FYS-3012 Home Exam

Candidate 78

UiT - Norges Arktiske Universitet

October 30, 2023

Task 1

1a)

To find J_{min} we look at the general l dimensional case where all the classes are identical.
We have:

$$J_3 = trace\{S_w^{-1} S_m\}$$

We also have that:

$$S_m = S_w + S_b$$

We can first look at S_b, which is defined as:

$$S_b = \sum_{i=1}^{M} P_i(\mu_i - \mu_0)(\mu_i - \mu_0)^T$$

Since all classes are identical we know that all the mean vectors, as well as the global mean vector are equal. Meaning S_b is a $l \times l$ zero matrix, and thus $S_m = S_w$
Which gives:

$$J_{min} = trace\{S_w^{-1} S_w\} = trace\{I\}$$

Where I is the $l \times l$ identity matrix. Since the trace operation is defined as the sum of diagonal elements we get that trace{I} is equal to l

$$J_{min} = l$$

But we can still simplify our modified J_3 score

$$\begin{aligned}
J_3 &= trace\{S_w^{-1} S_m\} - J_{min} \\
&= trace\{S_w^{-1}(S_w + S_b)\} - J_{min} \\
&= trace\{S_w^{-1} S_w + S_w^{-1} S_b\} - J_{min} \\
&= trace\{S_w^{-1} S_b\} + trace\{S_w^{-1} S_w\} - J_{min} \\
&= trace\{S_w^{-1} S_b\} + J_{min} - J_{min} \\
&= trace\{S_w^{-1} S_b\}
\end{aligned}$$

So for our modified J_3 score we have

$$J_3 = trace\{S_w^{-1} S_m\} - J_{min} = trace\{S_w^{-1} S_b\}$$

This modified J_3 score is very useful for feature selection, often in a classification task one will have an abundance of features, and for simplification and reducing the computational load be required to only keep the ones with most information. With this modified J_3 score we will now get a value of 0 if the classes are identical in a given feature space, for any number of dimensions. This means that we can check all or most of the combinations of features and check their J_3 score, and if one is zero or close to zero we know that the class separability is low and we don't get much information from that feature(s). If we didn't use the modified J_3 score this process would be more cumbersome, since we would have to remember how many dimensions we have for a given feature space and take that into account for the feature selection process.

1b)

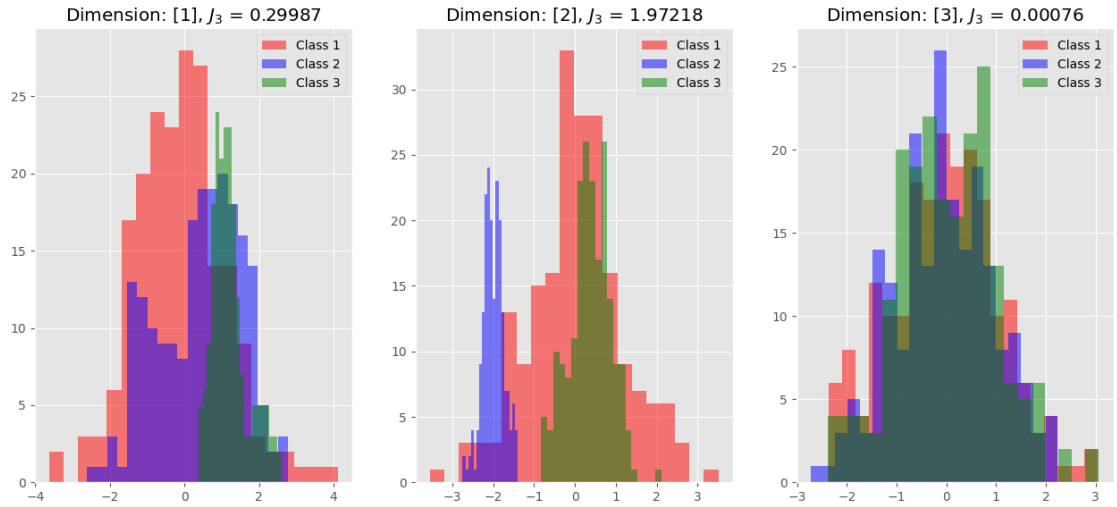


Figure 1: Histograms of the training data in all three dimensions

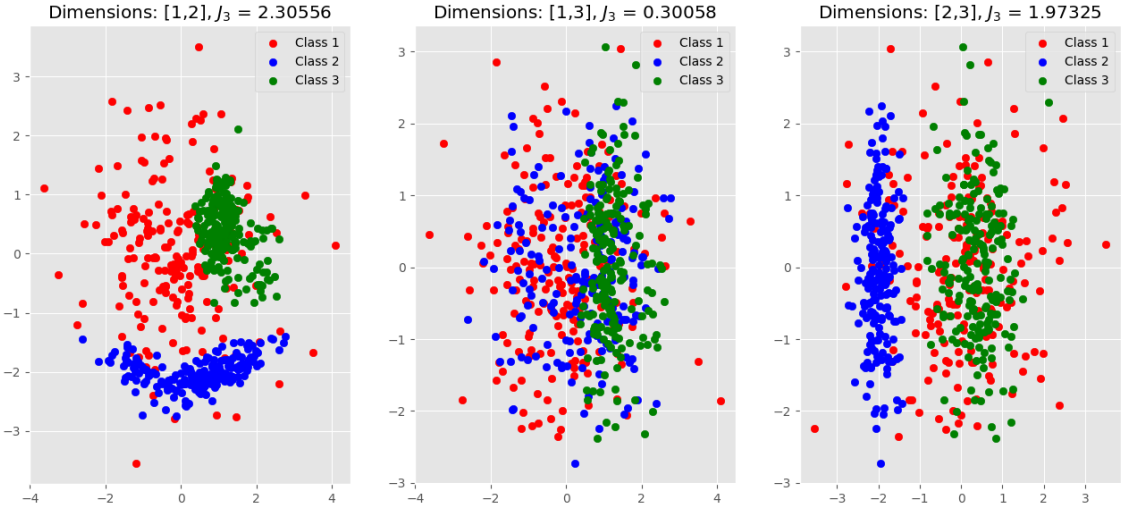


Figure 2: Scatter plots of all 2d combinations of the training data

1c)

In figure 1 we can see that the J_3 score correlates with the separability of the classes. The histogram plot of dimension 3 is a good example of this, we can see that the classes are more or less completely on top of each other, meaning the mean vectors and variance in this dimension would be close to identical for all classes. Which means we can't really separate the classes in this dimension, and the J_3 score is basically 0. J_3 is not exactly zero, but this won't really happen in the real world, but this is close enough that we can say that it is zero. On the other hand we can look at dimension 2, where we can see three distinct 'peaks', just by looking at the plot we can clearly see that the separability is higher, and that separating the classes is easier than in dimension 3. The J_3 score also backs this up, as it is much higher. Dimension 1 is somewhere in the middle of these two.

Figure 2 also shows that the J_3 score correlates with class separability. We can easily see that the most separable combination of 2 dimensions has the highest J_3 score. That being dimensions one and two. This makes sense considering our conclusion from figure 1. Since dimension 3 has the lowest separability, the combination of dimensions which don't include dimension 3 will be the one with the highest separability.

The 3-dimensional data set has a J_3 score of 2.30658, which is more or less the same as dimensions 1 and 2. Again this matches our conclusion earlier. Since the J_3 score for the 3 dimensions is more or less the same as for dimensions 1 and 2 we can get rid of the third dimension without any difference in our classification accuracy. For this reason the following tasks will be done in the 2-dimensional feature space, omitting dimension 3.

Task 2

2a)

This task was done by using the built in numpy functions for calculating the mean vectors and covariance matrices, for each class. These values from the training set were then used to calculate the Gaussian probability density function for each class, given by:

$$p(\boldsymbol{x}) = \frac{1}{(2\pi)^{l/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right)$$

This Gaussian PDF was calculated for each class, and the classification was done by assigning each feature vector to the class whose PDF has the largest value at that specific point in the feature space.

Gaussian classifier		
Data set	Accuracy	Misclassifications
Training set	89.5%	63
Test set	86.8%	79

Table 1: Classification results for the Gaussian classifier

Table 1 shows the results of the Gaussian maximum likelihood classifier, as we can see the results

are quite good, with an 86.5% accuracy for the test set. As we saw in task 1 the class separability is not that high, so this result is quite good, especially for such a simple classifier.

