_pred_points = np.array([]),
             incorrect_pred_points = np.array([]), drawGrid=True,
             showTitle=False, savePlot=False, path=''):
  """"Plots green and red points and markers as scatter graph.

  Args:
```

```
  dataSet: pandas.DataFrame or char, optional
    Dataframe with columns 'x_i1', 'x_i2', 'l_i1' or char 'A', 'B', or 'C'.
    Dataset to be plotted.
  correct_pred_points: 2-D list, optional
    List of shape (x,2) containing correctly predicted points. Marked as black
    'x' on scatter graph.
  incorrect_pred_points: 2-D list, optional
    List of shape (x,2) containing incorrectly predicted points. Marked as
    black '*' on scatter graph.
  drawGrid: boolean, optional
    Whether to draw a grid on the plot or not.
  showTitle: boolean, optional
    Whether to show the title of the plot or not.
  savePlot: boolean, optional
    Whether to save the plot or not.
  path: str, optional
    Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'

Raises:
  TypeError: If dataSet is not an instance of pd.DataFrame or char or the
  other parameters do not have the required shape.
"""
# Preparing optional parameters
dataSet_char = None
if type(dataSet) == str:
  dataSet_char = dataSet
  dataSet = getDataSet(dataSet)

if dataSet_char == None:
  dataSet_char = CURRENT_SET

if isinstance(correct_pred_points, list):
  correct_pred_points = np.array(correct_pred_points)
if isinstance(incorrect_pred_points, list):
  incorrect_pred_points = np.array(incorrect_pred_points)

# Checking for the right type
if not isinstance(dataSet, pd.DataFrame):
  raise TypeError(f'dataSet is of type: {type(dataSet)}, but should be ' +
                  f'{pd.DataFrame}')

# Checking for the right shape
if (correct_pred_points.shape != (correct_pred_points.shape[0],2)
    and np.array(correct_pred_points).shape != (0,)):
  raise TypeError(f'The shape of the parameter correct_pred_points is: \
    {np.array(correct_pred_points).shape}, but it should be 2 dimensional')

if (incorrect_pred_points.shape != (incorrect_pred_points.shape[0],2)
    and np.array(incorrect_pred_points).shape != (0,)):
  raise TypeError(f'The shape of the parameter incorrect_pred_points is: \
    {np.array(incorrect_pred_points).shape}, but it should be 2 dimensional')

# Creating a subplot
```

```
  fig, ax = plt.subplots()

  # Scattering all points
  x = dataSet['x_i1']
  y = dataSet['x_i2']
  c = [COLORS[i] for i in dataSet['l_i']]

  ax.scatter(x, y, c=c)

  # Adding markers to the specified points
  if correct_pred_points.shape[0] > 0:
    ax.scatter(correct_pred_points[:, 0], correct_pred_points[:, 1],
               marker = "x", c = 'black', label='correct')
  if incorrect_pred_points.shape[0] > 0:
    ax.scatter(incorrect_pred_points[:, 0], incorrect_pred_points[:, 1],
             marker = "*", c = 'black', label='incorrect')

  if correct_pred_points.shape[0] > 0 or incorrect_pred_points.shape[0] > 0:
    plt.legend()

  # Setting parameters for ploting
  plt.axis('scaled')
  if showTitle:
    ax.set_title(f'Dataset {dataSet_char}')
  ax.set_xlabel('x_i1')
  ax.set_ylabel('x_i2')
  ax.set_xlim((0,1))
  ax.set_ylim((0,1))
  ax.set_xticks([i/10 for i in range(11)])
  ax.set_yticks([i/10 for i in range(11)])
  if drawGrid == True:
    ax.grid(alpha=0.3, color='black')
  plt.show()

  # Save plot
  if savePlot == True:
    fig.savefig(f'{path}Dataset_{dataSet_char}.pdf')
    fig.savefig(f'{path}Dataset_{dataSet_char}.png', dpi=300)



def makeCertaintyMap(model, accuracy=100, specific_color=None,
                     useThresholdPredict=False, drawGrid=True, verbose=1,
                     savePlot=False, path=''):
  """Plots the prediction certainty of the model for a grid of data points.

  All data points have x and y values between 0 and 1.

  Args:
    model: keras model
      The model who's preidction certainty is to be plotted.
    accuracy: int, optional
      Data points are spaced 1/accuracy apart along the x and y axis. The grid
```

```
      of data points plotted has the dimension accuracy*accuracy.
    specific_color: 0 or 1, optional
      If 0, plots the model's certainty that a data point is green for all
      points in the grid. If 1, analogously for red.
    useThresholdPredict: boolean, optional
      Whether to use thresholdPredict (True) or regular model.predict (False).
    drawGrid: boolean, optional
      Whether to draw a grid on the plot or not.
    verbose: 0 or 1, optional
      Whether to plot the certainty map or not.
    savePlot: boolean, optional
      Whether to save the plot or not.
    path: str, optional
      Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'

  Returns:
    2-D np.array of the shape (accuracy,accuracy).

  Raises:
    TypeError: If specific_color is not 'None', '0' or '1', or if accuracy is not
      an int.
  """
  # Exceptions
  if specific_color != None:
    if specific_color != 0 and specific_color != 1:
      raise TypeError(f'Invalid value for specific_color. Value is {specific_color}, \
        but should be "None", "0" or "1".')

  if not isinstance(accuracy, int):
    raise TypeError(f'Invalid type for accuracy. Type is {type(accuracy)}, but \
      should be int.')

  accuracy_map = np.zeros((accuracy, accuracy))

  for i in range(accuracy):
    array = np.array([[j/accuracy, i/accuracy] for j in range(accuracy)])

    # Predict points
    if useThresholdPredict == True:
      result = thresholdPredict(array, model, MIN_GREEN_CERT)
    else:
      result = model.predict(array)

    if specific_color != None:
      # Saving the prediction for the specified color
      accuracy_map[i] = result[:, specific_color]

    else:
      result = result.max(axis=1) # Getting each max value

      # Normalize the values which are between 0.5 <-> 1 to 0 <-> 1
      accuracy_map[i] = result
```

```
    # Print current progress
    printProgressBar(i, accuracy-1)

  if verbose > 0:
    fig, ax = plt.subplots()

    if specific_color != None:
      plt.imshow(accuracy_map, origin='lower', cmap='tab20b', vmin=0, vmax=1,
                 extent=[0, 1, 0, 1])
      ax.set_title(f'Certainty for {COLORS[specific_color]} in {CURRENT_SET}')
    else:
      plt.imshow(accuracy_map, origin='lower', cmap='tab20b', vmin=0.5, vmax=1,
                 extent=[0, 1, 0, 1])
      ax.set_title(f'Total certainty in {CURRENT_SET}')

    plt.colorbar()
    ax.set_xlabel('x_i1')
    ax.set_ylabel('x_i2')
    ax.set_xlim((0,1))
    ax.set_ylim((0,1))
    ax.set_xticks([i/10 for i in range(11)])
    ax.set_yticks([i/10 for i in range(11)])
    if drawGrid == True:
      ax.grid(alpha=0.3, color='black')
    plt.show()

  # Save plot
  if savePlot == True:
    fig.savefig(f'{path}CertaintyMap_{CURRENT_SET}.pdf')
    fig.savefig(f'{path}CertaintyMap_{CURRENT_SET}.png', dpi=300)

  return accuracy_map



def plotLoss(history, savePlot=False, showTitle=False, drawGrid=True, path=''):
  """Plots training loss and validation loss with respect to training epochs.

  Args:
    history: keras History
      history of keras model.
    savePlot: boolean, optional
      Whether to save the plot or not
    showTitle: boolean, optional
      Whether to show the title of the plot or not.
    drawGrid: boolean, optional
      Whether to draw a grid on the plot or not.
    path: str, optional
      Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'
  """
  fig, ax = plt.subplots()
  if 'val_loss' in history.history:
    ax.plot(history.history['val_loss'])
```

```python
  ax.plot(history.history['loss'])
  if showTitle:
    ax.set_title(f'Training and validation loss on {CURRENT_SET}')
  ax.set_ylabel('Loss')
  ax.set_xlabel('Epoch')
  plt.legend(['Training', 'Validation'], loc='upper right')
  if drawGrid == True:
    ax.grid(alpha=0.3, color='black')
  plt.show()

  # Save plot
  if savePlot == True:
    fig.savefig(f'{path}LossPlot_{CURRENT_SET}.pdf')
    fig.savefig(f'{path}LossPlot_{CURRENT_SET}.png', dpi=300)



def showPredictions(model, history, valSet_points, valSet_labels,
                    useThresholdPredict=False, showCorrectPoints=False,
                    drawGrid=True):
  """Plots the predictions for the validation points.

  Args:
    model: keras model
      Model which performs the predictions.
    valSet_points: 2-D array of shape (x,2)
      Data points used for validation.
    valSet_labels: 1-D array of shape (x,)
      Ground truth labels of the validation points.
    useThresholdPredict: boolean, optional
      Whether to use thresholdPredict (True) or regular model.predict (False).
    showCorrectPoints: boolean, optional
      Whether correctly classified points should be marked as black 'x' or not.
    drawGrid: boolean, optional
      Whether to draw a grid on the plot or not.

  Returns:
    2-dimensional numpy array of shape (x,2) with the predictions for the
    validation points.
  """
  # Predict the validation points
  if useThresholdPredict == True:
    prediction = thresholdPredict(valSet_points, model, MIN_GREEN_CERT)
  else:
    prediction = model.predict(valSet_points)

  # Identifying correctly and incorrectly classified points
  correct_indices = np.where((valSet_labels == np.argmax(prediction, axis=1)) == True)
  incorrect_indices = np.where((valSet_labels == np.argmax(prediction, axis=1)) ==␣
↪False)

  number_of_points = np.bincount(np.argmax(prediction, axis=1))
```

```
total_misclassifications = np.bincount(valSet_labels == np.argmax(prediction, axis=1))[0]
red_misclassifications = len(np.where(valSet_labels[incorrect_indices] == 1)[0])
green_misclassifications = len(np.where(valSet_labels[incorrect_indices] == 0)[0])

#Average misclassification certainty
misclass_certainties = []
for i in incorrect_indices[0]:
  misclass_certainties.append(np.max(prediction[i]))
avg_misclass_certainty = sum(misclass_certainties)/total_misclassifications

valAccuracy = 100 - (total_misclassifications/sum(number_of_points))*100

print('Validation accuracy: {:.2f}%'.format(valAccuracy))
print(f'Predictions for green: {number_of_points[0]} / {len(valSet_labels)}')
print(f'Predictions for red: {number_of_points[1]} / {len(valSet_labels)}')
print(f'Points misclassified: {total_misclassifications}')
print(f'Red points misclassified: {red_misclassifications}')
print(f'Green points misclassified: {green_misclassifications}')
print('Average misclassification certainty: {:.2f}'.format(avg_misclass_certainty))

if showCorrectPoints:
  makePlot(correct_pred_points=valSet_points[correct_indices],
         incorrect_pred_points=valSet_points[incorrect_indices],
         drawGrid=drawGrid)
else:
  makePlot(incorrect_pred_points=valSet_points[incorrect_indices],
           drawGrid=drawGrid)

# Make bar graph showing red and green misclassifications
bars = ('Red', 'Green')
height = [red_misclassifications, green_misclassifications]
x_pos = np.arange(len(bars))

fig, ax = plt.subplots()
ax.bar(x_pos, height, width=0.35, color=['red', 'green'])

ax.set_ylabel('Misclassifications')
ax.set_title('Misclassifications by color')
ax.set_xticks(x_pos)
ax.set_xticklabels(bars)

rects = ax.patches # Array of bars

labels = [red_misclassifications, green_misclassifications]

for rect, label in zip(rects, labels): # Add labels above bars
    height = rect.get_height()
    ax.text(rect.get_x() + rect.get_width() / 2, height, label,
            ha='center', va='bottom')

plt.show()
```

```python
  return prediction



def makeDensityMap(accuracy, dataSet=CURRENT_SET, significance=0.1,
                   cmap=plt.cm.get_cmap('Spectral'), specific_color = None,
                   drawGrid=True, verbose=1, savePlot=False, path=''):
  """Creates a heatmap of the density of dataSet.

    Args:
      accuracy: int, optional
        Data points are spaced 1/accuracy apart along the x and y axis. The grid
        of data points plotted has the dimension accuracy*accuracy.
      dataSet: pandas.DataFrame or char, optional
        Dataframe with columns 'x_i1', 'x_i2', 'l_i1' or char 'A', 'B', or 'C'.
        Dataset to be plotted.
      signifcance: float between 0 and 1, optional
        Determines the radius in which neighbours are being counted for the
        density of a particular point.
      cmap: matplotlib colormap, optional
        Is used for color coding the density of the dataset at the end.
      specific_color: 0 or 1, optional
        If 0, a heatmap of only green points is computed. If 1, analogously for
        red.
      drawGrid: boolean, optional
        Whether to draw a grid on the plot or not.
      verbose: 0 or 1, optional
        Whether to plot the density map or not.
      savePlot: boolean, optional
        Whether to save the plot or not.
      path: str, optional
        Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'

    Returns:
      2-D np.array of the shape (accuracy,accuracy).
  """
  dataSet_char = None
  if type(dataSet) == str:
    dataSet_char = dataSet
    dataSet = getDataSet(dataSet)

  if dataSet_char == None:
    dataSet_char = CURRENT_SET

  if specific_color != None:
    dataSet = dataSet.loc[dataSet['l_i'] == specific_color]
    dataSet.reset_index(inplace=True)

  density_map = np.zeros((accuracy, accuracy))

  printProgressBar(0, accuracy**2)
```

```
  for i in range(accuracy):
    for j in range(accuracy):
      count = dataSet.loc[(dataSet['x_i1'] - j/accuracy)**2 +
              (dataSet['x_i2'] - i/accuracy)**2 <= significance**2]

      density_map[i,j] = len(count)

      printProgressBar(i*accuracy + j + 1, accuracy**2)

  # Used for the normalization
  norm = plt.Normalize(vmin=np.min(density_map),vmax=np.max(density_map))

  # Plotting
  if verbose > 0:
    fig, ax = plt.subplots()

    if specific_color != None:
      ax.set_title(f'Density of {COLORS[specific_color]} in {dataSet_char}')
    else:
      ax.set_title(f'Total density in {dataSet_char}')

    plt.imshow(density_map, origin='lower', cmap='Spectral', norm=norm,
               extent=[0, 1, 0, 1])
    plt.colorbar()
    ax.set_xlabel('x_i1')
    ax.set_ylabel('x_i2')
    ax.set_xlim((0,1))
    ax.set_ylim((0,1))
    ax.set_xticks([i/10 for i in range(11)])
    ax.set_yticks([i/10 for i in range(11)])
    if drawGrid == True:
      ax.grid(alpha=0.3, color='black')
    plt.show()

  # Save plot
  if savePlot == True:
    fig.savefig(f'{path}DensityMap_{dataSet_char}.pdf')
    fig.savefig(f'{path}DensityMap_{dataSet_char}.png', dpi=300)

  return density_map



def plotDensity(dataSet=CURRENT_SET, significance=0.1,
                cmap=plt.cm.get_cmap('Spectral'), specific_color = None,
                drawGrid=True, savePlot=False, path=''):
  """Colorises and plots the points of dataSet according to their numbers
    of neighbors.

    Args:
      dataSet: pandas.DataFrame or char, optional
        Dataframe with columns 'x_i1', 'x_i2', 'l_i1' or char 'A', 'B', or 'C'.
        Dataset to be plotted.
```

```
    signifcance: float between 0 and 1, optional
      Determines the radius in which neighbours are being counted for the
      density of a particular point.
    cmap: matplotlib colormap, optional
      Is used for color coding the density of the dataset at the end.
    specific_color: 0 or 1, optional
      If 0, a heatmap of only green points is computed. If 1, analogously for
      red.
    drawGrid: boolean, optional
      Whether to draw a grid on the plot or not.
    savePlot: boolean, optional
      Whether to save the plot or not.
    path: str, optional
      Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'
"""
dataSet_char = None
if type(dataSet) == str:
  dataSet = getDataSet(dataSet)

if dataSet_char == None:
  dataSet_char = CURRENT_SET

if specific_color != None:
  dataSet = dataSet.loc[dataSet['l_i'] == specific_color]
  dataSet.reset_index(inplace=True)

total_number_of_points = dataSet.shape[0]
array = np.zeros((total_number_of_points, 3))

# Counting all neighbours within a radius of significance
for i in range(total_number_of_points):
  count = dataSet.loc[(dataSet['x_i1'] - dataSet['x_i1'].loc[i])**2 +
   (dataSet['x_i2'] - dataSet['x_i2'].loc[i])**2 <= significance**2]

  array[i, 0] = dataSet['x_i1'].loc[i]
  array[i, 1] = dataSet['x_i2'].loc[i]
  array[i, 2] = len(count)

  printProgressBar(i+1, total_number_of_points)

print(f'Max: {np.max(array[:,2])}')
print(f'Min: {np.min(array[:,2])}')

fig, ax = plt.subplots()

# Used for the normalization
norm = plt.Normalize(vmin=np.min(array[:,2]),vmax=np.max(array[:,2]))

# Setting parameters for ploting
plt.scatter(array[:, 0], array[:, 1], c=array[:, 2], cmap=cmap, norm=norm)

ax.set_title(f'Density of dataset {dataSet_char}')
plt.axis('scaled')
```

```
  ax.set_xlabel('x_i1')
  ax.set_ylabel('x_i2')
  ax.set_xlim((0,1))
  ax.set_ylim((0,1))
  ax.set_xticks([i/10 for i in range(11)])
  ax.set_yticks([i/10 for i in range(11)])
  if drawGrid == True:
    ax.grid(alpha=0.3, color='black')
  plt.colorbar()
  plt.show()

  # Save plot
  if savePlot == True:
    fig.savefig(f'{path}Density_{dataSet_char}.pdf')
    fig.savefig(f'{path}Density_{dataSet_char}.png', dpi=300)


def makeWeightedCertaintyMap(model, accuracy, significance=0.1,
                          useThresholdPredict=False, referenceMethod='even',
                          referenceValue=None, drawGrid=True, savePlot=False,
                          path=''):
  """Plots the model's prediction certainty weighted with the density of points
    given in the dataset.