2b)

The basic principle for the Parzen window density estimator is that we create a probability density function (pdf) that is the sum of N Gaussians, each centered on a feature vector in the training set. Where N is the number of training vectors. Other kernels than the Gaussian can be used, but for this task we are only concerned with the Gaussian. This type of pdf can estimate more or less any kind of distribution, given that the training set is large enough. The kernel itself is a symmetric Gaussian distribution, but since we take the sum of many different Gaussians, the resulting pdf can take up any shape, Gaussian or not. The expression for the Parzen pdf is:

$$\hat{p}(\boldsymbol{x}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{(2\pi)^{\frac{l}{2}}h^l}\exp\left(-\frac{(\boldsymbol{x}-\boldsymbol{x}_i)^T(\boldsymbol{x}-\boldsymbol{x}_i)}{2h^2}\right)$$

The only free parameter here is h, which is a measure of the size of each Gaussian kernel. A low value for h will give a narrow and tall kernel. While a large h will give more flat and spread out kernel. To get a good approximation of the distribution of the training set a sufficiently small value for h should be chosen, but there is a problem with choosing h to be too small. In the limit where h goes to zero, each Gaussian will approximate the delta function, becoming infinitely tall and infinitely narrow. Meaning that we will get a perfect 100% correct estimation of the training set, but the classifier will not be able to classify anything else correctly, essentially making the classifier useless. This phenomenon where the classifier becomes too well adapted to the training set to the point where it can't classify anything else correctly is called overfitting. This means that we have to choose h carefully to get an accurate classifier, without overfitting.

2c)

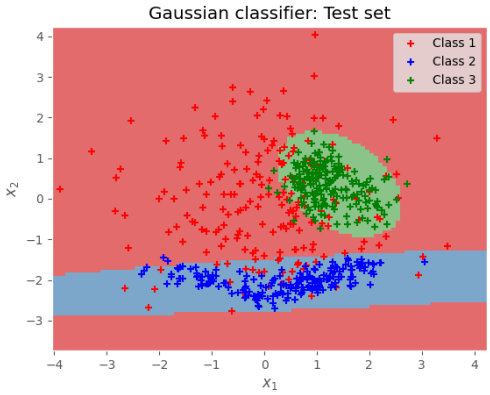
Implementing this classifier in Python is very straightforward. The PDF for each class is created by implementing the equation listed above into the code. Then the feature vectors are classified by evaluating the PDFs for each class at that feature vector, and choosing the one with the highest value

Parzen classifier		
Data set	Accuracy	Misclassifications
Training set	94.3%	34
Test set	87.3%	76

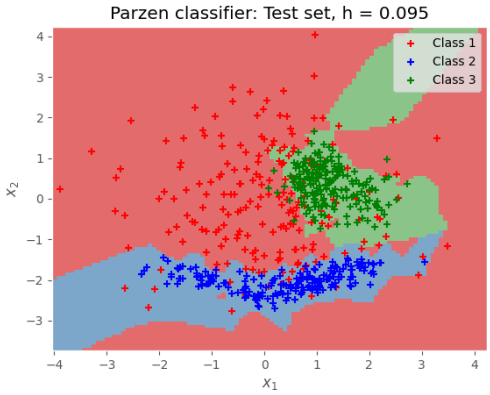
Table 2: Classification results for the Parzen window density estimator

The results of the Parzen classifier can be seen in table 2, for this result h was set to 0.095, which is the most optimal with regard to the classification accuracy for the test set. This value for h was found through trial and error. We can get a higher accuracy for the training set if we choose a low value for h, but as mentioned earlier this will cause overfitting, thus decreasing the accuracy for the test set. As we can see the results for the Parzen classifier are a bit better than the Gaussian classifier, but not by much. The test set has 3 fewer misclassifications than for the Gaussian classifier, which is good, but it is of course not a huge difference.

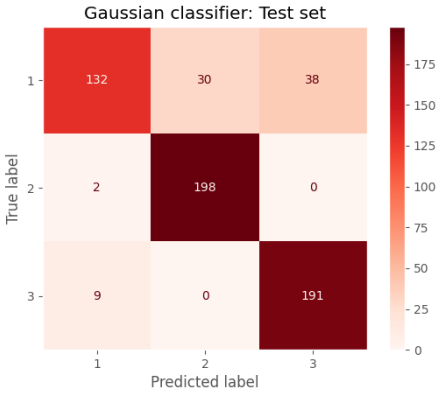
2d)



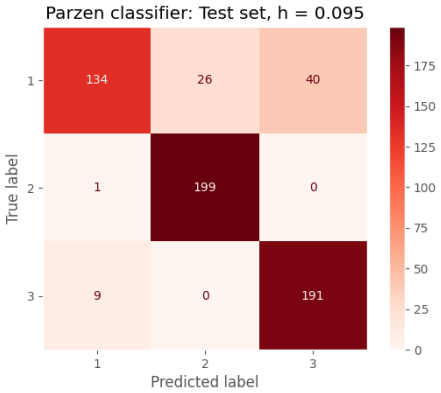
(a) Decision boundary of the Gaussian ML classifier



(b) Decision boundary of the Parzen classifier



(a) Confusion matrix for the Gaussian ML classifier



(b) Confusion matrix for the Parzen classifier

2e)

As we can see in figures 3a and 3b, the decision boundary for the Gaussian classifier is a bit simpler than the Parzen window classifier. The Gaussian classifier creates three Gaussians, one for each class, and the resulting decision boundary gives a visual understanding of the shapes of these Gaussians. The third class is very 'clumped' together, which gives a narrow and tall peak, which we can see gives a small region for class 3. The feature vectors in class 2 are placed more linearly, and narrow. This is also reflected in the decision boundary, which is a narrow band. The first class has the highest variance, meaning it's most spread out, which means that the pdf for this class is very spread out, we can see this in the decision boundary as the 'region' for class 1 is very large.

The Parzen window classifier has a rather different decision boundary, here we don't only have contiguous regions as for the Gaussian classifier. Here the decision boundary is shaped more to the specific 'shape' of the classes in the training set. The specific peculiarities of the training set are more clear here, for example outliers have more of an effect on the decision boundary. This is apparent in the top left corner, where one outlier in class 3 'stretch' the decision boundary to include the points in the top left in class 3. This doesn't help the classifier, as we can see that this region contains feature vectors from class 1. This could likely be 'fixed' by modifying h, but this would lead to misclassifications elsewhere.

Figures 4a and 4b show the confusion matrices of the two classifiers, these figures show us how many feature vectors are classified correctly, and how many are misclassified. We can also see what class the misclassified feature vectors are classified as. As we an see both classifiers are very good at classifying classes 2 and 3. Feature vectors in class 1 however often get misclassified into the other two classes.

As mentioned earlier we can cause overfitting in the Parzen classifier by choosing a small value for h. To visualize this we can look at an example of what happens when we choose h to be small. With h = 0.001 the decision boundary for the Paezen classifier becomes this:

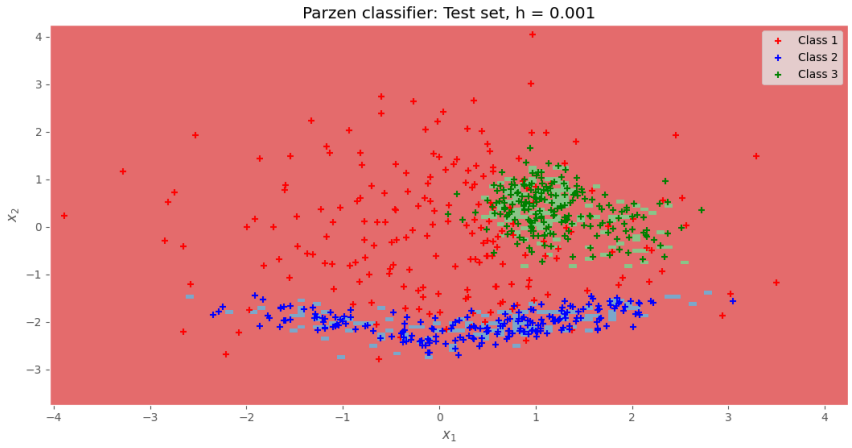


Figure 5: Decision boundary for Parzen classifier with h=0.001

Here we can see a good example of overfitting, the decision boundary is only focused at the feature vectors in the training set. This causes a drop in accuracy for the test set. For this classifier the accuracy for the test set is 50.2%, a huge drop from earlier. The accuracy for the training set however is at 100%. This shows the importance of choosing a good value for h.

Task 3

3a)

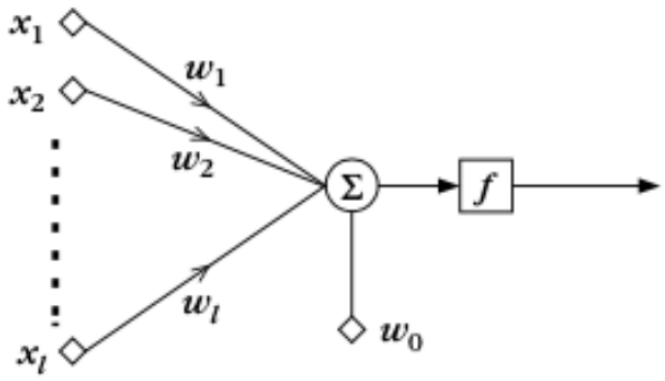


Figure 6: Perceptron node (image borrowed from exam sheet)

Figure 6 shows a Perceptron node.
First the input $\mathbf{x}$=[x_1, x_2, ..., x_l] gets sent to the Perceptron, for a single layer Perceptron network, or for the input layer Perceptrons of a multi-layer Perceptron, this is the feature vector we want to classify. For the hidden layers in a multi-layer network, this input will be the output of the Perceptrons in the previous layer. Then the dot product of the input with the weight vector is taken. The weight vector $\mathbf{w}$= [w_1, w_2, ..., w_l] will give a weighting to each feature, or input to the Perceptron. At this step we also add the threshold w_0. The weight vector along with the threshold will define the decision hyperplane used for class separation. This step, denoted by Σ in figure 6 is the discriminant function $g_j(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$. The value from the discriminant function is then 'sent' to the activation function f. The unit step function is often used for the activation function, that is the output is 1 if the input is positive, and 0 if the input is negative. More commonly used is a sigmoid function, this is due to the discontinuity of the unit step function. In the backpropagation algorithm we take the derivative of the activation function, which is not possible for the unit step function. The sigmoid functions however are differentiable, so they are more commonly used, and what will be used later in the task (more specifically the logistic function will be used). Thus the output of the Perceptron will be a value in the range [0,1], where the higher the value is, the higher likelihood that the feature vector belongs to a given class. More accurately, the output value is a measure of the distance between the feature vector and the decision hyperplane defined by $g_j(\mathbf{x})$.

3b)

This single layer Perceptron network can be coded by creating a class for the Perceptron, where the class attributes are the weights, prediction(output) and the true labels of the set of feature vectors. The methods for this class will be to run forward, which will be to calculate the output of the Perceptron. The other method will be to update the weights of the Perceptron. The expression

for updating the weights is:

$$\Delta \boldsymbol{w_j} = -\rho \frac{\partial J}{\partial \boldsymbol{w_j}}$$

Where $\Delta \boldsymbol{w_j}$ is the change in w_j in consecutive epochs. j denotes the weight vector for the jth Perceptron. Using the squared error cost function gives us:

$$\frac{\partial J}{\partial \boldsymbol{w_j}} = \frac{\partial}{\partial \boldsymbol{w_j}} \frac{1}{2} \sum_i^N (y(i) - \hat{y}(i))^2$$

y(i) is not dependent on w_j so we can treat it as a constant during derivation, and using the chain rule we get:

$$\frac{\partial J}{\partial \boldsymbol{w_j}} = -\sum_i^N (y(i) - \hat{y}(i)) \frac{\partial}{\partial \boldsymbol{w_j}} \hat{y}(i)$$

Again, we can use the chain rule to get:

$$\begin{aligned} \frac{\partial}{\partial w_j} \hat{y}(i) &= \frac{\partial}{\partial w_j} f({w_j}^T \mathbf{x}) \\ &= \frac{\partial f({w_j}^T \mathbf{x})}{\partial {w_j}^T \mathbf{x}} \frac{\partial {w_j}^T \mathbf{x}}{\partial w_j} \\ &= f'({w_j}^T \mathbf{x}) \mathbf{x} \end{aligned}$$

Giving us the final expression for updating the weights:

$$\Delta \boldsymbol{w_j} = \sum_i^N \rho (y(i) - \hat{y}(i)) f'({w_j}^T \mathbf{x}) \mathbf{x}$$

This expression is very straight forward to implement in the code.
To set up the network itself we can create a class called neural_network which creates instances of the Perceptron class, trains them by calling on the methods in the Perceptron class T times, where T is the number of epochs. The Perceptrons will take a one vs. all approach, which means it compares the likelihood that a feature vector belongs to one class against all the others. So to interpret the output of the three perceptrons in the network we can just choose the perceptron with the highest valued output, if that corresponds to class 1, then we classify the feature vector to class 1 etc.. To get the Perceptrons to do this one vs. all approach we have to modify the training labels. We will have three Perceptrons, one for each class, each will output a measure of the likelihood that the given feature vector belongs to their class. To give each Perceptron a class we have to modify the training labels so that for each Perceptron the label for the class they are to check is 1 if the feature vector belongs to that class, and 0 else. So for example for the Perceptron comparing class 1 against the other two classes we want to modify the training labels such that the label is 1 for feature vectors belonging to class 1, and 0 for feature vectors belonging to classes 2 and 3. These training labels are y(i) in the equation above.

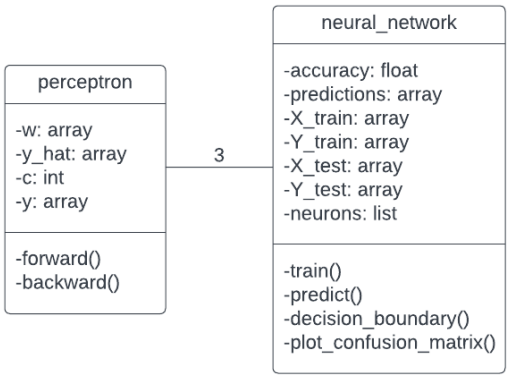
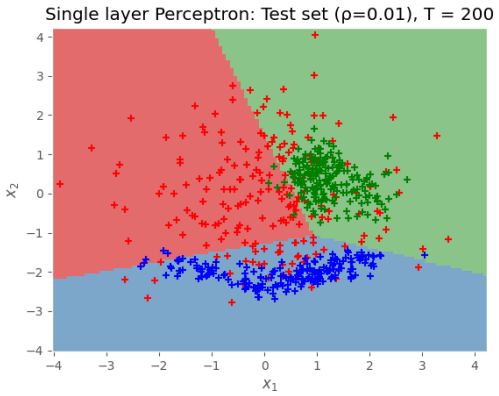


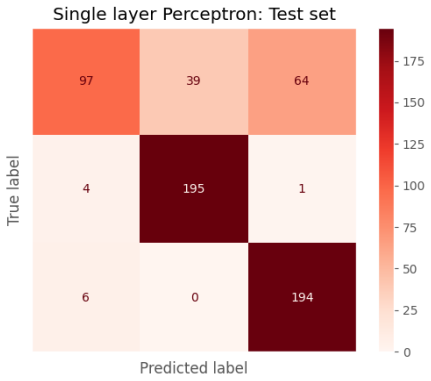
Figure 7: Simplified UML-diagram of single layer perceptron code

Figure 7 shows a simple schematic of the code used in the implementation of the single layer Perceptron network.

3c)



(a) Decision boundary for single layer Perceptron



(b) Confusion matrix for single layer Perceptron

Single layer Perceptron		
Data set	Accuracy	Misclassifications
Training set	86.7%	80
Test set	81.0%	115

Table 3: Classification results for the single layer Perceptron

Figure 8a shows the decision boundary for the single layer Perceptron network. The decision boundary looks more or less exactly as we would expect. The weight vector for each Perceptron describes a decision hyperplane (a line in this 2d case), and we have 3 Perceptrons, so we would expect to see three straight lines separating the feature vectors. Of course this data set is not linearly separable, nor is it very close to being linearly separable, so the accuracy of the network is not amazing. As table 3 shows we get an 81% accuracy with this classifier, which is pretty good. Figure 8b shows the confusion matrix for the network, and we can see that it is quite similar as for the previous classifiers in that it does a good job classifying classes 2 and 3, but struggles more on class 1. The parameters we have to adjust the network are the number of epochs T, which is the amount of times we update our weights, and we have the learning rate ρ. The learning rates decides how 'big' of a step we take when updating the weights. There is not necessarily an optimal value for the learning rate, but we have to choose it carefully depending on the classification task. If we have a big value for ρ we will approach the optimal solution faster, but we might overshoot it. Meaning that as we descend down the gradient of the cost function, we never reach the optimal solution. Choosing a small ρ we are sure to reach the optimal solution, but it will take a long time. With a small learning rate we also increase the chance of getting trapped in a local minima. Meaning we will get stuck at a suboptimal solution. For this task ρ was set to 0.01, which was the optimal value found by trial and error.