  Args:
    model: keras model
      The model who's weighted prediction certainty is to be plotted.
    accuracy: int
      Data points are spaced 1/accuracy apart along the x and y axis. The grid
      of data points plotted has the dimension accuracy*accuracy.
    signifcance: float between 0 and 1, optional
        Determines the radius in which neighbours are being counted for the
        density of a particular point.
    useThresholdPredict: boolean, optional
      Whether to use thresholdPredict (True) or regular model.predict (False)
      when calculating the model's prediction certainty.
    referenceMethod: str: 'even', 'maxDensity', or 'customValue', optional
      The method used to calculate the reference density used. 'even'
      calculates the density if all points in dataset were evenly spaced.
      'maxDensity' uses the maximum density from densityMap as the reference
      ´density. 'customValue' uses a custom reference density.
    referenceValue: float between 0 and 1, optional
      Defines the custom reference density when using 'customValue' reference
      method. Leave blank otherwise.
    drawGrid: boolean, optional
        Whether to draw a grid on the plot or not.
    savePlot: boolean, optional
        Whether to save the plot or not.
    path: str, optional
      Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'

  Returns:
    2-D np.array of the shape (accuracy,accuracy).
```

```
Raises:
  TypeError: if invalid referenceMethod was given.

"""
if not referenceMethod in ['even', 'evenSqrt', 'evenLog', 'maxDensity',
                           'customValue']:
 raise TypeError(f'Invalid referenceMethod given. referenceMethod should be' +
                 f' "even", "evenSqrt", "evenLog", "maxDensity", or ' +
                 f'"customValue", but "{referenceMethod}" was given.')

dataSet = getDataSet()

print(f'Calculating certainty map:')
certaintyMap = makeCertaintyMap(model, accuracy, None, useThresholdPredict,
                                verbose=0)
clear_output()

print(f'Calculating density map:')
densityMap = makeDensityMap(accuracy, significance=significance, verbose=0)
clear_output()

if referenceMethod == 'even':
  totalPoints = SOURCE_SIZE[CURRENT_SET]
  neighborhoodArea = math.pi*(significance**2)
  evenDensity = totalPoints*neighborhoodArea
  densityMap = densityMap/evenDensity

elif referenceMethod == 'evenSqrt':
  densityMap = np.sqrt(densityMap)

  totalPoints = SOURCE_SIZE[CURRENT_SET]
  neighborhoodArea = math.pi*(significance**2)
  evenDensity = totalPoints*neighborhoodArea
  densityMap = densityMap/evenDensity

elif referenceMethod == 'evenLog':
  densityMap = np.log(densityMap+1)

  totalPoints = SOURCE_SIZE[CURRENT_SET]
  neighborhoodArea = math.pi*(significance**2)
  evenDensity = totalPoints*neighborhoodArea
  densityMap = densityMap/evenDensity

elif referenceMethod == 'maxDensity':
  maxDensity = np.max(densityMap)
  densityMap = densityMap/maxDensity

elif referenceMethod == 'customValue':
  densityMap = densityMap/referenceValue

weightedCertaintyMap = certaintyMap*densityMap
```

```
  fig, ax = plt.subplots()

  plt.imshow(weightedCertaintyMap, origin='lower', cmap='tab20b', vmin=0,
             vmax=np.max(weightedCertaintyMap), extent=[0, 1, 0, 1])

  ax.set_title(f'Weighted certainty of datset {CURRENT_SET}')
  ax.set_xlabel('x_i1')
  ax.set_ylabel('x_i2')
  ax.set_xticks([i/10 for i in range(11)])
  ax.set_yticks([i/10 for i in range(11)])
  if drawGrid == True:
    ax.grid(alpha=0.3, color='black')
  plt.colorbar()
  plt.show()

  # Save plot
  if savePlot == True:
    fig.savefig(f'{path}WeightedCertaintyMap_{CURRENT_SET}.pdf')
    fig.savefig(f'{path}WeightedCertaintyMap_{CURRENT_SET}.png', dpi=300)

  return weightedCertaintyMap



def makeDistributionMap(dataSet=CURRENT_SET, accuracy=10, colorbarLim=-1,
                        drawGrid=True, showTitle=False, savePlot=False, path=''):
  """Plots the distribution of dataSet.

  Args:
    dataSet: char, optional
      'A', 'B', or 'C'. Dataset who's distribution is to be plotted.
    accuracy: int, optional
      The distribution map is split up into accuracy*accuracy many fields.
    colorbarLim: float between 0 and 1, optional
      Upper limit for the colorbar. Defaults to -1 where the maximum
      distribution percentage is used as the upper limit.
    drawGrid: boolean, optional
      Whether to draw a grid on the plot or not.
    savePlot: boolean, optional
      Whether to save the plot or not.
    showTitle: boolean, optional
      Wether to show the title of the plot or not.
    path: str, optional
      Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'

  Returns:
    2-D np.array of the shape (accuracy,accuracy).
  """
  dataSet_char = dataSet
  dataSet = getDataSet(dataSet)

  distribution_map = np.zeros((accuracy, accuracy))
```

```
# Multiply all entries with accuracy to calculate which
# square each point falls into
dataSet = dataSet[['x_i1', 'x_i2']]*accuracy

printProgressBar(0, len(dataSet))

for i in range(len(dataSet)):
  x_i1 = math.floor(dataSet.loc[i]['x_i1'])
  x_i2 = math.floor(dataSet.loc[i]['x_i2'])

  # If x_i1 or x_i2 coordinate is 1.0, reduce by 1 to prevent index out of
  # bounds
  if x_i1 == accuracy:
    x_i1 = accuracy-1
  if x_i2 == accuracy:
    x_i2 = accuracy-1

  distribution_map[x_i2,x_i1] = distribution_map[x_i2,x_i1]+1

  printProgressBar(i+1, len(dataSet))

distribution_map = distribution_map/len(dataSet)

# Plotting
fig, ax = plt.subplots()
if showTitle:
  ax.set_title(f'Distribution of datset {dataSet_char}')


if colorbarLim == -1:
  colorbarLim = np.max(distribution_map)

plt.imshow(distribution_map, origin='lower', cmap='Spectral', vmin=0,
           vmax=colorbarLim, extent=[0, 1, 0, 1])


plt.colorbar()
ax.set_xlabel('x_i1')
ax.set_ylabel('x_i2')
ax.set_xlim((0,1))
ax.set_ylim((0,1))
ax.set_xticks([i/10 for i in range(11)])
ax.set_yticks([i/10 for i in range(11)])
if drawGrid == True:
  ax.grid(alpha=0.3, color='black')
plt.show()

# Save plot
if savePlot == True:
  fig.savefig(f'{path}DistributionMap_{dataSet_char}.pdf')
  fig.savefig(f'{path}DistributionMap_{dataSet_char}.png', dpi=300)

return distribution_map
```

[ ]:
```python
#@title Penalty Effect

def calculatePenaltyEffect(model, x, y, validation_data, interval=(0,1),
                           accuracy=10, batch_size=32, epochs=200, verbose=0):
  """Calculates red, green, and total misclassifications in relation to penalty.

  Args:
    model: keras model
      Model for which the penalty effect is measured.
    x: 2-D array of shape (x,2)
      Training points.
    y: 1-D array of shape (x,)
      Training labels.
    validation_data: 2-tuple
      (valSet_points, valSet_labels) where valSet_points is a 2-D array of shape
      (x,2) and valSet_labels a 1-D array of shape (x,). Validation points and
      labels.
    interval: 2-tuple, optional
      (x,y) which defines the penalty interval plotted. x is the lowest penalty,
      y the highest.
    accuracy: int, optional
      Penalty interval is evenly split into 'accuracy' many points.
    verbose: boolean, optional
      Whether to print progress bar and plot results or not.
    All others: optional
      See tf.keras.Model.

  Returns:
    3-tuple of int lists (total_misclass_percentage, red_misclass_percentage,
    green_misclass_percentage).
  """
  total_misclass_percentages = []
  red_misclass_percentages = []
  green_misclass_percentages = []
  penalties = np.zeros(accuracy + 1)
  increments = (interval[1]-interval[0])/accuracy

  points = validation_data[0]
  labels = validation_data[1].astype(int)

  number_of_points = len(labels)
  red_points = len(np.where(labels==1)[0])
  green_points = len(np.where(labels==0)[0])

  if verbose > 0:
    printProgressBar(0, accuracy+1)

  # MAIN LOOP
  for i in range(accuracy+1):
    penalty = interval[0] + (interval[1]-interval[0])*(i/accuracy)
    model.set_weights(initialWeights)

    model.compile(optimizer='adam', loss=construct_custom_penalty_loss(penalty),
```

```
                 metrics=['accuracy']) # Compile model with penalty

    history = model.fit(x, y, batch_size, epochs, verbose=0,
                        validation_data=validation_data)

    prediction = model.predict(validation_data[0])

    correct_indices = np.where((labels == np.argmax(prediction, axis=1)) == True)
    incorrect_indices = np.where((labels == np.argmax(prediction, axis=1)) == False)

    total_misclassifications = np.bincount(labels == np.argmax(prediction, axis=1))[0]
    red_misclassifications = len(np.where(labels[incorrect_indices] == 1)[0])
    green_misclassifications = len(np.where(labels[incorrect_indices] == 0)[0])

    total_misclass_percentages.append((total_misclassifications/number_of_points)*100)
    red_misclass_percentages.append((red_misclassifications/red_points)*100)
    green_misclass_percentages.append((green_misclassifications/green_points)*100)

    penalties[i] = penalty

    if verbose > 0:
      printProgressBar(i+1, iterations+1)

  # PLOTTING RESULTS
  if verbose > 0:
    plt.figure(figsize=(20,15))
    plt.plot(penalties, total_misclass_percentages, 'b', penalties,
             red_misclass_percentages, 'r', penalties, green_misclass_percentages,
             'g')
    plt.title(f'Dataset {CURRENT_SET}: Misclassification by penalty')
    plt.ylabel('% misclassified')
    plt.xlabel('Penalty')
    plt.xticks(np.arange(interval[0], interval[1]+increments, increments))
    plt.legend(['total', 'red', 'green'], loc='upper left')
    plt.show()

  return (total_misclass_percentages, red_misclass_percentages,
         green_misclass_percentages)


def averagePenaltyEffect(model, n, valSet_size, path='', interval=(0,1),
                         accuracy=10, batch_size=32, epochs=200, verbose=1,
                         useBalanceDataset=False):
  """Plots average penalty effect over n iterations.

  Args:
    model: keras model
      Model for which the penalty effect is measured.
    n: int
      Number of iterations the penalty effect is measured and averaged over.
    valSet_size: int
      Size of the validation set.
    path: str, optional
```

```
      Path to which the excel sheet will be saved. e.g. '/content/drive/MyDrive/'
    verbose: boolean, optional
      Whether to print progress bar or not.
    useBalanceDataset: boolean, optional
      Whether to balance the dataset before training or not.
    All others:
      See calculatePenaltyEffect.

  Returns:
    3-tuple of np arrays (total_misclass_percentages_avg,
    red_misclass_percentages_avg, green_misclass_percentages_avg).
  """
  #Start time
  start_time = time.time()

  penalties = np.arange(interval[0], interval[1]+(interval[1]-interval[0])/accuracy,
                        (interval[1]-interval[0])/accuracy)

  # INITIALIZATION OF DATA COLLECTION OBJECTS
  # For averaging
  total_misclass_percentages_collected = []
  red_misclass_percentages_collected = []
  green_misclass_percentages_collected = []
  # For saving in excel
  validation_points_collected = np.zeros((valSet_size, 3*n))
  misclassification_matrix = np.zeros((len(penalties), 3*n))
  # Column names
  val_columns = []
  coll_columns = []

  # Initialize progress bar
  if verbose > 0:
    printProgressBar(0, n)

  # MAIN LOOP
  for i in range(n):
    # PREPARING DATA
    dataSet = getDataSet()
    dataSet.pop('Unnamed: 0') #Removing unnessary column

    # Choose random validation set
    val_indices = getBalancedValSetIndices(dataSet, valSet_size, THRESHOLD_VAL)

    valSet_points, valSet_labels = separateValidationSet(dataSet=dataSet,
                                          validationIndices=val_indices)

    if useBalanceDataset:
      dataSet = balanceDataset(dataSet, threshold=THRESHOLD_DATA, verbose=0)

    training_labels = np.array(dataSet['l_i']).astype('float')
    training_points = np.array(dataSet[['x_i1','x_i2']])

    # Collecting misclassification percentages
```

```
    allPercentages = calculatePenaltyEffect(model, training_points, training_labels,
                                            (valSet_points, valSet_labels),
                                            interval=interval, accuracy=accuracy,
                                            batch_size=batch_size, epochs=epochs,
                                            verbose=0)

    total_misclass_percentages_collected.append(allPercentages[0])
    red_misclass_percentages_collected.append(allPercentages[1])
    green_misclass_percentages_collected.append(allPercentages[2])

    # Creating separate columns for validation set
    val_columns.append(f'x_i1:{i}')
    val_columns.append(f'x_i2:{i}')
    val_columns.append(f'l_i:{i}')

    for j in range(valSet_size):
      validation_points_collected[j,3*i + 0] = valSet_points[j, 0]
      validation_points_collected[j,3*i + 1] = valSet_points[j, 1]
      validation_points_collected[j,3*i + 2] = valSet_labels[j]

    # Creating seperarte columns for current misclassification
    coll_columns.append(f'total:{i}')
    coll_columns.append(f'red:{i}')
    coll_columns.append(f'green:{i}')

    misclassification_matrix[:, 3*i + 0] = allPercentages[0]
    misclassification_matrix[:, 3*i + 1] = allPercentages[1]
    misclassification_matrix[:, 3*i + 2] = allPercentages[2]

    if verbose > 0:
      printProgressBar(i+1, n)

  # Averaging
  total_misclass_percentages_avg = np.average(total_misclass_percentages_collected,␣
↪axis=0)
  red_misclass_percentages_avg = np.average(red_misclass_percentages_collected, axis=0)
  green_misclass_percentages_avg = np.average(green_misclass_percentages_collected,␣
↪axis=0)

  result = (total_misclass_percentages_avg, red_misclass_percentages_avg,
         green_misclass_percentages_avg)

  # PLOTTING RESULTS
  plotPenaltyEffect(model, data=result, interval=interval, accuracy=accuracy, n=n,
                    valSet_size=valSet_size, batch_size=batch_size, epochs=epochs,
                    path=path)

  # Print time taken for calculation
  end_time = time.time()
  total_time = (end_time-start_time)/60
  print(f'Time taken: {round(total_time, 2)} minutes.')

  # Save results to excel
```

```
today = date.today()

writer = pd.ExcelWriter(f'{path}Penalty_Data_{CURRENT_SET}_{model.name}_' +
                        f'{today.strftime("%d-%m-%Y")}.xlsx')

# Average misclass percentages
pd.DataFrame([total_misclass_percentages_avg, red_misclass_percentages_avg,
              green_misclass_percentages_avg], ['total','red','green'],
             columns=penalties).to_excel(writer, sheet_name=f'Average')

# Misclass percentages collected
pd.DataFrame(misclassification_matrix, penalties,
             columns=coll_columns).to_excel(writer, sheet_name=f'Collected')

# Parameters
data = {'Values':[f'{model.name}', f'{CURRENT_SET}', f'{n}', f'{valSet_size}',
                  f'{interval}', f'{accuracy}', f'{batch_size}', f'{epochs}',
                  f'{useBalanceDataset}']}

index = ['model','dataset','n','valSet_size','interval','accuracy',
         'batch_size','epochs','useBalanceDataset']

pd.DataFrame(data, index=index).to_excel(writer, sheet_name='Parameters')

# Validation sets
pd.DataFrame(validation_points_collected,
             columns=val_columns).to_excel(writer, sheet_name=f'Validation Sets')

writer.save()

return result


def plotPenaltyEffect(model, data, interval, accuracy, n, valSet_size, batch_size,␣
 ↪epochs,
                      dataset=CURRENT_SET, ylim=[0,10], maj_yt_incr=1,
                      min_yt_incr=0.1, figsize=(14,10), showParameters=True,
                      resolution=300, path=''):
  """Plots average penalty effect given by 'data' and saves png and pdf of plot
    to the directory.