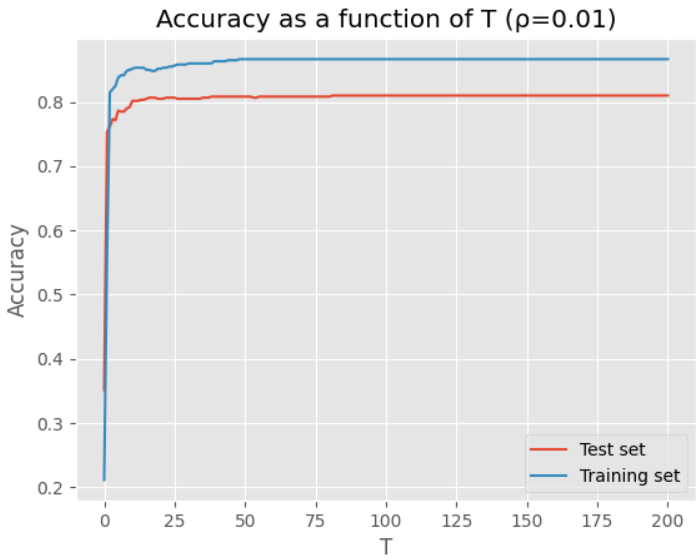


Figure 9: Accuracy as a function of the number of epochs

Figure 9 shows the accuracy of the Perceptron network as a function of the number of epochs. As we can see the accuracy begins at around 33%, which is expected since at T=0 the classifier is just guessing, so at T=0 we expect to get approximately 33% accuracy. Then it rapidly reaches around 80% for the test set, and after around 80 epochs it doesn't improve. For the training set it has reached its max value at around 50 epochs, and is constant after that. This means that we likely

have the optimal solution, since our guess doesn't improve any more.
Compared to the Bayesian classifiers from the previous task this classifier obviously doesn't perform as well. This stems from the fact that this is a linear classifier, and our data set is not linearly separable. For other data sets however the Perceptron network might perform much better than the Bayesian classifiers, but that is very dependant on the distribution of the data set. One thing that is better with this classifier than the others, especially the Gaussian classifier, is that we don't need to know anything about the distribution of the feature vectors before we classify them. For the Gaussian classifier we need knowledge about the data set to know if the Gaussian classifier will do a good job. In terms of computational efficiency the Perceptron network outperforms the Parzen window density estimator and the Gaussian classifier, given that the number of epochs is not to large. So what the Perceptron network wins out on is that it is quite general, in that it will do a good job on many types of distributions, and it is computationally efficient.

Task 4

4a)

The learning rate is, as mentioned previously, a parameter that decides the size of the step we take along the gradient of the cost function when updating the weights. A high learning rate means that we take a longer step along the gradient, and vice versa. If we choose a learning rate which is too low, then we take too long to reach the optimal solution, and we have a higher chance of getting stuck in a local minima. If we set the learning rate to be too high we risk never getting the optimal solution, and we can end up 'climbing up' the minima of the cost function. The learning rate that is just right is when we steadily approach the minima of the cost function in a reasonable amount of epochs.

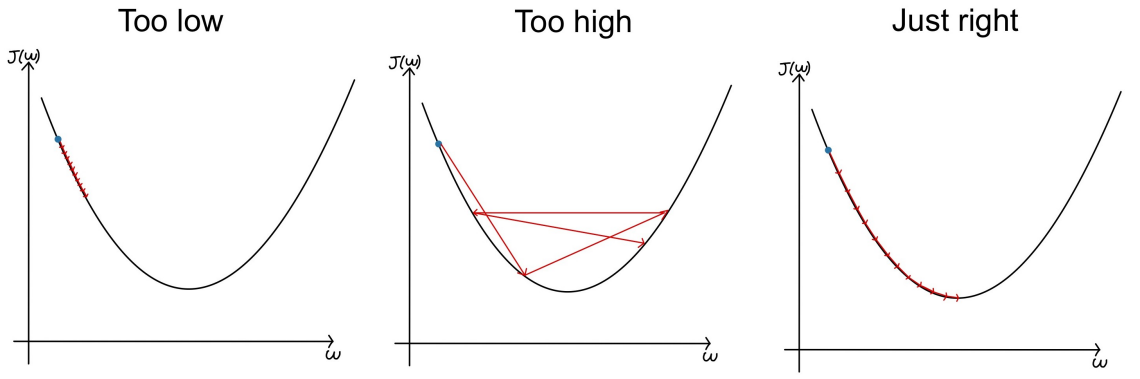


Figure 10: Gradient descent with different learning rates

Figure 10 is an attempt to illustrate the importance of choosing a correct learning rate. One could also choose an adaptive learning rate, which varies according to some predefined rule. Typically one will compare the cost function before and after backpropagation, and if it has increased we will decrease the learning rate and vice versa, but this adaptive learning rate has not been used here,

only the fixed learning rate. Note that figure 10 would be a simplified version of the cost function for a one dimensional feature space, in our two dimensional case the cost function can be thought of as a surface in 3d space, as opposed to the 2d function shown in the figure.

Momentum is often used in calculating the new weights, this is done by adding a new term to the updated weights which is dependant on the previous change in weights. What this does is that it smoothes out the oscillations we get on approach to the minima of the cost function, these oscillations are seen in figure 10 with high learning rate. So we get a smoothing of the oscillations, and faster convergence to the minimum. As T increases, the momentum term will decrease, which makes the smoothing effect of the momentum apparent. Since the momentum term is dependant on the change in weights in the previous iteration, momentum will also help get the network out of small local minima. The new expression for the updated weights for the jth Perceptron in layer r is given by:

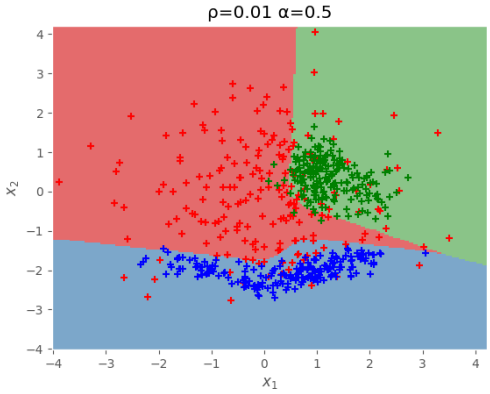
$$\boldsymbol{w}_j^r(\text{new}) = \boldsymbol{w}_j^r(\text{old}) + \alpha\Delta\boldsymbol{w}_j^r(\text{old}) - \rho\sum_i^N \delta_j^r(i)\boldsymbol{y}^{r-1}(i)$$

$\Delta\boldsymbol{w}_j^r(\text{old})$ here refers to the last two terms in the equation, for the previous iteration.
The final term is the 'gradient descent term', which is the gradient of the cost function multiplied by the learning rate ρ. The second term is the momentum term, which is the previous update in the weights, multiplied by α. α is the momentum factor, it is just a number between 0 and 1 which defines how much 'impact' the momentum factor has on the training of the network.

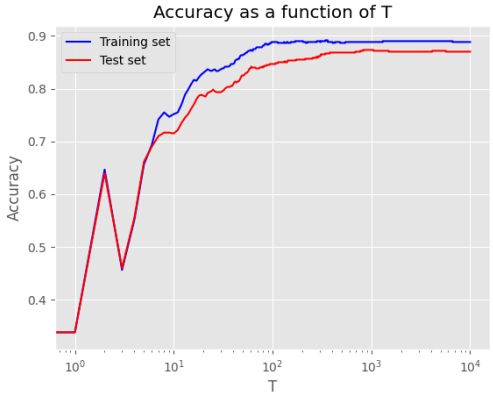
4b)

(Note: A quick explanation of the code used for task 4 is given at the end of this section.)

To see the effect of network complexity we can create different architectures and analyze the resulting decision boundaries. These following figures show the results of classifying the data sets with learning rate set to 0.01, momentum factor set to 0.5. For simpler notation the architecture of the neural network is written as a list where each number in the list represents the number of nodes in the respective layer, e.g. [4,3] means two hidden layers with 4 nodes in the first layer and 3 nodes in the second layer. The output layer with 3 output nodes comes in addition to these (As well as the input layer), the list only refers to the hidden layers.