  Args:
    model: keras model
      Model for which the penalty effect is measured.
    data: 3-tuple of np arrays, or str
      (total_misclass_percentages_avg, red_misclass_percentages_avg,
      green_misclass_percentages_avg) or the name of an Excel sheet present in
      the directory as a String (e.g. 'data.xlsx').
    interval: 2-tuple
      (x,y) which defines the penalty interval plotted. x is the lowest penalty,
      y the highest.
    accuracy: int
      Penalty interval is evenly split into 'accuracy' many points.
```

```
    n, valSet_size, batch_size, epochs:
      Parameters used for training and calculaing the average penalty effect.
      Shown in configurations text in plot.
    dataset: char, optional
      Dataset which the penalty effect was measured on. 'A', 'B' or 'C'.
    ylim: 1D list of floats or ints, optional
      [x,y] which defines the range of % misclassification shown on the y-axis.
    maj_yt_incr: float, optional
      The increments in which major y-ticks are plotted on the y-axis.
    min_yt_incr: float, optional
      The increments in which minor y-ticks are plotted on the y-axis.
    figsize: 2-tuple of floats, optional
      (x,y) where x is the width of the plot and y is the height of the plot.
    showParameters: boolean, optional
      Whether to include a configuratiuon text in the plot or not.
    resolution: int, optional
      Resolution of the plot png in dpi.
    path: str, optional
      Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'

  Raises:
    TypeError: if data is not of type String or 3-tuple of np arrays.
  """
  # Penalties to be plotted on the x-axis
  penalties = np.arange(interval[0], interval[1]+(interval[1]-interval[0])/accuracy,
                        (interval[1]-interval[0])/accuracy)

  # DATA PREPARATION
  if (isinstance(data, tuple) and isinstance(data[0], np.ndarray) and
      isinstance(data[1], np.ndarray) and isinstance(data[2], np.ndarray) and
      len(data)==3):
    total_misclass_percentages_avg = data[0]
    red_misclass_percentages_avg = data[1]
    green_misclass_percentages_avg = data [2]

  elif isinstance(data, str):
    data = pd.ExcelFile(data)
    avg_data = pd.read_excel(data, 'Average')

    total = pd.DataFrame(avg_data.loc[0])
    total = total.drop('Unnamed: 0')
    total_misclass_percentages_avg = total[0]

    red = pd.DataFrame(avg_data.loc[1])
    red = red.drop('Unnamed: 0')
    red_misclass_percentages_avg = red[1]

    green = pd.DataFrame(avg_data.loc[2])
    green = green.drop('Unnamed: 0')
    green_misclass_percentages_avg = green[2]

  else:
    raise TypeError(f'Invalid type of data. data should be of type String or '
```

```
                    + f'a 3-tuple of np arrays, but data is of type {type(data)}.')

  # Define yticks
  major_yticks = np.arange(0, ylim[1]+maj_yt_incr, maj_yt_incr)
  minor_yticks = np.arange(0, ylim[1]+min_yt_incr, min_yt_incr)

  # Create subplot
  fig, ax = plt.subplots(figsize=figsize)

  ax.plot(penalties, total_misclass_percentages_avg, 'b', penalties,
            red_misclass_percentages_avg, 'r', penalties,
            green_misclass_percentages_avg, 'g')

  ax.set_title(f'Dataset {dataset}: Average misclassification by penalty',
               fontsize='x-large')
  ax.set_ylabel('% misclassified', fontsize='large')
  ax.set_xlabel('Penalty', fontsize='large')

  # Ranges of x and y-axis
  ax.set_xlim(list(interval))
  ax.set_ylim(ylim)

  # Set ticks
  ax.set_xticks(penalties)
  ax.set_yticks(major_yticks)
  ax.set_yticks(minor_yticks, minor=True)

  # Color and grid
  ax.set_facecolor('white')
  ax.grid(which='minor', alpha=0.2, color='black')
  ax.grid(which='major', alpha=0.5, color='black')

  # Show configuration information on plot
  if showParameters==True:
    config_info = (f'{model.name}\nn: {n}\nVal. set size: {valSet_size}\n' +
                   f'Batch size: {batch_size}\nEpochs: {epochs}')
    ax.text(interval[1]+(interval[1]/(8*figsize[0])), ylim[1]-(ylim[1]/figsize[1]),
            config_info)

  plt.legend(['total', 'red', 'green'], loc='upper left', fontsize='medium')

  plt.show()

  # Get current date
  today = date.today()

  fig.savefig(f'{path}Penalty_Plt_{dataset}_{model.name}_' +
              f'{today.strftime("%d-%m-%Y")}.png', dpi=resolution)
  fig.savefig(f'{path}Penalty_Plt_{dataset}_{model.name}_' +
              f'{today.strftime("%d-%m-%Y")}.pdf')
```

[ ]:
```
#@title k-Nearest-Neighbour

def KNN(dataSet, point, k, significance=0.1, increment=0.05, show_plot=True):
```

```python
""" K-nearest neighbor classifier.

Statistical classifier. Uses the k nearest neighbors to predict the color of a
given point by comparing the number of neighbours of each color and weigthing
them with their squared distance to the point.

Args:
  dataSet: pandas.DataFrame, optional
    Dataframe with columns 'x_i1', 'x_i2', 'l_i1'. Dataset to be used for
    calculation. Defaults to dataset selected by CURRENT_SET.
  point: Array in the form of [x_i1, x_i2]
  k: Positive int
    Number of neighbours taken into account for classification
  significance: float between 0 and 1, optional
    Starting search radius.
  increment: float between 0 and 1, optional
    Amount of increment of the search radius while gathering k neighbours.
  show_plot: boolean, optional
    If 'True' the function plots the dataset and the selected neighbours.

Returns:
  A 2-tuple with the predictions for each class.
  (prediction_green, prediction_red)
"""
# Gathering points until at least k neighbours are found
neighb = np.array([])
while significance <= 1 and neighb.shape[0] < k:
    neighb = dataSet.loc[(dataSet['x_i1'] - point[0])**2 +
                     (dataSet['x_i2'] -point[1])**2 <= significance**2]
    significance += increment

# Reindexing
neighb = neighb.reset_index()

# Calculating the distances of each neighbour to the target point
dist = np.zeros(neighb.shape[0])
for i in range(neighb.shape[0]):
  dist[i] = (neighb['x_i1'].loc[i] - point[0])**2 + (neighb['x_i2'].loc[i] -
                                                     point[1])**2

# Removing all overhang neighbours until there are only k
while neighb.shape[0] > k:
  neighb = neighb.drop(np.argmax(dist))
  dist[np.argmax(dist)] = -1

# Reindexing
dist = np.zeros(neighb.shape[0])
neighb = neighb.reset_index()

# Calculating the distances of each neighbour to the target point
for i in range(neighb.shape[0]):
  dist[i] = (neighb['x_i1'].loc[i] - point[0])**2 + (neighb['x_i2'].loc[i] -
                                                     point[1])**2
```

```
  pred_g = 0
  pred_r = 0

  # Sum the neighbours of each color with the weight 1-dist^2
  for i in range(neighb.shape[0]):
    if neighb['l_i'].loc[i] == 0:
      pred_g += (1 - dist[i])
    elif neighb['l_i'].loc[i] == 1:
      pred_r += (1 - dist[i])

  # Normalize
  pred_g = pred_g / neighb.shape[0]
  pred_r = pred_r / neighb.shape[0]

  # Plot neighbours
  if show_plot:
    selected_neighb = [[neighb['x_i1'].loc[i], neighb['x_i2'].loc[i]]
                       for i in range(neighb.shape[0])]
    makePlot(dataSet, [point], selected_neighb)
    print(f'Prediction for green: \t{pred_g}')
    print(f'Prediction for red: \t{pred_r}')

  return (pred_g, pred_r)



def makeCertaintyMapKNN(k, accuracy = 100, specific_color = None):
  """Visualizes the prediction certainty of k-nearest-neighbour algorithm for a
    grid of data points.

  All data points have x and y values between 0 and 1.

  Args:
    k: postive int
      The number of neighbours specified for the KNN algorithm who's certainty
      is to bevisualized.
    accuracy: positive int, optional
      Data points are spaced 1/accuracy apart along the x and y axis. The grid
      of data points plotted has the dimension accuracy*accuracy.
    specific_color: 0 or 1, optional
      If 0, plots the model's certainty that a data point is green for all
      points in the grid. If 1, analogously for red.

  Raises:
    TypeError: If specific_color is not 'None', '0' or '1', or if accuracy and k
    is not an int.

  Returns:
    The certaintymap as an array with dimensions of (accuracy, accuracy)
  """
  #Exceptions
```

```python
if specific_color != None:
  if specific_color != 0 and specific_color != 1:
    raise TypeError(f'Invalid value for specific_color. Value is {specific_color}, \
      but should be "None", "0" or "1".')

if not isinstance(accuracy, int):
  raise TypeError(f'Invalid type for accuracy. Type is {type(accuracy)}, but \
    should be int.')

if not isinstance(k, int):
  raise TypeError(f'Invalid type for k. Type is {type(k)}, but \
    should be int.')

# Init Data
dataSet = getDataSet()
accuracy_map = np.zeros((accuracy, accuracy))

# Main Loop
for i in range(accuracy):
  for j in range(accuracy):
    result = KNN(dataSet, [j/accuracy, i/accuracy], k, show_plot=False)

    if specific_color != None:
      # Saving the prediction for the specified color
      accuracy_map[i,j] = result[specific_color]
    else:
      accuracy_map[i,j] = np.max(result)

    # Print current progress
    printProgressBar((j+1) + i*accuracy, accuracy**2)

# Choosing headline
if specific_color != None:
  plt.title(f'Certaintiy for {COLORS[specific_color]}')
  plt.imshow(accuracy_map, origin='lower', cmap='tab20b', vmin=0, vmax=1)

else:
  plt.title(f'General Certainty')
  plt.imshow(accuracy_map, origin='lower', cmap='tab20b', vmin=0.5, vmax=1)

# Plot
plt.colorbar()
plt.xlabel('x_i1')
plt.ylabel('x_i2')
plt.xticks([i for i in range(0, accuracy+1, accuracy//10)], [i/accuracy for i in␣
↪range(0, accuracy+1, accuracy//10)])
plt.yticks([i for i in range(0, accuracy+1, accuracy//10)], [i/accuracy for i in␣
↪range(0, accuracy+1, accuracy//10)])
plt.show()

return accuracy_map
```

[ ]:
```python
#@title Epoch Batch Size

def epochsBatchSize(model, initialWeights, valSet_size, batchRange,
                    batchIncrements, epochRange, epochIncrements, epsilon=0,
                    saveAndPlot=True, path='', verbose=1,
                    useBalanceDataset=False):
  """Calculates total, red, and green % misclassification in relation to batch
    size and epoch number for a random validation set on CURRENT_SET.

  Args:
    model: keras model
      Model which classifies the validation set.
    initialWeights: array-like
      Initial weights of model.
    valSet_size: int
      Size of the randomly chosen validation set.
    batchRange: 2-tuple of ints
      The range of batch sizes used. (x,y) where x is the smallest and y is the
      largest batch size used.
    batchIncrements: int
      Increment in which the batch size is increased.
    epochRange: 2-tuple of ints
      The range of epochs used. (x,y) where x is the smallest and y is the
      largest epoch number used.
    epochIncrements: int
      Increment in which the epoch number is increased.
    epsilon: float, optional
      The allowed absolute percentage difference between the misclass percentage
      of an optimum point and the minimum misclass percentage.
    saveAndPlot: boolean, optional
      Whever to save results to Excel and plot graphs or not. Set to false when
      using averageEpochsBatchSize.
    path: str, optional
      Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'
    verbose: boolean, optional
      Whether to print a progress bar or not.
    useBalanceDataset: boolean, optional
      Whether to balance the dataset before training or not.

  Returns:
    6-tuple (epochs, batch_sizes, total_misclass_percentage,
    red_misclass_percentage, green_misclass_percentage, valSet).
    First 5 elements are lists, valSet is pd.DataFrame.
  """
  # Start time
  if verbose > 0:
    start_time = time.time()

  # Preparing data collection lists
  epochs = []
  batch_sizes = []
  total_misclass_percentage = []
  red_misclass_percentage = []
```

```
green_misclass_percentage = []

# Defining iteration lists
batch_size_iter = np.arange(batchRange[0], batchRange[1]+1, batchIncrements)
epoch_iter = np.arange(epochRange[0], epochRange[1]+1, epochIncrements)

if batch_size_iter[0] == 0:
  batch_size_iter[0] = 1

# Preparing data
dataSet = getDataSet()
dataSet.pop('Unnamed: 0') #Removing unnessary column

# Choose random validation set
random.seed(time.time())
val_indices = getBalancedValSetIndices(dataSet, valSet_size, THRESHOLD_VAL)

valSet_points, valSet_labels = separateValidationSet(dataSet,val_indices)

if useBalanceDataset:
  dataSet = balanceDataset(dataSet, threshold=THRESHOLD_DATA, verbose=0)

training_labels = np.array(dataSet['l_i']).astype('float')
training_points = np.array(dataSet[['x_i1','x_i2']])

number_of_points = len(valSet_labels)
red_points = len(np.where(valSet_labels==1)[0])
green_points = len(np.where(valSet_labels==0)[0])

# Initialize progress bar
if verbose > 0:
  num_training_points = training_labels.shape[0]
  progress = 0
  full = 0
  # Calculate full progress
  for ep in epoch_iter:
    for ba in batch_size_iter:
      full += ep*math.ceil(num_training_points/ba)
  # Print bar
  printProgressBar(progress, full, suffix=f'{progress}/{full} steps')

# Epoch loop
for ep in epoch_iter:
  # Batch size loop
  for ba in batch_size_iter:
    epochs.append(ep)
    batch_sizes.append(ba)

    # Prepare model for classification
    model.set_weights(initialWeights)

    history = model.fit(x=training_points, y=training_labels, batch_size=ba,
                        epochs=ep, verbose=0)
```

```
    # Classification and saving results
    prediction = model.predict(valSet_points)

    correct_indices = np.where((valSet_labels == np.argmax(prediction, axis=1))
                               == True)
    incorrect_indices = np.where((valSet_labels == np.argmax(prediction, axis=1))
                                 == False)

    total_misclassifications = np.bincount(valSet_labels == np.argmax(prediction,␣
↪axis=1))[0]
    red_misclassifications = len(np.where(valSet_labels[incorrect_indices] == 1)[0])
    green_misclassifications = len(np.where(valSet_labels[incorrect_indices] ==␣
↪0)[0])

    total_misclass_percentage.append((total_misclassifications/number_of_points)*100)
    red_misclass_percentage.append((red_misclassifications/red_points)*100)
    green_misclass_percentage.append((green_misclassifications/green_points)*100)

    # Update progress bar
    if verbose > 0:
      progress += ep*math.ceil(num_training_points/ba)
      printProgressBar(progress, full, suffix=f'{progress}/{full} steps')

# Print time taken for calculation
if verbose > 0:
  end_time = time.time()
  total_time = (end_time-start_time)/60
  print(f'Time taken: {round(total_time, 2)} minutes.')

# Validation set
valSet = pd.DataFrame.from_dict({'x_i1':valSet_points[:,0],'x_i2':valSet_points[:,1],
                                'l_i':valSet_labels})

# Parameters
data = {'Values':[f'{model.name}', f'{CURRENT_SET}', f'{valSet_size}',
                  f'{PENALTY}', f'{batchRange}', f'{batchIncrements}',
                  f'{epochRange}', f'{epochIncrements}', f'{epsilon}',
                  f'{useBalanceDataset}']}

index = ['model','dataset','valSet_size','penalty','batchRange',
         'batchIncrements','epochRange','epochIncrements','epsilon',
         'useBalanceDataset']

parameters = pd.DataFrame(data, index=index)

# SAVE RESULTS TO EXCEL
if saveAndPlot==True:
  today = date.today()

  # Initialize writer
  writer = pd.ExcelWriter(f'{path}EBS_Data_{CURRENT_SET}_' +
                          f'{model.name}_{today.strftime("%d-%m-%Y")}.xlsx')
```

```
    # Create multiindex for epochs and batch_sizes
    arrays = [epochs,batch_sizes]

    tuples = list(zip(*arrays))

    multiindex = pd.MultiIndex.from_tuples(tuples,
                                    names=["epoch", "batch_size"])

    # All data
    allData = pd.DataFrame({'total':total_misclass_percentage,
                            'red':red_misclass_percentage,
                            'green':green_misclass_percentage}, index=multiindex)

    allData.to_excel(writer, sheet_name='All Data')

    # Optimum points
    result = (epochs, batch_sizes, total_misclass_percentage,
            red_misclass_percentage, green_misclass_percentage)
    optimumPoints = calculateOptimumPoints(result, epsilon)
    optimumPoints.to_excel(writer, sheet_name='Optimum Points')

    # Parameters
    parameters.to_excel(writer, sheet_name='Parameters')

    # Validation set
    valSet.to_excel(writer, sheet_name='Validation Set')

    writer.save()

  # Plot and return results
  result = (epochs, batch_sizes, total_misclass_percentage,
            red_misclass_percentage, green_misclass_percentage, valSet)

  if saveAndPlot==True:
    plotEpochsBatchSize(model, result, path=path)

  return result


def calculateOptimumPoints(data, epsilon):
  """Calculates optimum points of epoch and batch size for total, red, and green
    misclassification.

  Args:
    data: 5-tuple or str
      (epochs, batch_sizes, total_misclass_percentage, red_misclass_percentage,
      green_misclass_percentage) or the name of an Excel sheet present in the
      directory as a String (e.g. 'data.xlsx').
    epsilon: float
      The allowed absolute percentage difference between the misclass percentage
      of an optimum point and the minimum misclass percentage.
```

```python
    Returns: pd.DataFrame
      columns: [min_misclass, epsilon, opt_misclass, opt_epoch, opt_batch,
                t_misclass_here, r_misclass here, g_misclass_here]
      rows: [total, red, green]

    Raises:
      TypeError: if data is not in correct form.
    """
    # DATA PREPARATION
    if isinstance(data, tuple):
      # Converting data to list of np arrays
      data = list(data)
      for i in range(5):
        if isinstance(data[i], list):
          data[i] = np.array(data[i])

      # Checking list shapes
      for i in range(4):
        if data[i].shape != data[i+1].shape:
          raise TypeError(f'The elements of the 5-tuple data must all have the' +
                    f' same shape. The {i+1}. element has shape {data[i].shape}' +
                    f' and the {i+2}. element has shape {data[i+1].shape}.')

      if len(data[0].shape) != 1:
        raise TypeError(f'The elements of the 5-tuple data must all be' +
                    f' 1-dimensional.')

      epochs = data[0]
      batch_sizes = data[1]
      total_misclass_percentage = data[2]
      red_misclass_percentage = data[3]
      green_misclass_percentage = data[4]

    elif isinstance(data, str):
      data = pd.ExcelFile(data)
      data = pd.read_excel(data, 'All Data')

      # Check columns for equal length
      for col in list(data.columns):
        if data[col].isnull().values.any():
          raise TypeError(f'The columns of the excel sheet data are not of ' +
                    f'equal length. Column {col} contains NAN.')

      epochs = data['epoch']
      batch_sizes = data['batch_size']
      total_misclass_percentage = data['total']
      red_misclass_percentage = data['red']
      green_misclass_percentage = data['green']

    else:
      raise TypeError(f'data should be of type tuple or str, but is of type' +
                    f' {type(data)}.')
```

```
# CALCULATE OPTIMUM POINTS
# For total: Finds the configuration with total misclass within epsilon of
#   minimum total misclass which has the lowest red misclass.
# For green: Finds the configuration with green misclass within epsilon of
#   minimum green misclass which has the lowest red misclass.
# For red: Finds the configuration with red misclass within epsilon of
#   minimum red misclass which has the lowest total misclass.
columns = ['min_misclass', 'epsilon', 'opt_misclass', 'opt_epoch', 'opt_batch',
           't_misclass_here', 'r_misclass here', 'g_misclass_here']
rows = ['total', 'red', 'green']
t_considerable_indices = []
r_considerable_indices = []
g_considerable_indices = []

#Total
t_min = np.min(total_misclass_percentage)
t_opt = np.argmin(total_misclass_percentage)  # Index of optimum point for t
for index in range(len(total_misclass_percentage)):
  if total_misclass_percentage[index] <= (t_min+epsilon):
    t_considerable_indices.append(index)
for index in t_considerable_indices:  # Find point with lowest red misclass
  if red_misclass_percentage[index] < red_misclass_percentage[t_opt]:
    t_opt = index

#Green
g_min = np.min(green_misclass_percentage)
g_opt = np.argmin(green_misclass_percentage)  # Index of optimum point for g
for index in range(len(green_misclass_percentage)):
  if green_misclass_percentage[index] <= (g_min+epsilon):
    g_considerable_indices.append(index)
for index in g_considerable_indices:  # Find point with lowest red misclass
  if red_misclass_percentage[index] < red_misclass_percentage[g_opt]:
    g_opt = index

#Red
r_min = np.min(red_misclass_percentage)
r_opt = np.argmin(red_misclass_percentage)  # Index of optimum point for r
for index in range(len(red_misclass_percentage)):
  if red_misclass_percentage[index] <= (r_min+epsilon):
    r_considerable_indices.append(index)
# Find point with lowest total misclass
# Only change r_opt if the improvement in total misclass is greater than the
#   loss in red misclass
for index in r_considerable_indices:
  if (total_misclass_percentage[index] < total_misclass_percentage[r_opt] and
      (total_misclass_percentage[index]-total_misclass_percentage[r_opt] <
       red_misclass_percentage[r_opt]-red_misclass_percentage[index])):
    r_opt = index

total_row = [t_min, epsilon, total_misclass_percentage[t_opt], epochs[t_opt],
             batch_sizes[t_opt], total_misclass_percentage[t_opt],
             red_misclass_percentage[t_opt], green_misclass_percentage[t_opt]]
red_row = [r_min, epsilon, red_misclass_percentage[r_opt], epochs[r_opt],
```

```
            batch_sizes[r_opt], total_misclass_percentage[r_opt],
            red_misclass_percentage[r_opt], green_misclass_percentage[r_opt]]
  green_row = [g_min, epsilon, green_misclass_percentage[g_opt], epochs[g_opt],
            batch_sizes[g_opt], total_misclass_percentage[g_opt],
            red_misclass_percentage[g_opt], green_misclass_percentage[g_opt]]

  return pd.DataFrame([total_row, red_row, green_row], index=rows,
                   columns=columns)


def averageEpochsBatchSize(model, n, initialWeights, valSet_size, batchRange,
                  batchIncrements, epochRange, epochIncrements, epsilon=0,
                  path='', verbose=1, useBalanceDataset=False):
  """Calculates total, red, and green % misclassification in relation to batch
    size and epoch number averaged over n validation sets on CURRENT_SET.

  Args:
    n: int
      Number of iterations.
    All others:
      See epochsBatchSize.

  Returns:
    5 tuple of lists (epochs, batch_sizes, total_misclass_avg, red_misclass_avg,
    green_misclass_avg).
  """
  # Start time
  start_time = time.time()

  # Preparing data collection lists
  epochs_collected = []
  batch_sizes_collected = []
  total_misclass_collected = []
  red_misclass_collected = []
  green_misclass_collected = []

  # For saving in excel
  validationSets = {}
  misclassCollected = {}

  # Initialize progress bar
  if verbose > 0:
    printProgressBar(0, n)

  # MAIN LOOP
  for i in range(n):
    # Collecting misclassification percentages
    data = epochsBatchSize(model, initialWeights, valSet_size, batchRange,
                      batchIncrements, epochRange, epochIncrements,
                      epsilon=0, saveAndPlot=False, verbose=0,
                      useBalanceDataset=useBalanceDataset)

    epochs_collected.append(data[0])
```

```
    batch_sizes_collected.append(data[1])
    total_misclass_collected.append(data[2])
    red_misclass_collected.append(data[3])
    green_misclass_collected.append(data[4])

    # Adding validation set to dictionary for dataframe
    validationSets[f'x_i1:{i}'] = data[5]['x_i1']
    validationSets[f'x_i2:{i}'] = data[5]['x_i2']
    validationSets[f'l_i:{i}'] = data[5]['l_i']

    # Adding misclassification data to dictionary for dataframe
    misclassCollected[f'total:{i}'] = data[2]
    misclassCollected[f'red:{i}'] = data[3]
    misclassCollected[f'green{i}'] = data[4]

    # Update progress bar
    if verbose > 0:
      printProgressBar(i+1, n)


  # Averaging
  epochs = np.average(epochs_collected, axis=0)
  batch_sizes = np.average(batch_sizes_collected, axis=0)
  total_misclass_avg = np.average(total_misclass_collected, axis=0)
  red_misclass_avg = np.average(red_misclass_collected, axis=0)
  green_misclass_avg = np.average(green_misclass_collected, axis=0)


  # SAVE RESULTS TO EXCEL
  today = date.today()

  # Create multiindex for epochs and batch_sizes
  arrays = [epochs,batch_sizes]

  tuples = list(zip(*arrays))

  multiindex = pd.MultiIndex.from_tuples(tuples, names=["epoch", "batch_size"])

  # Initialize writer
  writer = pd.ExcelWriter(f'{path}Avg_EBS_Data_{CURRENT_SET}_' +
                        f'{model.name}_{today.strftime("%d-%m-%Y")}.xlsx')

  # Average
  average = pd.DataFrame({'total':total_misclass_avg,
                          'red':red_misclass_avg,
                          'green':green_misclass_avg}, index=multiindex)

  average.to_excel(writer, sheet_name='Average')

  # Collected
  misclassCollected = pd.DataFrame(misclassCollected, index=multiindex)
  misclassCollected.to_excel(writer, sheet_name='Collected')
```

```
  # Optimum points
  result = (epochs, batch_sizes, total_misclass_avg, red_misclass_avg,
            green_misclass_avg)
  optimumPoints = calculateOptimumPoints(result, epsilon)
  optimumPoints.to_excel(writer, sheet_name='Optimum Points')

  # Parameters
  data = {'Values':[f'{model.name}', f'{CURRENT_SET}', f'{n}', f'{valSet_size}',
                    f'{PENALTY}', f'{batchRange}', f'{batchIncrements}',
                    f'{epochRange}', f'{epochIncrements}', f'{epsilon}',
                    f'{useBalanceDataset}']}

  index = ['model','dataset','n','valSet_size','penalty','batchRange',
           'batchIncrements','epochRange','epochIncrements','epsilon',
           'useBalaceDataset']

  parameters = pd.DataFrame(data, index=index)
  parameters.to_excel(writer, sheet_name='Parameters')

  # Validation sets
  validationSets = pd.DataFrame.from_dict(validationSets)
  validationSets.to_excel(writer, sheet_name='Validation Sets')

  writer.save()


  # Plot and return results
  result = (epochs, batch_sizes, total_misclass_avg, red_misclass_avg,
            green_misclass_avg)

  plotEpochsBatchSize(model, result, path=path, prefix='Avg_')

  # Print time taken for calculation
  if verbose > 0:
    end_time = time.time()
    total_time = (end_time-start_time)/60
    print(f'Time taken: {round(total_time, 2)} minutes.')

  return result


def plotEpochsBatchSize(model, data, dataset=CURRENT_SET,
                        misclass_range=(0,15), figsize=(14,10), resolution=300,
                        cmap='viridis', path='', prefix=''):
  """Plots a 3D graph showing the relation between epoch number, batch size,
    and percentage misclassification.

  Args:
    model: keras model
      The model which was used for training and classification.
    data: 6-tuple or str
      (epochs, batch_sizes, total_misclass_percentage, red_misclass_percentage,
      green_misclass_percentage, valSet) or the name of an Excel
```

```
    sheet present in the directory as a String (e.g. 'data.xlsx').
  dataset: char, optional
    The dataset used. 'A', 'B' or 'C'.
  misclass_range: 2-tuple, optional
    The range of misclassification percentages plotted (limits of the z-axis).
  figsize: 2-tuple, optional
    (x,y) where x is the width of the plot and y is the height of the plot.
  resolution: int, optional
    Resolution of the plot png in dpi.
  cmap: Colormap, optional
    A colormap for the surface patches.
  path: str, optional
    Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'
  prefix: str, optional
    appended to the front of the pnd and pdf file names

Raises:
  TypeError: if data is of invalid type or shape.
"""
# DATA PREPARATION
if isinstance(data, tuple):
  # Converting data to list of np arrays
  data = list(data)
  for i in range(5):
    if isinstance(data[i], list):
      data[i] = np.array(data[i])

  # Checking list shapes
  for i in range(4):
    if data[i].shape != data[i+1].shape:
      raise TypeError(f'The first 5 elements of the tuple data must all ' +
                f'have the same shape. The {i+1}. element has shape ' +
                f'{data[i].shape} and the {i+2}. element has shape ' +
                f'{data[i+1].shape}.')

  if len(data[0].shape) != 1:
    raise TypeError(f'The first 5 elements of the tuple data must all be ' +
                    f'1-dimensional.')

  epochs = data[0]
  batch_sizes = data[1]
  total_misclass = data[2]
  red_misclass = data[3]
  green_misclass = data[4]

elif isinstance(data, str):
  data = pd.ExcelFile(data)
  data = pd.read_excel(data, sheet_name=0, index_col=[0,1])

  # Check columns for equal length
  for col in list(data.columns):
    if data[col].isnull().values.any():
      raise TypeError(f'The columns of the excel sheet data are not of ' +
```

```
                        f'equal length. Column {col} contains NAN.')

    index_list = list(data.index)
    index_len = len(index_list)

    epochs = []
    batch_sizes = []
    for i in range(index_len):
      epochs.append(index_list[i][0])
      batch_sizes.append(index_list[i][1])

    total_misclass = data['total']
    red_misclass = data['red']
    green_misclass = data['green']

  else:
    raise TypeError(f'data should be of type tuple or str, but is of type' +
                    f' {type(data)}.')

  # Plotting
  fig = plt.figure(figsize=(figsize[0], figsize[1]*3))
  # Total misclassification
  ax_t = fig.add_subplot(3, 1, 1, projection='3d')
  ax_t.plot_trisurf(epochs, batch_sizes, total_misclass, cmap=cmap)
  ax_t.set_title(f'Dataset {dataset}: Total misclassification by epoch and batch size')
  ax_t.set_xlabel('Epochs')
  ax_t.set_ylabel('Batch size')
  ax_t.set_zlabel('% misclassification')
  ax_t.set_zlim3d(misclass_range[0], misclass_range[1])
  # Red misclassification
  ax_r = fig.add_subplot(3, 1, 2, projection='3d')
  ax_r.plot_trisurf(epochs, batch_sizes, red_misclass, cmap=cmap)
  ax_r.set_title(f'Dataset {dataset}: Red misclassification by epoch and batch size')
  ax_r.set_xlabel('Epochs')
  ax_r.set_ylabel('Batch size')
  ax_r.set_zlabel('% misclassification')
  ax_r.set_zlim3d(misclass_range[0], misclass_range[1])
  # Green misclassification
  ax_g = fig.add_subplot(3, 1, 3, projection='3d')
  ax_g.plot_trisurf(epochs, batch_sizes, green_misclass, cmap=cmap)
  ax_g.set_title(f'Dataset {dataset}: Green misclassification by epoch and batch size')
  ax_g.set_xlabel('Epochs')
  ax_g.set_ylabel('Batch size')
  ax_g.set_zlabel('% misclassification')
  ax_g.set_zlim3d(misclass_range[0], misclass_range[1])

  plt.show()

  # Saving
  today = date.today()

  fig.savefig(f'{path}{prefix}EBS_Plt_{dataset}_{model.name}_' +
              f'{today.strftime("%d-%m-%Y")}.png', dpi=resolution)
```

```
fig.savefig(f'{path}{prefix}EBS_Plt_{dataset}_{model.name}_' +
            f'{today.strftime("%d-%m-%Y")}.pdf')
```

[ ]:
```
#@title Different Training Approaches

def getProportionOfMisclassification(model, val_data):
  ''' Helperfunction only for the abandoned custom training approaches

      Args:
        model: keras model
          Model to be tested
        val_data: two-tuple of the form (val_points, val_labels)

      Returns:
        The calculated proprotions of misclassifcation on the predictions of the
        model for the given validation data
  '''

  # Creating Numpy arrays from tensors
  points = val_data[0]
  labels = val_data[1].astype('float')

  # Counting number of points for each class
  number_of_points = len(labels)
  red_points = len(np.where(labels==1)[0])
  green_points = len(np.where(labels==0)[0])

  prediction = model.predict(val_data[0])

  # Determining the incorrect predictions
  incorrect_indices = np.where((labels == np.argmax(prediction, axis=1)) == False)

  # Counting the number of misclassifications
  total_misclassifications = np.bincount(labels == np.argmax(prediction, axis=1))[0]
  red_misclassifications = len(np.where(labels[incorrect_indices] == 1)[0])
  green_misclassifications = len(np.where(labels[incorrect_indices] == 0)[0])

  return ((total_misclassifications/number_of_points)*100,
          (red_misclassifications/red_points)*100,
          (green_misclassifications/green_points)*100)


def penaltyIncreasingTraining(model, penalty, epochs, batch_size, increment,␣
 ↪epoch_end_of_inc, training_points, training_labels, increasing=True, verbose=0):
  ''' Trains the modell with an eihter increasing or decreasing penalty value

  Args:
    model: keras model
      Model to be trained
    penalty: float between 0 and 1
      penalty
    epochs: int
    batch_size: int
    increment: int
```

```
      The penalty values is updatet every 'increment' epochs
    epoch_end_of_inc: int, <= epochs
      Indicatess the end of incresing/decreasing training.
    training_points: numpy.array
      An array of shape (x,2) containg the points to be trained on
    training_labels: numpy.array
      An array of shape (x,1) with the corresponding labels
    increasing: boolean, optional
      Weather the penalty should be increased during training or decreased
    verbose: int, optional
      Sets the verbosity of the keras fitting process

  '''
  if increasing:
    array_penalties = np.linspace(0, penalty, (epochs - epoch_end_of_inc) // increment)
  else:
    array_penalties = np.linspace(penalty, 0, (epochs - epoch_end_of_inc) // increment)

  for i in range((epochs - epoch_end_of_inc) // increment):
    model.compile(optimizer='adam',␣
↪loss=construct_custom_penalty_loss(array_penalties[i]), metrics=['accuracy'])

    model.fit(training_points, training_labels, batch_size=batch_size,␣
↪epochs=increment,
                shuffle=True, verbose=verbose)

  model.fit(training_points, training_labels, batch_size=batch_size, epochs=epochs -␣
↪epoch_end_of_inc,
                shuffle=True, verbose=verbose)


def diffPenaltyAproach(n, val_size, model, penalty, epochs, batch_size, increment,
                       epoch_end_of_inc, verbose=0, figsize=(14,10), path='',
                                                       useBalanceDataset = False):
  '''Plots the average misclassification of each class for penalty increasing,
  consistent penalty and penalty decreasing fitting in a bar graph.