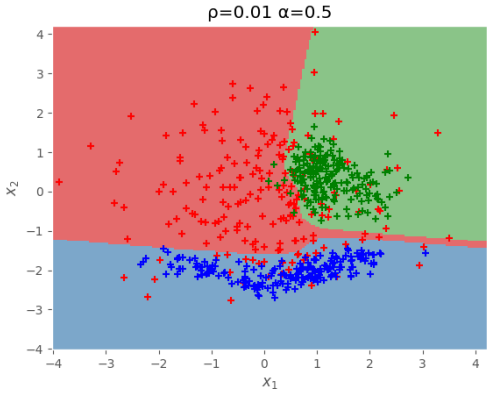


(a) Decision boundary for network with one hidden layer with 3 nodes

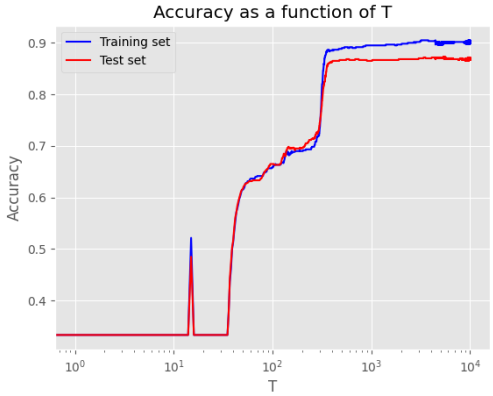


(b) Accuracy vs Epoch for network with one hidden layer with 3 nodes

Figure 11: [3]

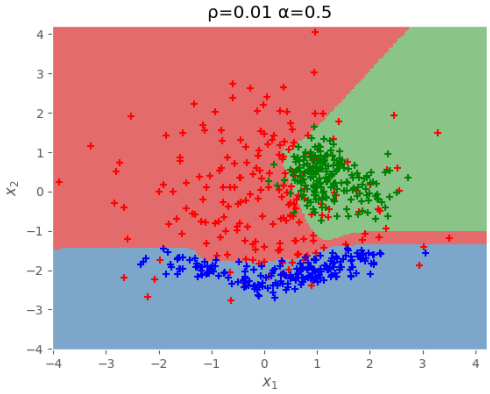


(a) Decision boundary for network with three hidden layers with 3 nodes each

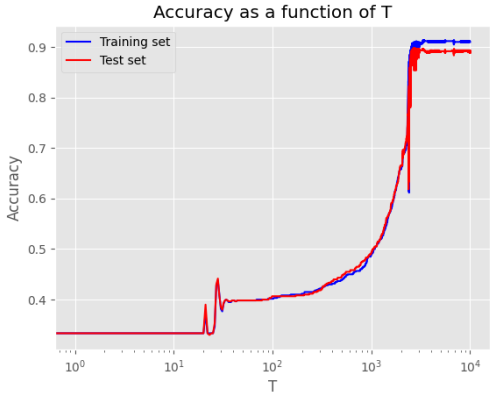


(b) Accuracy vs Epoch for network with three hidden layers with 3 nodes each

Figure 12: [3, 3, 3]

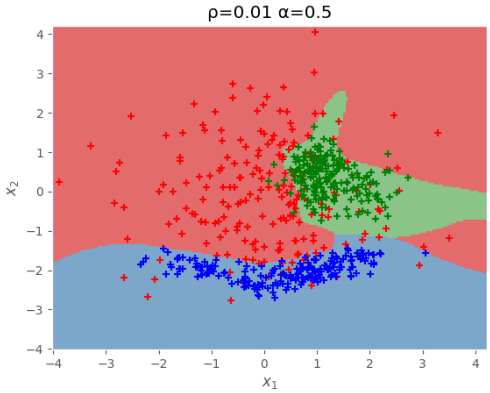


(a) Decision boundary for network with five hidden layers with 3 nodes each

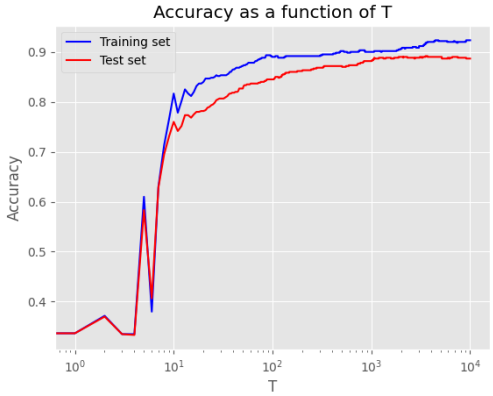


(b) Accuracy vs Epoch for network with five hidden layers with 3 nodes each

Figure 13: [3, 3, 3, 3, 3]



(a) Decision boundary for network with one hidden layer with 15 nodes



(b) Accuracy vs Epoch for network with one hidden layers with 15 nodes

Figure 14: [15]

Multi-layer Perceptron network			
Architecture	Data set	Accuracy	Misclassifications
[3]	Training	88.8%	67
	Test	87.0%	78
[3, 3, 3]	Training	90.0%	60
	Test	86.7%	80
[3, 3, 3, 3, 3]	Training	91.0%	54
	Test	88.8%	67
[15]	Training	92.3%	46
	Test	88.7%	68

Table 4: Classification results for different network complexities

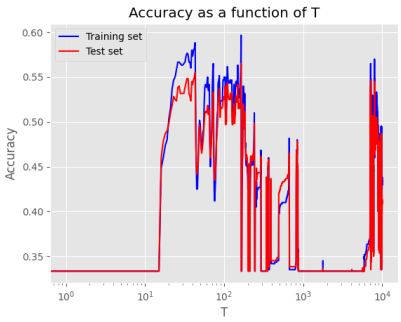
We can learn a lot by viewing these figures. The most obvious conclusion we can draw comes from the complexity of the decision boundaries for the different networks. From figures 11a, 12a and 13a we can see that as the number of hidden nodes increases we get a more complex decision boundary. This makes sense if we think back to the single layer Perceptron network, where each discriminant function describes a line (hyperplane in the general l dimensional case) that separates the feature space. More nodes should, by the same logic, give more lines to divide the feature space giving us regions that are more complex, and not just separated by single lines. Meaning we have a network that can separate classes that are not linearly separable. By looking at the accuracies of the network as a function of T we can see the accuracy steadily increases, and after a while reaches what appears to be an asymptotic value. This tells us two things, first that the network has found a minima and is 'stuck' there, thus the accuracy wont increase much with more epoch. The other thing we can learn from this plot is that our learning rate is quite good, if it were too low we would see a shallower rise in accuracy, and a higher learning rate would give us more erratic behaviour. One very interesting thing we can see in many of the figures, e.g. fig 12b is that we can clearly see that the network ended up in a small local minima, and then climb out of that local minima. It is very interesting to actually see that this happens in our code, fig 11b is especially interesting, since here we can see that the accuracy jumped to 66% only to drop down again right after. What this means is that the network found a solution where two classes (probably 2 and 3) are classified accurately, but no vectors in class 1 are correctly classified. Then the network is able to get out of this local minima and find a much better solution.

Now we know that the decision boundary gets more complex with more hidden nodes, but what about how they are distributed in the hidden layers? To answer this we can look at figures 13a and 14a. These networks have the same amount of nodes, but distributed differently, and the resulting decision boundaries are very different. It is clear that the decision boundary in figure 14a is more complex than figure 13a, even though the number of hidden nodes is the same. This points to having many nodes in one layer causes further complexity in the decision boundary.

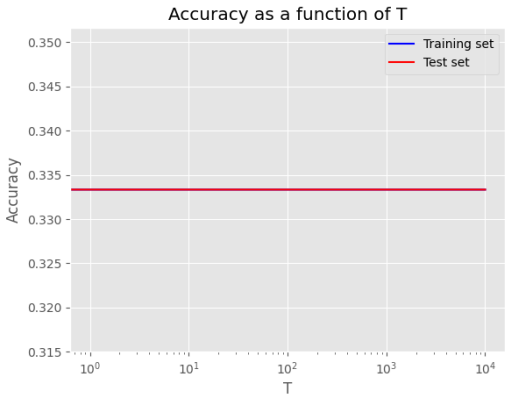
The fact that the complexity of the decision boundary increases as we increase the number of nodes means that we run the risk of overfitting if we have to many nodes and many epochs. So we need to be aware of how the neural network is designed.

4c)

For this task the architecture of the network will be set to have only one hidden layer with 15 nodes, the momentum factor has been set to 0.5 for all figures. Since the accuracy vs epoch plot is most interesting for comparing the learning rates, this plot will be the only shown in this section to save space and time, but the accompanying plots, as well as for more values of ρ will be listed in the appendix.