  Args:
    n: int
      Number of cyclces for averaging
    model: keras model
      Model which classifies the validation set.
    initialWeights: array-like
      Initial weights of model.
    penalty: float between 0 and 1
      Penalty to be added to the loss of misclassified red points during fitting
    valSet_size: int
      Size of the randomly chosen validation set.
    epochs: int
      Number of epochs for each training cycle
    batch_size: int
      Batch size to be used for training
    epoch_increment: int
```

```
      Amount of epochs after the penalty should be incremented
    epoch_end_of_inc: int
      Defines the point at which the increasing or decreasing stops and the modell
      will train with a consistent penalty value for the rest of the epochs
    verbose: boolean, optional
      Whether to print a progress bar or not.
    figsize: 2-tuple of int, optional
      Determines the size of the figure
    path: str, optional
      Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'
  '''
  dataSet_original = getDataSet()
  valSets = [getBalancedValSetIndices(dataSet_original, val_size, THRESHOLD_VAL) for i␣
↪in range(n)]

  history_1 = np.zeros((n,3))
  history_2 = np.zeros((n,3))
  history_3 = np.zeros((n,3))

  printProgressBar(0, 3*n)

  model.set_weights(initialWeights)
  for i in range(n):
    dataSet = dataSet_original.copy()
    model.set_weights(initialWeights)

    val_data = separateValidationSet(dataSet, valSets[i])

    if useBalanceDataset:
      dataSet = balanceDataset(dataSet, threshold=THRESHOLD_DATA, verbose=0)

    training_labels = np.array(dataSet['l_i']).astype(float)
    training_points = np.array(dataSet[['x_i1','x_i2']])

    penaltyIncreasingTraining(model, penalty, epochs, batch_size, increment,␣
↪epoch_end_of_inc, training_points, training_labels)

    history_1[i] = getProportionOfMisclassification(model, val_data)
    printProgressBar(i+1, 3*n)

  for i in range(n):
    dataSet = dataSet_original.copy()
    model.set_weights(initialWeights)

    val_data = separateValidationSet(dataSet, valSets[i])

    if useBalanceDataset:
      dataSet = balanceDataset(dataSet, threshold=THRESHOLD_DATA, verbose=0)

    training_labels = np.array(dataSet['l_i']).astype(float)
    training_points = np.array(dataSet[['x_i1','x_i2']])

    model.fit(training_points, training_labels, epochs=epochs,
```

```
                              batch_size=batch_size, shuffle=True, verbose=0)

    history_2[i] = getProportionOfMisclassification(model, val_data)
    printProgressBar(i + n+1, 3*n)

  for i in range(n):
    dataSet = dataSet_original.copy()
    model.set_weights(initialWeights)

    val_data = separateValidationSet(dataSet, valSets[i])

    if useBalanceDataset:
      dataSet = balanceDataset(dataSet, threshold=THRESHOLD_DATA, verbose=0)

    training_labels = np.array(dataSet['l_i']).astype(float)
    training_points = np.array(dataSet[['x_i1','x_i2']])

    penaltyIncreasingTraining(model, penalty, epochs, batch_size, increment,␣
↪epoch_end_of_inc, training_points, training_labels, increasing=False)

    history_3[i] = getProportionOfMisclassification(model, val_data)
    printProgressBar(i + 2*n+1, 3*n)

  clear_output()

  labels = ['total', 'red', 'green']
  y_1 = [i/n for i in np.sum(history_1, axis=0)]
  y_2 = [i/n for i in np.sum(history_2, axis=0)]
  y_3 = [i/n for i in np.sum(history_3, axis=0)]

  x = np.arange(len(labels))  # the label locations
  width = 0.2  # the width of the bars

  fig, ax = plt.subplots(figsize=figsize)
  rects1 = ax.bar(x - width, y_1, width, label='With Increment')
  rects2 = ax.bar(x, y_2, width, label='Normal')
  rects3 = ax.bar(x + width, y_3, width, label='With Decrement')

  # Add some text for labels, title and custom x-axis tick labels, etc.
  ax.set_ylabel('Misclassification in %')
  ax.set_xticks(x)
  ax.set_xticklabels(labels)
  ax.legend()

  fig.text(0,0, f'Dataset: {CURRENT_SET}, Epochs: {epochs}, Batch Size: {batch_size},␣
↪Epoch Increment: {increment}, Epoch end of Increment: {epoch_end_of_inc}')
  fig.tight_layout()
  plt.show()

  # Saving
  today = date.today()

  fig.savefig(f'{path}Comparison{CURRENT_SET}_{model.name}_' +
```

```
            f'{today.strftime("%d-%m-%Y")}.pdf')

    # Save results to excel
  today = date.today()

  writer = pd.ExcelWriter(f'{path}DiffPenTrain_Data_{CURRENT_SET}_' +
                          f'{model.name}_{today.strftime("%d-%m-%Y")}.xlsx')

  # Average misclass percentages
  pd.DataFrame([y_1, y_2, y_3], ['total','red','green'], columns=['Increment',␣
↪'Normal', 'Decrement']).to_excel(writer, sheet_name=f'Average')

  data = []
  data_names = []
  for i in range(n):
    data.append(history_1[i])
    data.append(history_2[i])
    data.append(history_3[i])
    data_names.append(f'Increment {i}:')
    data_names.append(f'Normal {i}:')
    data_names.append(f'Decrement {i}:')


  # Misclass percentages collected
  pd.DataFrame(data, data_names, columns=['total','red','green']).to_excel(writer,␣
↪sheet_name=f'Collected')

  # Parameters
  data = {'Values':[f'{model.name}', f'{CURRENT_SET}', f'{n}', f'{val_size}',
                    f'{PENALTY}', f'{epoch_end_of_inc}', f'{increment}',␣
↪f'{batch_size}',
                    f'{epochs}', f'{useBalanceDataset}']}

  index = ['model','dataset','n','valSet_size','penalty','epoch_end_of_inc',␣
↪'Increment',
           'batch_size','epochs','useBalanceDataset']

  pd.DataFrame(data, index=index).to_excel(writer, sheet_name='Parameters')

  # Validation sets
  pd.DataFrame(valSets).to_excel(writer, sheet_name=f'Validation Set Indices')

  writer.save()
```

[ ]:
```
#@title Custom Loss Function

def construct_custom_penalty_loss(penalty,
                                  lossFunction=keras.losses.
↪sparse_categorical_crossentropy):
  """Constructs a loss function which penalizes 'red as green' misclassifications.

  Args:
    penalty: float between 0 and 1
```

```
      Value added to the loss if a green point is misclassified as red.
    lossFunction: loss function, optional
      The loss function used after adapting the loss values.

  Returns:
    custom_penalty_loss function with specified penalty and loss function. Can be
    used like a regular loss function.
  """

  def custom_penalty_loss(y_true, y_pred):
    length = tf.shape(y_true)[0]

    #Creating a vector with all values set to the penalty: [0.3, 0.3, ... 0.3]
    error = tf.multiply(tf.constant(penalty, tf.float32), tf.ones(length))

    #Setting every entry to 0 if the corresponding entry in y_true is 1
    error = tf.where(tf.equal(y_true[:, 0], tf.zeros(length)), error, tf.zeros(length))

    #Setting every entry to 0 if the algorithm predicted 0
    error = tf.where(tf.greater(y_pred[:, 0], y_pred[:, 1]), tf.zeros(length), error)

    #Transforms the vector from [0, 0, 0.3, ... 0,3] to [[0, -0], [0, -0], [0.3, -0.
↪3], ... [0.3, -0.3]]
    error = tf.stack([error, tf.multiply(tf.constant(-1, tf.float32), error)], 1)

    #Adding the artificial loss
    y_pred = y_pred + error

    #Eliminating values > 1 or < 0
    y_pred0 = tf.where(tf.greater(y_pred[:, 0], tf.ones(length)), tf.ones(length),␣
↪y_pred[:, 0])
    y_pred1 = tf.where(tf.greater(y_pred[:, 1], tf.zeros(length)), y_pred[:, 1], tf.
↪zeros(length))
    y_pred = tf.stack([y_pred0, y_pred1], axis=1)


    loss = lossFunction(y_pred=y_pred, y_true=y_true)
    return loss

  return custom_penalty_loss
```

[ ]:

```
#@title Balance Effect

def calculateBalanceEffect(model, dataset, validation_data, interval=(0.8,1),
                           accuracy=10, batch_size=64, epochs=500, verbose=0):
  """Calculates red, green, and total misclassifications in relation to the
    treshold for balancing the model

  Args:
    model: keras model
      Model for which the balance threshold effect is measured.
    dataset: pandas.DataFrame
      Dataframe with columns 'x_i1', 'x_i2', 'l_i1'.
```

```
    validation_data: 2-tuple
      (valSet_points, valSet_labels) where valSet_points is a 2-D array of shape
      (x,2) and valSet_labels a 1-D array of shape (x,). Validation points and
      labels.
    interval: 2-tuple, optional
      (x,y) which defines the threshold interval plotted. x is the lowest
      threshold, y the highest.
    accuracy: int, optional
      Threshold interval is evenly split into 'accuracy' many points.
    epoch: According to its name
    batch_size: According to its name
    verbose: boolean, optional
      Whether to print progress bar and plot results or not.
    All others: optional
      See tf.keras.Model.

  Returns:
    3-tuple of int lists (total_misclass_percentage, red_misclass_percentage,
    green_misclass_percentage).
  """
  total_misclass_percentages = []
  red_misclass_percentages = []
  green_misclass_percentages = []
  thresholds = np.zeros(accuracy + 1)
  increments = (interval[1]-interval[0])/accuracy

  points = validation_data[0]
  labels = validation_data[1].astype(int)

  number_of_points = len(labels)
  red_points = len(np.where(labels==1)[0])
  green_points = len(np.where(labels==0)[0])

  # Initialize and fit model
  model.set_weights(initialWeights)

  model.compile(optimizer='adam', loss=construct_custom_penalty_loss(PENALTY),
                metrics=['accuracy']) # Compile model with penalty



  if verbose > 0:
    printProgressBar(0, accuracy+1)

  # MAIN LOOP
  for i in range(accuracy+1):
    threshold = interval[0] + (interval[1]-interval[0])*(i/accuracy)

    current_data = dataset.copy()
    current_data = balanceDataset(current_data, threshold, verbose=0)

    y = np.array(current_data['l_i']).astype('float')
    x = np.array(current_data[['x_i1','x_i2']])
```

```
    model.set_weights(initialWeights)
    history = model.fit(x, y, batch_size, epochs, verbose=0,
                        validation_data=validation_data)

    prediction = model.predict(validation_data[0])

    correct_indices = np.where((labels == np.argmax(prediction, axis=1)) == True)
    incorrect_indices = np.where((labels == np.argmax(prediction, axis=1)) == False)

    total_misclassifications = np.bincount(labels == np.argmax(prediction,axis=1))[0]
    red_misclassifications = len(np.where(labels[incorrect_indices] == 1)[0])
    green_misclassifications = len(np.where(labels[incorrect_indices] == 0)[0])

    total_misclass_percentages.append((total_misclassifications/number_of_points)*100)
    red_misclass_percentages.append((red_misclassifications/red_points)*100)
    green_misclass_percentages.append((green_misclassifications/green_points)*100)

    thresholds[i] = threshold

    if verbose > 0:
      printProgressBar(i+1, accuracy+1)

  # PLOTTING RESULTS
  if verbose > 0:
    plt.figure(figsize=(20,15))
    plt.plot(thresholds, total_misclass_percentages, 'b', thresholds,
             red_misclass_percentages, 'r', thresholds, green_misclass_percentages,
             'g')
    plt.title(f'Dataset {CURRENT_SET}: Misclassification by balance threshold')
    plt.ylabel('% misclassified')
    plt.xlabel('Balance threshold')
    plt.xticks(np.arange(interval[0], interval[1]+increments, increments))
    plt.legend(['total', 'red', 'green'], loc='upper left')
    plt.show()

  return (total_misclass_percentages, red_misclass_percentages,
         green_misclass_percentages)


def averageBalanceEffect(model, n, valSet_size, path='', interval=(0.8,1),
                         accuracy=10, batch_size=64, epochs=500, verbose=1):
  """Plots average balance effect over n iterations.

  Args:
    model: keras model
      Model for which the balance threshold effect is measured.
    n: int
      Number of iterations the balance threshold effect is measured and
      averaged over.
    valSet_size: int
      Size of the validation set.
    path: str, optional
```

```
    Path to which the excel sheet will be saved. e.g. '/content/drive/MyDrive/'
  epoch: According to its name
  batch_size: According to its name
  verbose: boolean, optional
    Whether to print progress bar or not.
  useBalanceDataset: boolean, optional
    Whether to balance the dataset before training or not.
  All others:
    See calculateBalanceEffect.

Returns:
  3-tuple of np arrays (total_misclass_percentages_avg,
  red_misclass_percentages_avg, green_misclass_percentages_avg).
"""
#Start time
start_time = time.time()

thresholds = np.arange(interval[0], interval[1]+(interval[1]-interval[0])/accuracy,
                      (interval[1]-interval[0])/accuracy)

# INITIALIZATION OF DATA COLLECTION OBJECTS
# For averaging
total_misclass_percentages_collected = []
red_misclass_percentages_collected = []
green_misclass_percentages_collected = []
# For saving in excel
validation_points_collected = np.zeros((valSet_size, 3*n))
misclassification_matrix = np.zeros((len(thresholds), 3*n))
# Column names
val_columns = []
coll_columns = []

# Initialize progress bar
if verbose > 0:
  printProgressBar(0, n)

# MAIN LOOP
for i in range(n):
  # PREPARING DATA
  dataSet = getDataSet()
  dataSet.pop('Unnamed: 0') #Removing unnessary column

  # Choose random validation set
  val_indices = random.sample(range(SOURCE_SIZE[CURRENT_SET]), valSet_size)

  valSet_points, valSet_labels = separateValidationSet(dataSet=dataSet,
                                          validationIndices=val_indices)

  # Collecting misclassification percentages
  allPercentages = calculateBalanceEffect(model, dataSet,
                                          (valSet_points, valSet_labels),
                                          interval=interval, accuracy=accuracy,
                                          batch_size=batch_size, epochs=epochs,
```

```
                                      verbose=0)

    total_misclass_percentages_collected.append(allPercentages[0])
    red_misclass_percentages_collected.append(allPercentages[1])
    green_misclass_percentages_collected.append(allPercentages[2])

    # Creating separate columns for validation set
    val_columns.append(f'x_i1:{i}')
    val_columns.append(f'x_i2:{i}')
    val_columns.append(f'l_i:{i}')

    for j in range(valSet_size):
      validation_points_collected[j,3*i + 0] = valSet_points[j, 0]
      validation_points_collected[j,3*i + 1] = valSet_points[j, 1]
      validation_points_collected[j,3*i + 2] = valSet_labels[j]

    # Creating seperarte columns for current misclassification
    coll_columns.append(f'total:{i}')
    coll_columns.append(f'red:{i}')
    coll_columns.append(f'green:{i}')

    misclassification_matrix[:, 3*i + 0] = allPercentages[0]
    misclassification_matrix[:, 3*i + 1] = allPercentages[1]
    misclassification_matrix[:, 3*i + 2] = allPercentages[2]

    if verbose > 0:
      printProgressBar(i+1, n)

  # Averaging
  total_misclass_percentages_avg = np.average(total_misclass_percentages_collected,␣
↪axis=0)
  red_misclass_percentages_avg = np.average(red_misclass_percentages_collected, axis=0)
  green_misclass_percentages_avg = np.average(green_misclass_percentages_collected,␣
↪axis=0)

  result = (total_misclass_percentages_avg, red_misclass_percentages_avg,
            green_misclass_percentages_avg)

  # PLOTTING RESULTS
  plotBalanceEffect(model, data=result, interval=interval, accuracy=accuracy,
                    n=n, valSet_size=valSet_size, batch_size=batch_size,
                    epochs=epochs, path=path)

  # Print time taken for calculation
  end_time = time.time()
  total_time = (end_time-start_time)/60
  print(f'Time taken: {round(total_time, 2)} minutes.')

  # Save results to excel
  today = date.today()

  writer = pd.ExcelWriter(f'{path}BalanceThreshold_Data_{CURRENT_SET}_' +
                          f'{model.name}_{today.strftime("%d-%m-%Y")}.xlsx')
```

```python
    # Average misclass percentages
    pd.DataFrame([total_misclass_percentages_avg, red_misclass_percentages_avg,
                  green_misclass_percentages_avg], ['total','red','green'],
                 columns=thresholds).to_excel(writer, sheet_name=f'Average')

    # Misclass percentages collected
    pd.DataFrame(misclassification_matrix, thresholds,
                 columns=coll_columns).to_excel(writer, sheet_name=f'Collected')

    # Parameters
    data = {'Values':[f'{model.name}', f'{CURRENT_SET}', f'{n}', f'{valSet_size}',
                      f'{PENALTY}', f'{interval}', f'{accuracy}', f'{batch_size}',
                      f'{epochs}']}

    index = ['model','dataset','n','valSet_size','penalty','interval','accuracy',
             'batch_size','epochs']

    pd.DataFrame(data, index=index).to_excel(writer, sheet_name='Parameters')

    # Validation sets
    pd.DataFrame(validation_points_collected,
                 columns=val_columns).to_excel(writer, sheet_name=f'Validation Sets')

    writer.save()

    return result


def plotBalanceEffect(model, data, interval, accuracy, n, valSet_size,
                      batch_size, epochs, penalty=PENALTY, dataset=CURRENT_SET,
                      ylim=[0,10], maj_yt_incr=1, min_yt_incr=0.1,
                      figsize=(14,10), showParameters=True, resolution=300,
                      path=''):
    """Plots average balance threshold effect given by 'data' and saves png and
      pdf of plot to the directory.

    Args:
      model: keras model
        Model for which the balance threshold effect is measured.
      data: 3-tuple of np arrays, or str
        (total_misclass_percentages_avg, red_misclass_percentages_avg,
        green_misclass_percentages_avg) or the name of an Excel sheet present in
        the directory as a String (e.g. 'data.xlsx').
      interval: 2-tuple
        (x,y) which defines the balance threshold interval plotted. x is the
        lowest penalty, y the highest.
      accuracy: int
        Balance threshold interval is evenly split into 'accuracy' many points.
      n, valSet_size, batch_size, epochs, penalty:
        Parameters used for training and calculaing the average balance
        threshold effect. Shown in configurations text in plot.
      dataset: char, optional
```

```
    Dataset which the balance penalty effect was measured on. 'A', 'B' or
    'C'.
  ylim: 1D list of floats or ints, optional
    [x,y] which defines the range of % misclassification shown on the y-axis.
  maj_yt_incr: float, optional
    The increments in which major y-ticks are plotted on the y-axis.
  min_yt_incr: float, optional
    The increments in which minor y-ticks are plotted on the y-axis.
  figsize: 2-tuple of floats, optional
    (x,y) where x is the width of the plot and y is the height of the plot.
  showParameters: boolean, optional
    Whether to include a configuratiuon text in the plot or not.
  resolution: int, optional
    Resolution of the plot png in dpi.
  path: str, optional
    Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'

Raises:
  TypeError: if data is not of type String or 3-tuple of np arrays.
"""
# Thresholds to be plotted on the x-axis
thresholds = np.arange(interval[0], interval[1]+(interval[1]-interval[0])/accuracy,
                        (interval[1]-interval[0])/accuracy)

# DATA PREPARATION
if (isinstance(data, tuple) and isinstance(data[0], np.ndarray) and
    isinstance(data[1], np.ndarray) and isinstance(data[2], np.ndarray) and
    len(data)==3):
  total_misclass_percentages_avg = data[0]
  red_misclass_percentages_avg = data[1]
  green_misclass_percentages_avg = data [2]

elif isinstance(data, str):
  data = pd.ExcelFile(data)
  avg_data = pd.read_excel(data, 'Average')

  total = pd.DataFrame(avg_data.loc[0])
  total = total.drop('Unnamed: 0')
  total_misclass_percentages_avg = total[0]

  red = pd.DataFrame(avg_data.loc[1])
  red = red.drop('Unnamed: 0')
  red_misclass_percentages_avg = red[1]

  green = pd.DataFrame(avg_data.loc[2])
  green = green.drop('Unnamed: 0')
  green_misclass_percentages_avg = green[2]

else:
  raise TypeError(f'Invalid type of data. data should be of type String or '
                  + f'a 3-tuple of np arrays, but data is of type {type(data)}.')

# Define yticks
```

```
  major_yticks = np.arange(0, ylim[1]+maj_yt_incr, maj_yt_incr)
  minor_yticks = np.arange(0, ylim[1]+min_yt_incr, min_yt_incr)

  # Create subplot
  fig, ax = plt.subplots(figsize=figsize)

  ax.plot(thresholds, total_misclass_percentages_avg, 'b', thresholds,
          red_misclass_percentages_avg, 'r', thresholds,
          green_misclass_percentages_avg, 'g')

  ax.set_title(f'Dataset {dataset}: Average misclassification by balance threshold',
               fontsize='x-large')
  ax.set_ylabel('% misclassified', fontsize='large')
  ax.set_xlabel('Balance threshold', fontsize='large')

  # Ranges of x and y-axis
  ax.set_xlim(list(interval))
  ax.set_ylim(ylim)

  # Set ticks
  ax.set_xticks(thresholds)
  ax.set_yticks(major_yticks)
  ax.set_yticks(minor_yticks, minor=True)

  # Color and grid
  ax.set_facecolor('white')
  ax.grid(which='minor', alpha=0.2, color='black')
  ax.grid(which='major', alpha=0.5, color='black')

  # Show configuration information on plot
  if showParameters==True:
    config_info = (f'{model.name}\nn: {n}\nVal. set size: {valSet_size}\n' +
                   f'Batch size: {batch_size}\nEpochs: {epochs}\n' +
                   f'Penalty: {penalty}')
    ax.text(interval[1]+(interval[1]/(8*figsize[0])), ylim[1]-(ylim[1]/figsize[1]),
            config_info)

  plt.legend(['total', 'red', 'green'], loc='upper left', fontsize='medium')

  plt.show()

  # Get current date
  today = date.today()

  fig.savefig(f'{path}BalanceEffect_Plt_{dataset}_{model.name}_' +
              f'{today.strftime("%d-%m-%Y")}.png', dpi=resolution)
  fig.savefig(f'{path}BalanceEffect_Plt_{dataset}_{model.name}_' +
              f'{today.strftime("%d-%m-%Y")}.pdf')
```

[ ]:
```
#@title Certainty Threshold Effect

def calculateThresholdEffect(model, x, y, validation_data, interval=(0.8,1),
                             accuracy=10, batch_size=64, epochs=500, verbose=0):
  """Calculates red, green, and total misclassifications in relation to the
```

```
    certainty threshold for predicting points as green.

  Args:
    model: keras model
      Model for which the certainty threshold effect is measured.
    x: 2-D array of shape (x,2)
      Training points.
    y: 1-D array of shape (x,)
      Training labels.
    validation_data: 2-tuple
      (valSet_points, valSet_labels) where valSet_points is a 2-D array of shape
      (x,2) and valSet_labels a 1-D array of shape (x,). Validation points and
      labels.
    interval: 2-tuple, optional
      (x,y) which defines the threshold interval plotted. x is the lowest
      threshold, y the highest.
    accuracy: int, optional
      Threshold interval is evenly split into 'accuracy' many points.
    verbose: boolean, optional
      Whether to print progress bar and plot results or not.
    All others: optional
      See tf.keras.Model.

  Returns:
    3-tuple of int lists (total_misclass_percentage, red_misclass_percentage,
    green_misclass_percentage).
  """
  total_misclass_percentages = []
  red_misclass_percentages = []
  green_misclass_percentages = []
  thresholds = np.zeros(accuracy + 1)
  increments = (interval[1]-interval[0])/accuracy

  points = validation_data[0]
  labels = validation_data[1].astype(int)

  number_of_points = len(labels)
  red_points = len(np.where(labels==1)[0])
  green_points = len(np.where(labels==0)[0])

  # Initialize and fit model
  model.set_weights(initialWeights)

  model.compile(optimizer='adam', loss=construct_custom_penalty_loss(PENALTY),
                metrics=['accuracy']) # Compile model with penalty

  history = model.fit(x, y, batch_size, epochs, verbose=0,
                      validation_data=validation_data)

  if verbose > 0:
    printProgressBar(0, accuracy+1)

  # MAIN LOOP
```

```
  for i in range(accuracy+1):
    threshold = interval[0] + (interval[1]-interval[0])*(i/accuracy)

    prediction = thresholdPredict(validation_data[0], model, threshold)

    correct_indices = np.where((labels == np.argmax(prediction, axis=1)) == True)
    incorrect_indices = np.where((labels == np.argmax(prediction, axis=1)) == False)

    total_misclassifications = np.bincount(labels == np.argmax(prediction,axis=1))[0]
    red_misclassifications = len(np.where(labels[incorrect_indices] == 1)[0])
    green_misclassifications = len(np.where(labels[incorrect_indices] == 0)[0])

    total_misclass_percentages.append((total_misclassifications/number_of_points)*100)
    red_misclass_percentages.append((red_misclassifications/red_points)*100)
    green_misclass_percentages.append((green_misclassifications/green_points)*100)

    thresholds[i] = threshold

    if verbose > 0:
      printProgressBar(i+1, accuracy+1)

  # PLOTTING RESULTS
  if verbose > 0:
    plt.figure(figsize=(20,15))
    plt.plot(thresholds, total_misclass_percentages, 'b', thresholds,
             red_misclass_percentages, 'r', thresholds, green_misclass_percentages,
             'g')
    plt.title(f'Dataset {CURRENT_SET}: Misclassification by certainty threshold')
    plt.ylabel('% misclassified')
    plt.xlabel('Certainty threshold')
    plt.xticks(np.arange(interval[0], interval[1]+increments, increments))
    plt.legend(['total', 'red', 'green'], loc='upper left')
    plt.show()

  return (total_misclass_percentages, red_misclass_percentages,
       green_misclass_percentages)


def averageThresholdEffect(model, n, valSet_size, path='', interval=(0.8,1),
                           accuracy=10, batch_size=64, epochs=500, verbose=1,
                           useBalanceDataset=False):
  """Plots average certainty threshold effect over n iterations.

  Args:
    model: keras model
      Model for which the certainty threshold effect is measured.
    n: int
      Number of iterations the certainty threshold effect is measured and
      averaged over.
    valSet_size: int
      Size of the validation set.
    path: str, optional
      Path to which the excel sheet will be saved. e.g. '/content/drive/MyDrive/'
```

```
  verbose: boolean, optional
    Whether to print progress bar or not.
  useBalanceDataset: boolean, optional
    Whether to balance the dataset before training or not.
  All others:
    See calculateThresholdEffect.

Returns:
  3-tuple of np arrays (total_misclass_percentages_avg,
  red_misclass_percentages_avg, green_misclass_percentages_avg).
"""
#Start time
start_time = time.time()

thresholds = np.arange(interval[0], interval[1]+(interval[1]-interval[0])/accuracy,
                       (interval[1]-interval[0])/accuracy)

# INITIALIZATION OF DATA COLLECTION OBJECTS
# For averaging
total_misclass_percentages_collected = []
red_misclass_percentages_collected = []
green_misclass_percentages_collected = []
# For saving in excel
validation_points_collected = np.zeros((valSet_size, 3*n))
misclassification_matrix = np.zeros((len(thresholds), 3*n))
# Column names
val_columns = []
coll_columns = []

# Initialize progress bar
if verbose > 0:
  printProgressBar(0, n)

# MAIN LOOP
for i in range(n):
  # PREPARING DATA
  dataSet = getDataSet()
  dataSet.pop('Unnamed: 0') #Removing unnessary column

  # Choose random validation set
  val_indices = random.sample(range(SOURCE_SIZE[CURRENT_SET]), valSet_size)

  valSet_points, valSet_labels = separateValidationSet(dataSet=dataSet,
                                                      validationIndices=val_indices)

  if useBalanceDataset:
    dataSet = balanceDataset(dataSet, threshold=THRESHOLD_DATA, verbose=0)

  training_labels = np.array(dataSet['l_i']).astype('float')
  training_points = np.array(dataSet[['x_i1','x_i2']])

  # Collecting misclassification percentages
  allPercentages = calculateThresholdEffect(model, training_points, training_labels,
```

```
                                            (valSet_points, valSet_labels),
                                            interval=interval, accuracy=accuracy,
                                            batch_size=batch_size, epochs=epochs,
                                            verbose=0)

    total_misclass_percentages_collected.append(allPercentages[0])
    red_misclass_percentages_collected.append(allPercentages[1])
    green_misclass_percentages_collected.append(allPercentages[2])

    # Creating separate columns for validation set
    val_columns.append(f'x_i1:{i}')
    val_columns.append(f'x_i2:{i}')
    val_columns.append(f'l_i:{i}')

    for j in range(valSet_size):
      validation_points_collected[j,3*i + 0] = valSet_points[j, 0]
      validation_points_collected[j,3*i + 1] = valSet_points[j, 1]
      validation_points_collected[j,3*i + 2] = valSet_labels[j]

    # Creating seperarte columns for current misclassification
    coll_columns.append(f'total:{i}')
    coll_columns.append(f'red:{i}')
    coll_columns.append(f'green:{i}')

    misclassification_matrix[:, 3*i + 0] = allPercentages[0]
    misclassification_matrix[:, 3*i + 1] = allPercentages[1]
    misclassification_matrix[:, 3*i + 2] = allPercentages[2]

    if verbose > 0:
      printProgressBar(i+1, n)

  # Averaging
  total_misclass_percentages_avg = np.average(total_misclass_percentages_collected,␣
↪axis=0)
  red_misclass_percentages_avg = np.average(red_misclass_percentages_collected, axis=0)
  green_misclass_percentages_avg = np.average(green_misclass_percentages_collected,␣
↪axis=0)

  result = (total_misclass_percentages_avg, red_misclass_percentages_avg,
            green_misclass_percentages_avg)

  # PLOTTING RESULTS
  plotThresholdEffect(model, data=result, interval=interval, accuracy=accuracy,
                      n=n, valSet_size=valSet_size, batch_size=batch_size,
                      epochs=epochs, path=path)

  # Print time taken for calculation
  end_time = time.time()
  total_time = (end_time-start_time)/60
  print(f'Time taken: {round(total_time, 2)} minutes.')

  # Save results to excel
  today = date.today()
```

```python
    writer = pd.ExcelWriter(f'{path}CertaintyThreshold_Data_{CURRENT_SET}_' +
                            f'{model.name}_{today.strftime("%d-%m-%Y")}.xlsx')

    # Average misclass percentages
    pd.DataFrame([total_misclass_percentages_avg, red_misclass_percentages_avg,
                  green_misclass_percentages_avg], ['total','red','green'],
                 columns=thresholds).to_excel(writer, sheet_name=f'Average')

    # Misclass percentages collected
    pd.DataFrame(misclassification_matrix, thresholds,
                 columns=coll_columns).to_excel(writer, sheet_name=f'Collected')

    # Parameters
    data = {'Values':[f'{model.name}', f'{CURRENT_SET}', f'{n}', f'{valSet_size}',
                      f'{PENALTY}', f'{interval}', f'{accuracy}', f'{batch_size}',
                      f'{epochs}', f'{useBalanceDataset}']}

    index = ['model','dataset','n','valSet_size','penalty','interval','accuracy',
             'batch_size','epochs','useBalanceDataset']

    pd.DataFrame(data, index=index).to_excel(writer, sheet_name='Parameters')

    # Validation sets
    pd.DataFrame(validation_points_collected,
                 columns=val_columns).to_excel(writer, sheet_name=f'Validation Sets')

    writer.save()

    return result


def plotThresholdEffect(model, data, interval, accuracy, n, valSet_size,
                        batch_size, epochs, penalty=PENALTY, dataset=CURRENT_SET,
                        ylim=[0,10], maj_yt_incr=1, min_yt_incr=0.1,
                        figsize=(14,10), showParameters=True, resolution=300,
                        path=''):
    """Plots average certainty threshold effect given by 'data' and saves png and
      pdf of plot to the directory.

    Args:
      model: keras model
        Model for which the certainty threshold effect is measured.
      data: 3-tuple of np arrays, or str
        (total_misclass_percentages_avg, red_misclass_percentages_avg,
        green_misclass_percentages_avg) or the name of an Excel sheet present in
        the directory as a String (e.g. 'data.xlsx').
      interval: 2-tuple
        (x,y) which defines the certainty threshold interval plotted. x is the
        lowest penalty, y the highest.
      accuracy: int
        Certainty threshold interval is evenly split into 'accuracy' many points.
      n, valSet_size, batch_size, epochs, penalty:
```

```
    Parameters used for training and calculaing the average certainty
    threshold effect. Shown in configurations text in plot.
  dataset: char, optional
    Dataset which the certainty penalty effect was measured on. 'A', 'B' or
    'C'.
  ylim: 1D list of floats or ints, optional
    [x,y] which defines the range of % misclassification shown on the y-axis.
  maj_yt_incr: float, optional
    The increments in which major y-ticks are plotted on the y-axis.
  min_yt_incr: float, optional
    The increments in which minor y-ticks are plotted on the y-axis.
  figsize: 2-tuple of floats, optional
    (x,y) where x is the width of the plot and y is the height of the plot.
  showParameters: boolean, optional
    Whether to include a configuratiuon text in the plot or not.
  resolution: int, optional
    Resolution of the plot png in dpi.
  path: str, optional
    Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'

Raises:
  TypeError: if data is not of type String or 3-tuple of np arrays.
"""
# Thresholds to be plotted on the x-axis
thresholds = np.arange(interval[0], interval[1]+(interval[1]-interval[0])/accuracy,
                        (interval[1]-interval[0])/accuracy)

# DATA PREPARATION
if (isinstance(data, tuple) and isinstance(data[0], np.ndarray) and
    isinstance(data[1], np.ndarray) and isinstance(data[2], np.ndarray) and
    len(data)==3):
  total_misclass_percentages_avg = data[0]
  red_misclass_percentages_avg = data[1]
  green_misclass_percentages_avg = data [2]

elif isinstance(data, str):
  data = pd.ExcelFile(data)
  avg_data = pd.read_excel(data, 'Average')

  total = pd.DataFrame(avg_data.loc[0])
  total = total.drop('Unnamed: 0')
  total_misclass_percentages_avg = total[0]

  red = pd.DataFrame(avg_data.loc[1])
  red = red.drop('Unnamed: 0')
  red_misclass_percentages_avg = red[1]

  green = pd.DataFrame(avg_data.loc[2])
  green = green.drop('Unnamed: 0')
  green_misclass_percentages_avg = green[2]

else:
  raise TypeError(f'Invalid type of data. data should be of type String or '
```

```
                 + f'a 3-tuple of np arrays, but data is of type {type(data)}.')