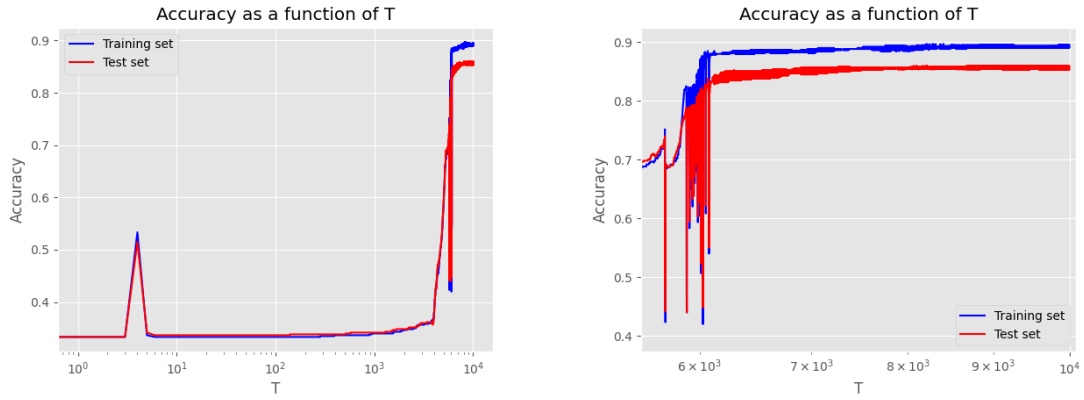
(a) Accuracy vs epochs for $\rho = 1$



(b) Accuracy vs epochs for $\rho = 10^{-6}$

Figure 15a is an excellent example of what happens when the learning rate is set too high. Here we can see very erratic behaviour as the classifier is 'bouncing around' the cost function. There is no convergence taking place here, only jumping around erratically. The accuracy on the test set for these parameters was only 41.0%, which is extremely low compared to what we got with a good learning rate.
Figure 15b shows the complete other end of the spectrum. Here the learning rate is so low that the change in the weights being made each iteration is too small to make any progress towards a solution. That is why we see no improvement over 10 000 epochs for this classifier, it is just stuck in the case where all feature vectors are classified as the same class.
These two cases are of course very extreme, for more 'moderate' values of ρ we would expect to see a steeper gradient in the accuracy for a higher learning rate, until we reach the point where we see the erratic behaviour shown in figure 15a. Both of these figures match what we would expect to see, and they both match figure 10.

We can also take a look at how the momentum factor affects the accuracy as a function of epochs. As mentioned previously the momentum dampens the oscillations that happen near the minima of the cost function, thus speeding up convergence. So we should expect to see this if we vary the momentum factor while keeping all other parameters constant. (Only the accuracy vs epoch will be plotted here, since the decision boundary does not contain much information in the sense of the effect of the momentum.)



(a) (b) (zoomed in)

Figure 16: Accuracy for $\alpha = 0$

Figure 16 shows us how much of an effect the momentum has on the convergence of the network. We can very clearly see a lot of oscillations here that are not present when we have momentum (such as the plots in task 4b). This shows us how important the momentum is since we don't really get any proper convergence without momentum, meaning we don't get as good a result for our classifier.

4d)

The best performing Neural Network for this task is [10, 8, 6] with $\rho = 0.008$ and $\alpha = 0.65$. This design was found by trial and error. Ideally a technique for optimizing the design should have been used, like e.g. pruning, but this was not implemented due to time. The result for this design is:

Multi-layer Perceptron		
Data set	Accuracy	Misclassifications
Training	92.5%	45
Test	90.2%	59

Table 5: Result for optimal multi-layer Perceptron

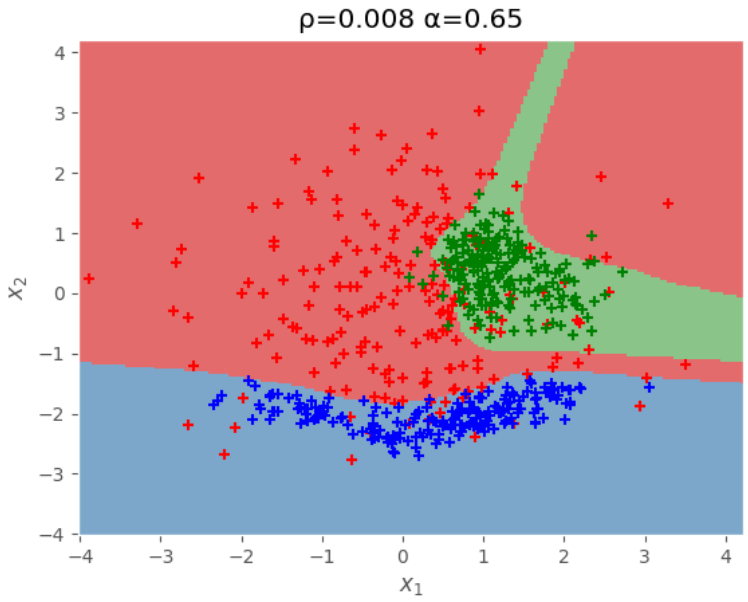
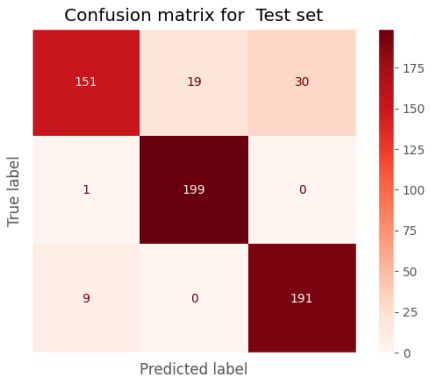
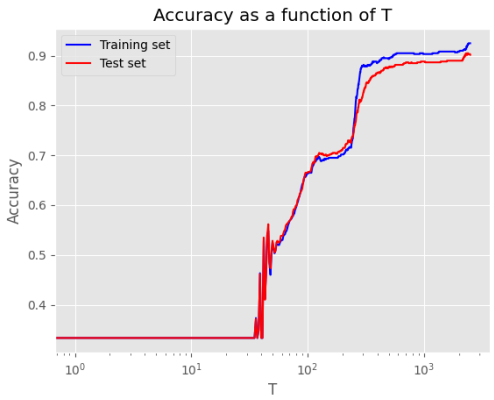


Figure 17: Decision boundary for optimal neural network



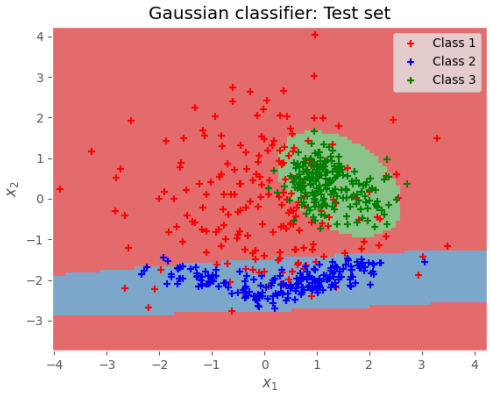
(a) Confusion matrix for optimal neural network



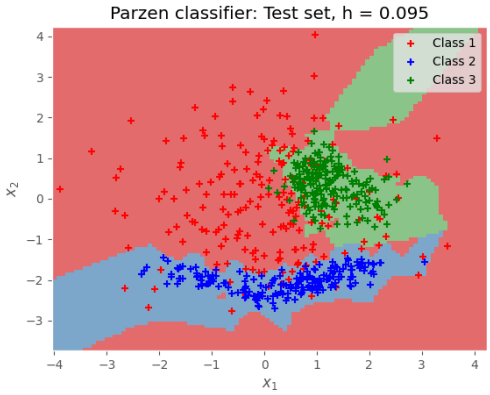
(b) Accuracy plot for optimal neural network

As we can see this network performs extremely well, getting an accuracy of 90.2% on the test set after 2500 epochs. This design performs so well since it has many hidden nodes, so it can create more complex decision regions, but not too much complexity as to cause overfitting. Also the parameters (ρ, α) are tuned for this network to perform optimally.
There are probably more optimal designs for this network, but in the short time span of this exam no better design was found. One limitation in finding a better design is the computation time for this network, especially with many nodes this network takes quite a long time to run. The most

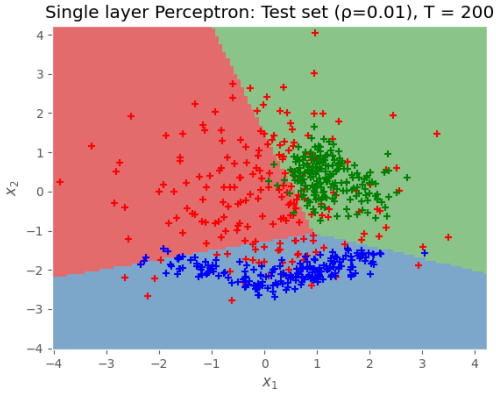
optimal solution possible with such a network is probably a few percent better than this design (speculation).
This result is the best of all the classifiers used on this data set, by a decent margin. The training set had a higher accuracy with the Parzen classifier, but obviously we don't care as much about this since this was when we got overfitting, causing the classification of the test set to drop drastically. To properly compare all classifiers we can compare their decision boundaries, as well as their performance:



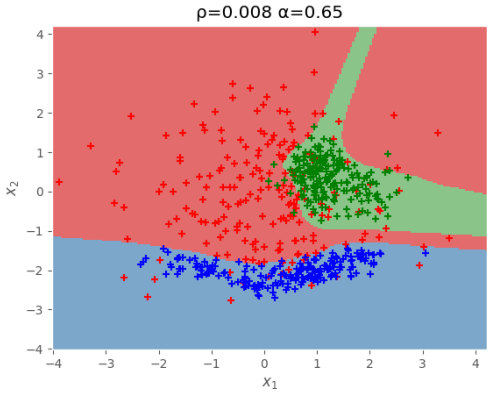
(a) Gaussian classifier



(b) Parzen classifier



(c) Single layer Perceptron



(d) Multi-layer Perceptron network

Figure 19: Decision boundaries for all classifiers

Comparison of classifiers				
Classifier	Data set	Accuracy	Misclassifications	Time (s)
Gaussian	Training	89.5%	63	$1.42 * 10^{-2}$
	Test	86.8%	79	
Parzen	Training	94.3%	34	$1.936 * 10^{0}$
	Test	87.3%	76	
Perceptron	Training	86.7%	80	$3.344 * 10^{-2}$
	Test	81.0%	115	
MLP-network	Training	92.5%	45	$2.016 * 10^{1}$
	Test	90.2%	59	

Table 6: Comparison of all classifiers used on the data set

Here we can clearly see performance of all the classifiers and compare them properly against each other. The most accurate classifier is as mentioned the Multi layer neural network, with an accuracy of 90.2%. The decision boundary for this classifier is a bit complicated, especially when compared to the Gaussian classifier and the single layer Perceptron. The second best classifier is the Parzen classifier, followed closely by the Gaussian, and then the single layer Perceptron trails a bit behind. The dfference in number of misclassified feature vectors for the three best classifiers is not that really that big at only 20 feature vectors. However the data set is pretty small, so we should look at the percentage difference, and a 2.9% difference (between MLP and Parzen) is quite a bit. Another very interesting point of comparison is the computation time for each classifier. We can see in table 6 that the single layer Perceptron and Gaussian classifier are by far the quickest, both taking on the order of 10ms to classify the test set. The Parzen classifier takes quite a bit more time, 2 orders of magnitude more, taking roughly 2 seconds to classify the data set. Then comes the MLP-network, using approximately 20 seconds to classify the test data. This is a huge difference in time for the classifiers, the MLP-network is over 1000 times slower than the Gaussian classifier. For this relatively small data set the difference doesn't matter that much, but if the data set was much larger, and the network more complicated, then this difference in magnitude is huge. 3 orders of magnitude doesn't necessarily sound like much, but it is the difference of running the code for 1 day and running it for around 3 years. So computation time is really important, and especially for large data sets one has to take computation time into account when designing the classifier.

The easiest classifier to code was by far the Gaussian ML classifier, this classifier is about as easy as it gets and is very straight forward to implement. The Parzen classifier was also relatively easy, but trickier than the Gaussian classifier. The multi-layer network was by far the hardest to create, mainly because the backpropagation algorithm is very tricky to properly implement due to its complex nature, especially since there are so many indices to keep track of. The multi-layer network is also a bit more tricky to use since there are more free parameters, and if one doesn't know how they work then it can be tricky to get good results in the beginning. My personal favorite is the multi-layer neural network. The reason being that it is very general, and works on many types of classification problems. Also it was extremely satisfying to get it working. The single layer network is in a close second place, mainly since the concept of finding the ideal line (hyperplane) to separate the feature vectors is so simple, yet very effective. Also the concept that these two networks 'learn' is very fascinating and super interesting.

The code for the MLP-network

The multi-layer Perceptron network was coded in a similar way to the single layer network in task 3. First a class for the Perceptrons was created, these Perceptrons have all the necessary attributes and methods for both forward, and backward propagation. A new class for the output Perceptrons, which inherits from the Perceptron class was also made which is almost the same, but contains different methods for calculating the error and delta. This was done since the expressions for calculating these values are different for the output layer Perceptrons, than for the hidden layers. Then a class for the network itself is created, which contains the Perceptrons, stored in a nested list to keep track of which Perceptrons are in which layer. The forward- and backpropagation are done as methods in this class which implements the respective algorithms as they are explained in the texbook/ lecture notes. The most tricky part here is to keep track of which node is which, and not messing up the indices (I won't go into more detail on the backpropaation since it is a straightforward implementation of the steps laid out in the textbook). The training, prediction etc. is done in the same way as in task 3. A simplified UML diagram of the code used is given here:

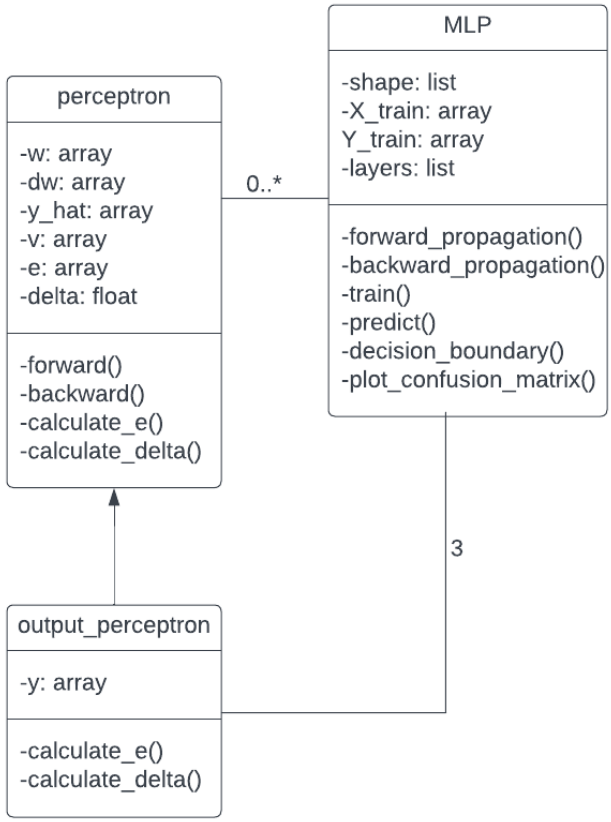
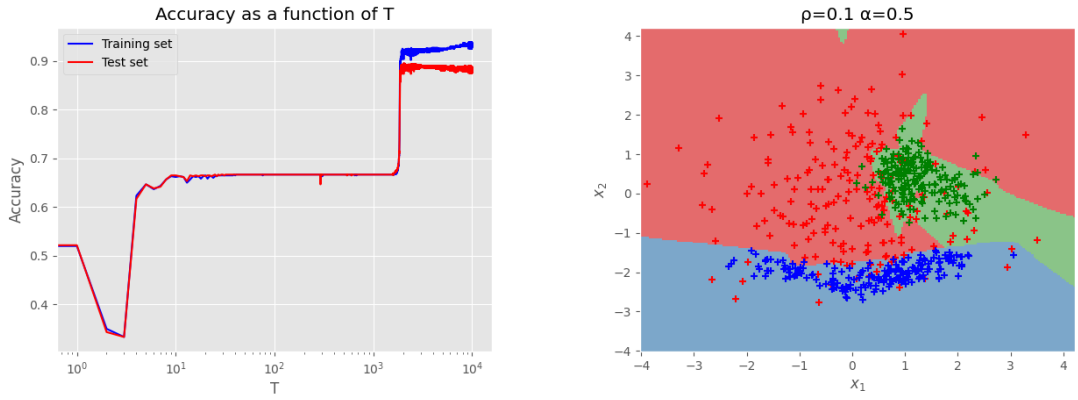