# Define yticks
major_yticks = np.arange(0, ylim[1]+maj_yt_incr, maj_yt_incr)
minor_yticks = np.arange(0, ylim[1]+min_yt_incr, min_yt_incr)

# Create subplot
fig, ax = plt.subplots(figsize=figsize)

ax.plot(thresholds, total_misclass_percentages_avg, 'b', thresholds,
          red_misclass_percentages_avg, 'r', thresholds,
          green_misclass_percentages_avg, 'g')

ax.set_title(f'Dataset {dataset}: Average misclassification by certainty threshold',
             fontsize='x-large')
ax.set_ylabel('% misclassified', fontsize='large')
ax.set_xlabel('Certainty threshold', fontsize='large')

# Ranges of x and y-axis
ax.set_xlim(list(interval))
ax.set_ylim(ylim)

# Set ticks
ax.set_xticks(thresholds)
ax.set_yticks(major_yticks)
ax.set_yticks(minor_yticks, minor=True)

# Color and grid
ax.set_facecolor('white')
ax.grid(which='minor', alpha=0.2, color='black')
ax.grid(which='major', alpha=0.5, color='black')

# Show configuration information on plot
if showParameters==True:
  config_info = (f'{model.name}\nn: {n}\nVal. set size: {valSet_size}\n' +
                 f'Batch size: {batch_size}\nEpochs: {epochs}\n' +
                 f'Penalty: {penalty}')
  ax.text(interval[1]+(interval[1]/(8*figsize[0])), ylim[1]-(ylim[1]/figsize[1]),
          config_info)

plt.legend(['total', 'red', 'green'], loc='upper left', fontsize='medium')

plt.show()

# Get current date
today = date.today()

fig.savefig(f'{path}CertaintyThreshold_Plt_{dataset}_{model.name}_' +
            f'{today.strftime("%d-%m-%Y")}.png', dpi=resolution)
fig.savefig(f'{path}CertaintyThreshold_Plt_{dataset}_{model.name}_' +
            f'{today.strftime("%d-%m-%Y")}.pdf')
```

```
[ ]: #@title Points Per Square
def pointsPerSquare(dataSet=CURRENT_SET, accuracy=100):
  """Calculates the number of points from dataSet present in each square of an
    accuracy x accuracy grid.

  Args:
    dataSet: char, optional
      'A', 'B', or 'C'.
    accuracy: int, optional
      The grid consists of accuracy x accuracy many squares.

  Returns:
    Three 2-D np.array of the shape (accuracy,accuracy): squares, red and green.
    squares contains the number of total points present in each square, red
    contains the number of red points present in each square, and green contains
    the number of green points present in each square.
  """
  dataSet = getDataSet(dataSet)

  squares = np.zeros((accuracy, accuracy))
  red = np.zeros((accuracy, accuracy))
  green = np.zeros((accuracy, accuracy))

  # Multiply all entries with accuracy to calculate which
  # square each point falls into
  dataSet = dataSet[['x_i1', 'x_i2', 'l_i']]*accuracy

  printProgressBar(0, len(dataSet))

  for i in range(len(dataSet)):
    x_i1 = math.floor(dataSet.loc[i]['x_i1'])
    x_i2 = math.floor(dataSet.loc[i]['x_i2'])

    # If x_i1 or x_i2 coordinate is 1.0, reduce by 1 to prevent index out of
    # bounds
    if x_i1 == accuracy:
      x_i1 = accuracy-1
    if x_i2 == accuracy:
      x_i2 = accuracy-1

    squares[x_i2,x_i1] = squares[x_i2,x_i1]+1

    if (dataSet.loc[i]['l_i']) == 0:
      green[x_i2,x_i1] = green[x_i2,x_i1]+1
    else:
      red[x_i2,x_i1] = red[x_i2,x_i1]+1

    printProgressBar(i+1, len(dataSet))

  return squares, red, green
```

```
[ ]: #@title Misclassifications per square
def misclassPerSquare(model, validationSet, accuracy=10, useThresholdPredict=False,
                      drawGrid=True, verbose=0, savePlot=False, path='',
```

```
                    colorbarLim=-1):
"""Calculates proportion of red, green, and total validaion points
  misclassified per square in an accuracy*accuracy grid.

Args:
  model: keras.model
    The model to perform the predictions.
  validationSet: 2-tuple of np.arrays
    2-tuple of the form (valSet_points, valSet_labels), where valSet_points is
    a np.array of shape (x,2) and valSet_labels is a np.array of shape (x,1).
  accuracy: int, optional
    The dataset is split up into accuracy*accuracy many fields.
  useThresholdPredict: boolean, optional
    Whether to use thresholdPredict (True) or regular model.predict (False).
  drawGrid: boolean, optional
    Whether to draw a grid on the plot or not.
  verbose: 0 or 1, optional
    Whether to plot the misclassifications per square or not.
  savePlot: boolean, optional
    Whether to save the plot or not.
  path: str, optional
    Path to which the plot will be saved. e.g. '/content/drive/MyDrive/'
  colorbarLim: float between 0 and 1, optional
    Upper limit for the colorbar. Defaults to -1 where the maximum misclass
    proportion is used as the upper limit.

Returns:
  3-tuple 2-D np.arrays (total, red, green) of shape (accuracy,accuracy)
  containing the proportions of total, red, and green misclassifications per
  square as floats between 0 and 1.
"""
# Predicting the validation points
valSet_points = validationSet[0]
valSet_labels = validationSet[1]

# Preparing arrays
totalPoints = np.zeros((accuracy,accuracy))
totalMisclass = np.zeros((accuracy,accuracy))
redPoints = np.zeros((accuracy,accuracy))
redMisclass = np.zeros((accuracy,accuracy))
greenPoints = np.zeros((accuracy,accuracy))
greenMisclass = np.zeros((accuracy,accuracy))

# Predicting points
if useThresholdPredict:
  prediction = thresholdPredict(valSet_points, model, MIN_GREEN_CERT)
else:
  prediction = model.predict(valSet_points)

# Identifying incorrectly classified points
incorrect_indices = np.where((valSet_labels != np.argmax(prediction, axis=1)))

# Multiplying all entries with accuracy to calculate which square each
```

```
# validation point falls into
valSet_points = valSet_points*accuracy

if verbose > 0:
  printProgressBar(0, len(valSet_points))

for i in range(len(valSet_points)):
  x_i1 = math.floor(valSet_points[i,0])
  x_i2 = math.floor(valSet_points[i,1])

  # If x_i1 or x_i2 coordinate is 1.0, reduce by 1 to prevent index out of
  # bounds
  if x_i1 == accuracy:
    x_i1 = accuracy-1
  if x_i2 == accuracy:
    x_i2 = accuracy-1

  # Total
  totalPoints[x_i2,x_i1] += 1
  if i in incorrect_indices[0]:
    totalMisclass[x_i2,x_i1] += 1

  # Red
  if valSet_labels[i] == 1:
    redPoints[x_i2,x_i1] += 1

    if i in incorrect_indices[0]:
      redMisclass[x_i2,x_i1] += 1

  # Green
  if valSet_labels[i] == 0:
    greenPoints[x_i2,x_i1] += 1

    if i in incorrect_indices[0]:
      greenMisclass[x_i2,x_i1] += 1

  if verbose > 0:
    printProgressBar(i+1, len(valSet_points))

# Setting all 0 entries in xPoints to 1 to prevent div by zero
totalPoints[totalPoints == 0] = 1
redPoints[redPoints == 0] = 1
greenPoints[greenPoints == 0] = 1

# Calculating proportion of validation points misclassified
totalMisclassPerSquare = totalMisclass/totalPoints
redMisclassPerSquare = redMisclass/redPoints
greenMisclassPerSquare = greenMisclass/greenPoints

# PLOTTING
today = date.today()
# Total
if verbose > 0:
```

```python
    fig, ax = plt.subplots()
    ax.set_title(f'Prop. of val points misclassified in {CURRENT_SET}')

    if colorbarLim == -1:
      colorbarLim = np.max(totalMisclassPerSquare)

    plt.imshow(totalMisclassPerSquare, origin='lower', cmap='Spectral', vmin=0,
              vmax=colorbarLim, extent=[0, 1, 0, 1])

    plt.colorbar()
    ax.set_xlabel('x_i1')
    ax.set_ylabel('x_i2')
    ax.set_xlim((0,1))
    ax.set_ylim((0,1))
    ax.set_xticks([i/10 for i in range(11)])
    ax.set_yticks([i/10 for i in range(11)])
    if drawGrid == True:
      ax.grid(alpha=0.3, color='black')
    plt.show()

  # Saving total plot
  if savePlot == True:
    fig.savefig(f'{path}Total_Misclass_Per_Square_{CURRENT_SET}_' +
                f'{model.name}_{today.strftime("%d-%m-%Y")}.pdf')
    fig.savefig(f'{path}Total_Misclass_Per_Square_{CURRENT_SET}_' +
                f'{model.name}_{today.strftime("%d-%m-%Y")}.png', dpi=300)

  # Red
  if verbose > 0:
    fig, ax = plt.subplots()
    ax.set_title(f'Prop. of r val points misclassified in {CURRENT_SET}')

    if colorbarLim == -1:
      colorbarLim = np.max(redMisclassPerSquare)

    plt.imshow(redMisclassPerSquare, origin='lower', cmap='Spectral', vmin=0,
              vmax=colorbarLim, extent=[0, 1, 0, 1])

    plt.colorbar()
    ax.set_xlabel('x_i1')
    ax.set_ylabel('x_i2')
    ax.set_xlim((0,1))
    ax.set_ylim((0,1))
    ax.set_xticks([i/10 for i in range(11)])
    ax.set_yticks([i/10 for i in range(11)])
    if drawGrid == True:
      ax.grid(alpha=0.3, color='black')
    plt.show()

  # Saving red plot
  if savePlot == True:
    fig.savefig(f'{path}Red_Misclass_Per_Square_{CURRENT_SET}_' +
                f'{model.name}_{today.strftime("%d-%m-%Y")}.pdf')
```

```
    fig.savefig(f'{path}Red_Misclass_Per_Square_{CURRENT_SET}_' +
                f'{model.name}_{today.strftime("%d-%m-%Y")}.png', dpi=300)

  # Green
  if verbose > 0:
    fig, ax = plt.subplots()
    ax.set_title(f'Prop. of g val points misclassified in {CURRENT_SET}')

    if colorbarLim == -1:
      colorbarLim = np.max(greenMisclassPerSquare)

    plt.imshow(greenMisclassPerSquare, origin='lower', cmap='Spectral', vmin=0,
              vmax=colorbarLim, extent=[0, 1, 0, 1])

    plt.colorbar()
    ax.set_xlabel('x_i1')
    ax.set_ylabel('x_i2')
    ax.set_xlim((0,1))
    ax.set_ylim((0,1))
    ax.set_xticks([i/10 for i in range(11)])
    ax.set_yticks([i/10 for i in range(11)])
    if drawGrid == True:
      ax.grid(alpha=0.3, color='black')
    plt.show()

  # Saving green plot
  if savePlot == True:
    fig.savefig(f'{path}Green_Misclass_Per_Square_{CURRENT_SET}_' +
                f'{model.name}_{today.strftime("%d-%m-%Y")}.pdf')
    fig.savefig(f'{path}Green_Misclass_Per_Square_{CURRENT_SET}_' +
                f'{model.name}_{today.strftime("%d-%m-%Y")}.png', dpi=300)

  return (totalMisclassPerSquare, redMisclassPerSquare, greenMisclassPerSquare)

def avgMisclassPerSquare(model, initialWeights, n, valSet_size, batch_size,
                         epochs, accuracy=10, useThresholdPredict=False,
                         drawGrid=True, verbose=1, savePlot=False,
                         saveToExcel=False, path='', colorbarLim=-1,
                         useBalanceDataset=False):
  """Calculates average proportions of red, green, and total validaion points
    misclassified per square in an accuracy*accuracy grid over n training rounds
    and validation sets.

  Args:
    model: keras.model
      The model to perform the predictions.
    initialWeights: array-like
      Initial weights of model.
    n: int
      The number of training/predition rounds to average over.
    valSet_size: int
      Size of the randomly chosen validation sets.
    batch_size: int
```

```
      Batch size used for training the model.
    epochs: int
      Number of epochs used for training the model.
    accuracy: int, optional
      The dataset is split up into accuracy*accuracy many fields.
    useThresholdPredict: boolean, optional
      Whether to use thresholdPredict (True) or regular model.predict (False).
    drawGrid: boolean, optional
      Whether to draw a grid on the plot or not.
    verbose: 0 or 1, optional
      Whether to plot results or not.
    savePlot: boolean, optional
      Whether to save the plot or not.
    saveToExcel: boolean, optional
      Whether to save the results as an Excel document or not.
    path: str, optional
      Path to which the results will be saved. e.g. '/content/drive/MyDrive/'
    colorbarLim: float between 0 and 1, optional
      Upper limit for the colorbar. Defaults to -1 where the maximum avg
      misclass proportion is used as the upper limit.
    useBalanceDataset: boolean, optional
      Whether to balance the dataset before training or not.

  Returns: 6-tuple (at, ar, ag, ct, cr, cg)
    - (at, ar, ag): 2-D np.arrays (avgTotal, avgRed, avgGreen) of shape
    (accuracy,accuracy) containing the avg proportions of total, red, and green
    misclassifications per square as floats between 0 and 1.
    - (ct, cr, cg): 1-D lists of shape (n,) of 2-D np.arrays of shape
    (accuracy,accuracy) containing the proportions of total, red, and green
    misclassifications per square of each run as floats between 0 and 1.
  """
  # Start time
  start_time = time.time()

  # Preparing data collection lists
  total_misclass_collected = []
  red_misclass_collected = []
  green_misclass_collected = []
  avgTotalMisclassPerSquare = np.zeros((accuracy,accuracy))
  avgRedMisclassPerSquare = np.zeros((accuracy,accuracy))
  avgGreenMisclassPerSquare = np.zeros((accuracy,accuracy))
  validationSets = {}

  if verbose > 0:
    printProgressBar(0, n)

  for i in range(n):
    # Preparing data
    dataSet = getDataSet()
    dataSet = dataSet[['x_i1', 'x_i2', 'l_i']]

    # Choose random validation set
    random.seed(time.time())
```

```
  val_indices = random.sample(range(len(dataSet)), valSet_size)

  valSet_points, valSet_labels = separateValidationSet(dataSet, val_indices)

  if useBalanceDataset:
    dataSet = balanceDataset(dataSet, THRESHOLD_DATA, verbose=0)

  training_labels = np.array(dataSet['l_i']).astype('float')
  training_points = np.array(dataSet[['x_i1','x_i2']])

  # Classifying validation set
  model.set_weights(initialWeights)

  history = model.fit(x=training_points, y=training_labels,
                      batch_size=batch_size, epochs=epochs, verbose=0)

  # Calculating misclassification per square
  result = misclassPerSquare(model, (valSet_points,valSet_labels), accuracy,
                             useThresholdPredict)

  total_misclass_collected.append(result[0])
  red_misclass_collected.append(result[1])
  green_misclass_collected.append(result[2])

  # Saving validation set for Excel
  if saveToExcel == True:
    valSet_points = pd.DataFrame(valSet_points)
    valSet_labels = pd.DataFrame(valSet_labels)

    validationSets[f'x_i1:{i}'] = valSet_points[0]
    validationSets[f'x_i2:{i}'] = valSet_points[1]
    validationSets[f'l_i:{i}'] = valSet_labels[0]

  if verbose > 0:
    printProgressBar(i+1, n)


# Averaging
avgTotalMisclassPerSquare = np.average(total_misclass_collected, axis=0)
avgRedMisclassPerSquare = np.average(red_misclass_collected, axis=0)
avgGreenMisclassPerSquare = np.average(green_misclass_collected, axis=0)

# PLOTTING
today = date.today()
# Total
if verbose > 0:
  fig, ax = plt.subplots()
  ax.set_title(f'Avg prop. of points misclassified in {CURRENT_SET}')

  if colorbarLim == -1:
    colorbarLim = np.max(avgTotalMisclassPerSquare)

  plt.imshow(avgTotalMisclassPerSquare, origin='lower', cmap='Spectral', vmin=0,
```

```
            vmax=colorbarLim, extent=[0, 1, 0, 1])

  plt.colorbar()
  ax.set_xlabel('x_i1')
  ax.set_ylabel('x_i2')
  ax.set_xlim((0,1))
  ax.set_ylim((0,1))
  ax.set_xticks([i/10 for i in range(11)])
  ax.set_yticks([i/10 for i in range(11)])
  if drawGrid == True:
    ax.grid(alpha=0.3, color='black')
  plt.show()

# Saving total plot
if savePlot == True:
  fig.savefig(f'{path}Avg_Total_Misclass_Per_Square_{CURRENT_SET}_' +
              f'{model.name}_{today.strftime("%d-%m-%Y")}.pdf')
  fig.savefig(f'{path}Avg_Total_Misclass_Per_Square_{CURRENT_SET}_' +
              f'{model.name}_{today.strftime("%d-%m-%Y")}.png', dpi=300)

# Red
if verbose > 0:
  fig, ax = plt.subplots()
  ax.set_title(f'Avg prop. of r points misclassified in {CURRENT_SET}')

  if colorbarLim == -1:
    colorbarLim = np.max(avgRedMisclassPerSquare)

  plt.imshow(avgRedMisclassPerSquare, origin='lower', cmap='Spectral', vmin=0,
            vmax=colorbarLim, extent=[0, 1, 0, 1])

  plt.colorbar()
  ax.set_xlabel('x_i1')
  ax.set_ylabel('x_i2')
  ax.set_xlim((0,1))
  ax.set_ylim((0,1))
  ax.set_xticks([i/10 for i in range(11)])
  ax.set_yticks([i/10 for i in range(11)])
  if drawGrid == True:
    ax.grid(alpha=0.3, color='black')
  plt.show()

# Saving red plot
if savePlot == True:
  fig.savefig(f'{path}Avg_Red_Misclass_Per_Square_{CURRENT_SET}_' +
              f'{model.name}_{today.strftime("%d-%m-%Y")}.pdf')
  fig.savefig(f'{path}Avg_Red_Misclass_Per_Square_{CURRENT_SET}_' +
              f'{model.name}_{today.strftime("%d-%m-%Y")}.png', dpi=300)

# Green
if verbose > 0:
  fig, ax = plt.subplots()
  ax.set_title(f'Avg prop. of g points misclassified in {CURRENT_SET}')
```

```
  if colorbarLim == -1:
    colorbarLim = np.max(avgGreenMisclassPerSquare)

  plt.imshow(avgGreenMisclassPerSquare, origin='lower', cmap='Spectral', vmin=0,
            vmax=colorbarLim, extent=[0, 1, 0, 1])

  plt.colorbar()
  ax.set_xlabel('x_i1')
  ax.set_ylabel('x_i2')
  ax.set_xlim((0,1))
  ax.set_ylim((0,1))
  ax.set_xticks([i/10 for i in range(11)])
  ax.set_yticks([i/10 for i in range(11)])
  if drawGrid == True:
    ax.grid(alpha=0.3, color='black')
  plt.show()

# Saving green plot
if savePlot == True:
  fig.savefig(f'{path}Avg_Green_Misclass_Per_Square_{CURRENT_SET}_' +
              f'{model.name}_{today.strftime("%d-%m-%Y")}.pdf')
  fig.savefig(f'{path}Avg_Green_Misclass_Per_Square_{CURRENT_SET}_' +
              f'{model.name}_{today.strftime("%d-%m-%Y")}.png', dpi=300)


# Saving to Excel
if saveToExcel == True:
  # Averages
  index = [i/accuracy for i in range(accuracy)]
  index.reverse()  # Reverse index for correct orientation in Excel
  columns = [i/accuracy for i in range(accuracy)]
  # Flip for correct orientation in Excel
  totalFlipped = np.flipud(avgTotalMisclassPerSquare)
  redFlipped = np.flipud(avgRedMisclassPerSquare)
  greenFlipped = np.flipud(avgGreenMisclassPerSquare)

  # Initialize writer
  writer = pd.ExcelWriter(f'{path}Avg_Misclass_Per_Square_' +
                          f'{CURRENT_SET}_{model.name}_' +
                          f'{today.strftime("%d-%m-%Y")}.xlsx')

  # Total
  total = pd.DataFrame(totalFlipped, columns=columns,
                       index=index)

  total.to_excel(writer, sheet_name='Total')

  # Red
  red = pd.DataFrame(redFlipped, columns=columns,
                     index=index)

  red.to_excel(writer, sheet_name='Red')
```

```
# Green
green = pd.DataFrame(greenFlipped, columns=columns,
                   index=index)

green.to_excel(writer, sheet_name='Green')

# Parameters
data = {'Values':[f'{model.name}', f'{CURRENT_SET}', f'{n}', f'{accuracy}',
                  f'{valSet_size}', f'{batch_size}', f'{epochs}',
                  f'{PENALTY}', f'{useThresholdPredict}',
                  f'{MIN_GREEN_CERT}', f'{useBalanceDataset}']}

index = ['model','dataset','n','accuracy','valSet_size','batch_size',
         'epochs','penalty','useThresholdPredict','min_green_cert',
         'useBalanceDataset']

parameters = pd.DataFrame(data, index=index)
parameters.to_excel(writer, sheet_name='Parameters')

# Validation sets
validationSets = pd.DataFrame(validationSets)
validationSets.to_excel(writer, sheet_name='Validation Sets')

writer.save()

# Collected
index = [i/accuracy for i in range(accuracy)]
index.reverse()  # Reverse index for correct orientation in Excel
columns = [i/accuracy for i in range(accuracy)]
# Initialize writer
writer = pd.ExcelWriter(f'{path}Collected_Misclass_Per_Square_' +
                        f'{CURRENT_SET}_{model.name}_' +
                        f'{today.strftime("%d-%m-%Y")}.xlsx')

# Iterate over all training/validation runs
for i in range(n):
  # Flip for correct orientation in Excel
  totalCollectedFlipped = np.flipud(total_misclass_collected[i])
  redCollectedFlipped = np.flipud(red_misclass_collected[i])
  greenCollectedFlipped = np.flipud(green_misclass_collected[i])

  # Total
  total = pd.DataFrame(totalCollectedFlipped, columns=columns,
                     index=index)

  total.to_excel(writer, sheet_name=f'total_{i}')

  # Red
  red = pd.DataFrame(redCollectedFlipped, columns=columns,
                   index=index)

  red.to_excel(writer, sheet_name=f'red_{i}')
```

```
    # Green
    green = pd.DataFrame(greenCollectedFlipped, columns=columns,
                         index=index)

    green.to_excel(writer, sheet_name=f'green_{i}')

  writer.save()

# Print time taken for calculation
if verbose > 0:
  end_time = time.time()
  total_time = (end_time-start_time)/60
  print(f'Time taken: {round(total_time, 2)} minutes.')

return (avgTotalMisclassPerSquare, avgRedMisclassPerSquare,
        avgGreenMisclassPerSquare, total_misclass_collected,
        red_misclass_collected, green_misclass_collected)
```

[ ]:

```
#@title Weighted Misclassification Probability
def weightedMisclassProbability(distributionMap, misclassPerSquare, model,
                                specific_color=None, verbose=1, colorbarLim=-1,
                                drawGrid=True, showTitle=False, savePlot=False,
                                path=''):
  """Calculates the probability that the next point will be misclassified
    by weighting the misclassification probaility of each square with the
    probability distribution of the datset.

  Args:
    distributionMap: 2-D np.array of shape (x,x)
      The distribution map of the dataset.
    misclassPerSquare: 6-tuple of 2-D np.array of shape (x,x) or str
      (at, ar, ag, ct, cr, cg) misclassification probabilities per square as
      floats between 0 and 1 or the name of an Excel sheet present in the
      directory as a String (e.g. 'data.xlsx').
    model: keras.model
      The model used for calculating misclassPerSquare (only used for naming
      files here).
    specific_color: 0 or 1, optional
      If 0, calculates weighted green misclassification probability. If 1,
      analogously for red. If none given, total misclassification probability is
      calculated.
    verbose: 0 or 1, optional
      Whether to show the plot or not.
    colorbarLim: float between 0 and 1, optional
      Upper limit for the colorbar. Defaults to -1 where the maximum
      misclassification probability is used as the upper limit.
    drawGrid: boolean, optional
      Whether to draw a grid on the plot or not.
    showTitle: boolean, optional
      Whether to show the title of the plot or not.
    savePlot: boolean, optional
      Whether to save the plot or not.
    path: str, optional
```

```
    Path to which the plots will be saved. e.g. '/content/drive/MyDrive/'

  Returns:
    Float between 0 and 1. The total probability that the next point will be
    misclassified.

  Raises:
    TypeError: if distributionMap.shape != misclassPerSquare.shape.
    TypeError: if specific_color is not 0, 1, or None.
  """
  if specific_color != None and specific_color != 0 and specific_color != 1:
    raise TypeError(f'specific_color should be 0, 1, or None, but is ' +
                    f'{specific_color}.')

  # Getting data from Excel sheet
  if type(misclassPerSquare) == str:
    data = pd.ExcelFile(misclassPerSquare)
    if specific_color == None:
      data = pd.read_excel(data, sheet_name='Total', index_col=0)
    elif specific_color == 1:
      data = pd.read_excel(data, sheet_name='Red', index_col=0)
    elif specific_color == 0:
      data = pd.read_excel(data, sheet_name='Green', index_col=0)

    misclassPerSquare = np.array(data)
    misclassPerSquare = np.flipud(misclassPerSquare)

  # Getting selected color from tuple
  if type(misclassPerSquare) == tuple:
    if specific_color == None:
      misclassPerSquare = misclassPerSquare[0]
    elif specific_color == 1:
      misclassPerSquare = misclassPerSquare[1]
    elif specific_color == 0:
      misclassPerSquare = misclassPerSquare[2]

  # Checking shapes
  if distributionMap.shape != misclassPerSquare.shape:
    raise TypeError(f'distributionMap.shape and misclassPerSquare.shape must ' +
                    f'be equal. distributionMap.shape is ' +
                    f'{distributionMap.shape} and misclassPerSquare.shape is ' +
                    f'{misclassPerSquare.shape}.')

  weightedMap = distributionMap*misclassPerSquare

  result = np.sum(weightedMap)

  if specific_color == None:
    char = ''
    color = 'total'
    prefix = 'Total_'
  elif specific_color == 0:
    char = 'g '
```

```
    color = 'green'
    prefix = 'Green_'
  elif specific_color == 1:
    char = 'r '
    color = 'red'
    prefix = 'Red_'

  # Plotting
  if verbose > 0:
    fig, ax = plt.subplots()
    if showTitle:
      ax.set_title(f'Weighted {char}misclass probability in {CURRENT_SET}')

    if colorbarLim == -1:
      colorbarLim = np.max(weightedMap)

    plt.imshow(weightedMap, origin='lower', cmap='Spectral', vmin=0,
              vmax=colorbarLim, extent=[0, 1, 0, 1])

    plt.colorbar()
    ax.set_xlabel('x_i1')
    ax.set_ylabel('x_i2')
    ax.set_xlim((0,1))
    ax.set_ylim((0,1))
    ax.set_xticks([i/10 for i in range(11)])
    ax.set_yticks([i/10 for i in range(11)])
    if drawGrid == True:
      ax.grid(alpha=0.3, color='black')
    config_info = (f'Total: {round(result*100,2)}%')
    ax.text(1.05, 1.03, config_info, weight='bold')
    plt.show()

  # Save plot
  today = date.today()
  if savePlot == True:
    fig.savefig(f'{path}Weighted_{prefix}Misclass_Prob_{CURRENT_SET}_' +
                f'{model.name}_{today.strftime("%d-%m-%Y")}.pdf')
    fig.savefig(f'{path}Weighted_{prefix}Misclass_Prob_{CURRENT_SET}_' +
                f'{model.name}_{today.strftime("%d-%m-%Y")}.png', dpi=300)

  print(f'Weighted {color} misclassification probability: {round(result*100,2)}%')

  return result
```

[ ]:

```
#@title Calculate Upper Bounds
def calculateUpperBounds(distributionMap, misclassPerSquare, n, proportionOfRuns,
                         model, verbose=1, saveToExcel=False, path=''):
  """Calculates the upper bounds for total, red, and green misclassification
    probability fulfilled by a given proportion of runs.

  Args:
    distributionMap: 2-D np.array of shape (x,x)
      The distribution map of the dataset.
    misclassPerSquare: 6-tuple of 2-D np.array of shape (x,x) or str
```

```
    (at, ar, ag, ct, cr, cg) misclassification probabilities per square as
    floats between 0 and 1 or the name of an Excel sheet present in the
    directory as a String (e.g. 'data.xlsx').
  n: int
    The number of training/prediction rounds. (The number of misclassPerSquare
    maps collected in misclassPerSquare).
  proportionOfRuns: float between 0 and 1
    The proportion (as a float) of runs which must fulfill the upper bound.
  model: keras.model
    The model used for calculating misclassPerSquare (only used for naming
    files here).
  verbose: 0 or 1, optional
    Whether to print the results or not.
  saveToExcel: boolean, optional
    Whether to save the calculated upper bounds to Excel or not.
  path: str, optional
    Path to which the sheet will be saved. e.g. '/content/drive/MyDrive/'

Returns:
  3-tuple of floats (t,r,g) containing the upper bounds for total, red, and
  green misclassification probability fulfilled by the given proportion of
  runs.

Raises:
  TypeError: if distributionMap.shape is not equal to shape of each
  misclassPerSquare array/sheet.
"""
# Preparing data arrays
total_misclass_collected = []
red_misclass_collected = []
green_misclass_collected = []
total_misclass_probs = []
red_misclass_probs = []
green_misclass_probs = []

# Getting data from Excel sheet
if type(misclassPerSquare) == str:
  data = pd.ExcelFile(misclassPerSquare)
  for i in range(n):
    data_t = pd.read_excel(data, sheet_name=f'total_{i}', index_col=0)
    data_t = np.array(data_t)
    total_misclass_collected.append(np.flipud(data_t))

    data_r = pd.read_excel(data, sheet_name=f'red_{i}', index_col=0)
    data_r = np.array(data_r)
    red_misclass_collected.append(np.flipud(data_r))

    data_g = pd.read_excel(data, sheet_name=f'green_{i}', index_col=0)
    data_g = np.array(data_g)
    green_misclass_collected.append(np.flipud(data_g))

# Getting selected color from tuple
if type(misclassPerSquare) == tuple:
```

```
    total_misclass_collected = misclassPerSquare[3]
    red_misclass_collected = misclassPerSquare[4]
    green_misclass_collected = misclassPerSquare[5]

  # Checking for correct shapes
  if distributionMap.shape != total_misclass_collected[0].shape:
    raise TypeError(f'distributionMap.shape and shape of misclassPerSquare ' +
                    f'arrays must be equal. distributionMap.shape is ' +
                    f'{distributionMap.shape} and shape of misclassPerSquare ' +
                    f'is {total_misclass_collected[0].shape}.')

  # Calculating misclassification probabilities for each run and color
  for i in range(n):
    t_prob = (np.sum(distributionMap*total_misclass_collected[i]))
    total_misclass_probs.append(t_prob)

    r_prob = (np.sum(distributionMap*red_misclass_collected[i]))
    red_misclass_probs.append(r_prob)

    g_prob = (np.sum(distributionMap*green_misclass_collected[i]))
    green_misclass_probs.append(g_prob)

  # Calculating upper bounds
  total_misclass_probs = np.sort(total_misclass_probs)
  red_misclass_probs = np.sort(red_misclass_probs)
  green_misclass_probs = np.sort(green_misclass_probs)

  target = math.ceil(n*proportionOfRuns)

  res_t = total_misclass_probs[target-1]
  res_r = red_misclass_probs[target-1]
  res_g = green_misclass_probs[target-1]

  max_t = total_misclass_probs[-1]
  max_r = red_misclass_probs[-1]
  max_g = green_misclass_probs[-1]

  res = pd.DataFrame([[res_t, max_t],
                      [res_r, max_r],
                      [res_g, max_g]],
                     index=['total','red','green'],
                     columns=[f'Fulfilled by {round(proportionOfRuns*100,2)}%',
                              f'Fulfilled by 100%'])

  # Save to Excel
  today = date.today()
  if saveToExcel == True:
    res.to_excel(f'{path}Upper_Bounds_{CURRENT_SET}_{model.name}_' +
                 f'{today.strftime("%d-%m-%Y")}.xlsx', sheet_name='Upper bounds')

  if verbose > 0:
    print(res)
```

```
    return (res_t, res_r, res_g)
```

4 Magic

[ ]:
```
# Data Preparation
dataSet = getDataSet()
dataSet = dataSet[['x_i1', 'x_i2', 'l_i']] # Removing every unnecessary columns

# Separate the validation set
valSet_points, valSet_labels = separateValidationSet(dataSet=dataSet,␣
 ↪validationIndices=VAL_INDICES)

# Balance dataset
dataSet = balanceDataset(dataSet, threshold=THRESHOLD_DATA)

# Creating training arrays
training_labels = np.array(dataSet['l_i']).astype('float')
training_points = np.array(dataSet[['x_i1','x_i2']])
```

[ ]:
```
# Configure and compile model
initalizer = keras.initializers.GlorotNormal()

model_0 = keras.Sequential([
          keras.layers.Flatten(input_shape=(2,)),
          keras.layers.Dense(100,activation='relu', kernel_initializer=initalizer),
          keras.layers.Dense(70,activation='relu', kernel_initializer=initalizer),
          keras.layers.Dense(50,activation='relu', kernel_initializer=initalizer),
          keras.layers.Dense(10,activation='relu', kernel_initializer=initalizer),
          keras.layers.Dense(2,activation='softmax', kernel_initializer=initalizer)
          ], name="model_0")

model_0.compile(optimizer='adam', loss=construct_custom_penalty_loss(PENALTY),
                metrics=['accuracy'])

# Save initial weights
initialWeights = model_0.get_weights()

# Fit model
history = model_0.fit(training_points, training_labels, batch_size=2000, epochs=50,
                    shuffle=True, validation_data=(valSet_points, valSet_labels))
clear_output()
```