Figure 20: UML diagram of code used in task 4

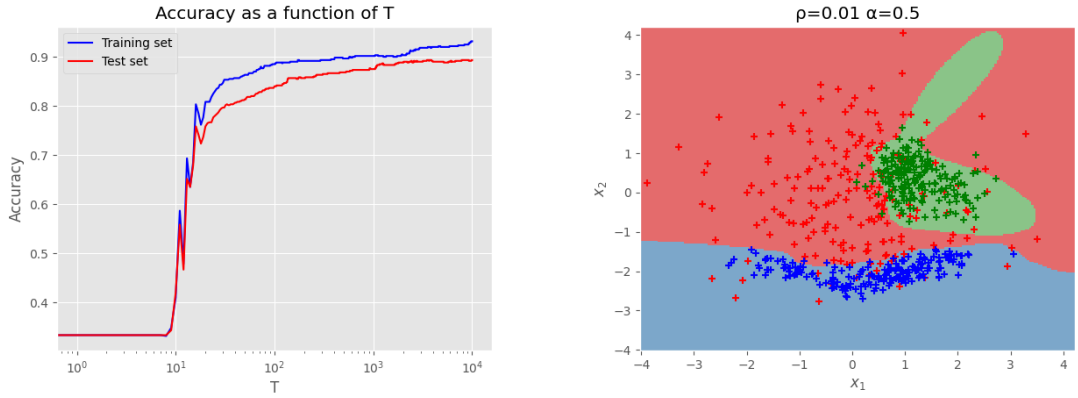
Appendix

Additional plots for task 4c



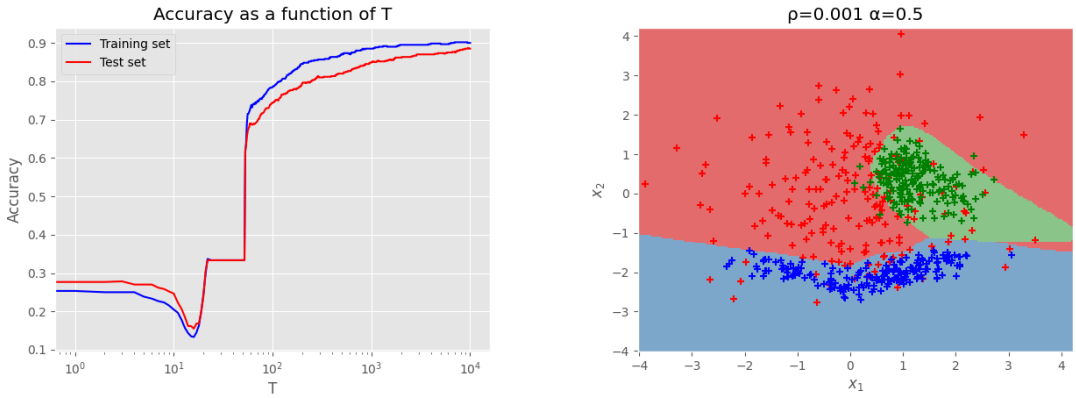
(a) (b)

Figure 21: Accuracy and decision boundary for $\rho = 0.1$



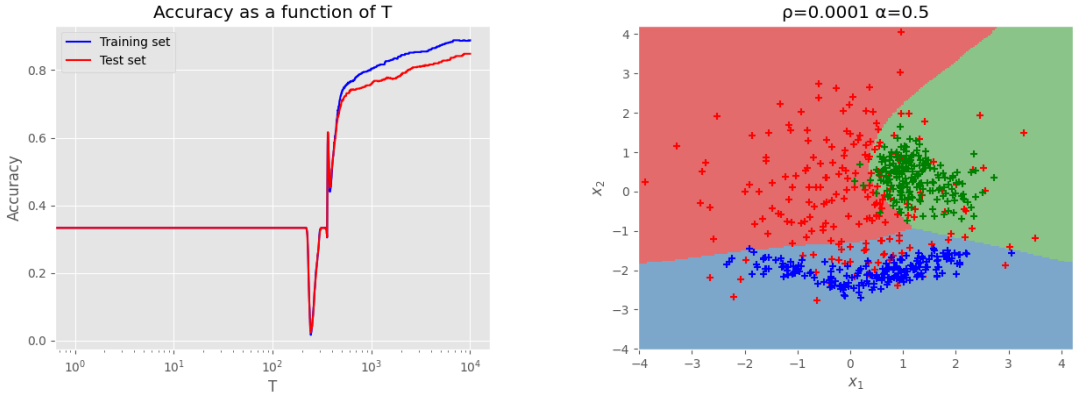
(a) (b)

Figure 22: Accuracy and decision boundary for $\rho = 0.01$



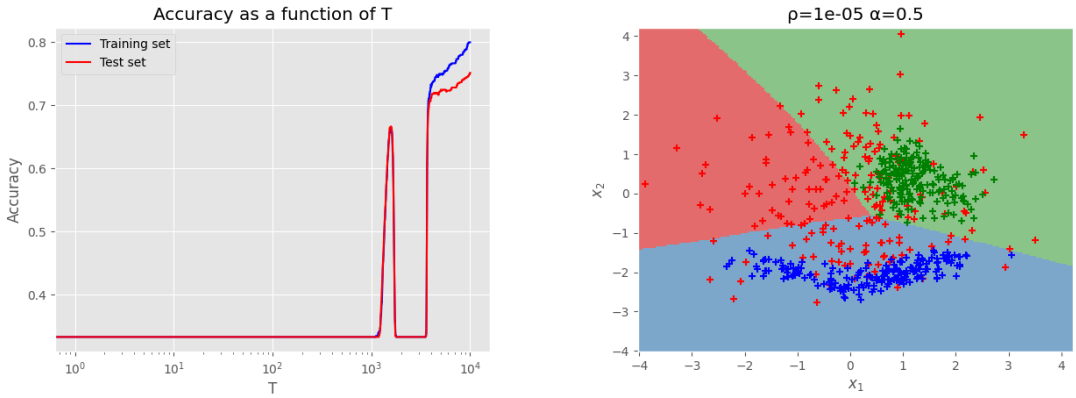
(a) (b)

Figure 23: Accuracy and decision boundary for $\rho = 0.001$



(a) (b)

Figure 24: Accuracy and decision boundary for $\rho = 0.0001$



(a) (b)

Figure 25: Accuracy and decision boundary for $\rho = 0.00001$

Multi-layer network ([15]) ($\alpha = 0.5$)			
Learning rate	Data set	Accuracy	Misclassifications
10^{-6}	Training	33.3%	400
	Test	33.3%	400
10^{-5}	Training	80.0%	120
	Test	75.2%	150
10^{-4}	Training	88.8%	67
	Test	84.8%	91
10^{-3}	Training	90.0%	60
	Test	88.5%	69
10^{-2}	Training	93.2%	41
	Test	89.3%	64
10^{-1}	Training	93.3%	40
	Test	88,7%	68
0.5	Training	48.3%	310
	Test	48.8%	307
1	Training	43.0%	342
	Test	41.0%	354

Table 7: Result of neural network with varying learning rate

Code

(All the code used for this exam is also available in a zipped folder delivered along with this PDF)

Tasks 1 & 2

```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
import scipy as sc
from sklearn import metrics
from prettytable import PrettyTable

# Importing the data
dataset = sc.io.loadmat('Home-exam\ExamData3D.mat')

# Splitting the data into training and test sets
X_train = dataset['X_train']
X_test = dataset['X_test']
Y_train = dataset['Y_train'][0]
Y_test = dataset['Y_test'][0]

# defining the probability for the classes
P = 1/3 # using the a priori knowledge that the classes are equiprobable (to do properly it should be calculated from the data set)

def mean_vector(x, y, class_nr):
    """
    Calculates the mean vector for a given class

    Parameters
    ----------
    - x : The set of feature vectors
    - y: Known labels for the feature vectors
    - class_nr: The class for which the mean vector is calculated

    Returns
    -------
    - mu: The mean vector for the given class
    """

    # special case for 1 dimension (only difference is the indexing of the array)
    if x.ndim == 1:
        mu = np.mean(x[y == class_nr]) # calculating the mean vector for the given class
    else:
        mu = np.mean(x[:,y == class_nr], axis = 1) # calculating the mean vector for the given class

    return np.matrix(mu).T # returning the mean vector as a column matrix

def J3_score(x, y):
    """
    Calculates the modified J3 score for a given data set

    Parameters
    ----------
```

```
    - x: The set of feature vectors
    - y: Known labels for the feature vectors

    Returns
    -------
    - J3: The modified J3 score for the given data set
    """

    # calculating the mean vector for each class
    mu1 = mean_vector(x, y, 1)
    mu2 = mean_vector(x, y, 2)
    mu3 = mean_vector(x, y, 3)

    # calculating the global mean vector
    mu0 = np.matrix((mu1 + mu2 + mu3)*P)

    # calculating the covariance matrix for each class
    if x.ndim == 1:
        cov1 = np.cov(np.array((x[y == 1])))
        cov3 = np.cov(np.array((x[y == 3])))
        cov2 = np.cov(np.array((x[y == 2])))
    else:
        cov1 = np.cov(np.array((x[:,y == 1])))
        cov3 = np.cov(np.array((x[:,y == 3])))
        cov2 = np.cov(np.array((x[:,y == 2])))

    # calculating the within class scatter matrix
    Sw = (cov1 + cov2 + cov3)*P

    # calculating the between class scatter matrix
    Sb  = ((mu1-mu0)@((mu1-mu0).T) + (mu2-mu0)@((mu2-mu0).T) + (mu3-mu0)@((mu3-mu0).
    T))*P

    # calculating the modified J3 score
    if x.ndim == 1:
        J3 = Sb/Sw
    else:
        J3 = np.trace(np.linalg.inv(Sw)@Sb)
    return J3

def print_J3():
    """
    Prints the J3 score for each dimension and combination of dimensions
    """
    print("------------Task 1------------")
    table = PrettyTable()
    table.field_names = ["Dimension", "J3 score"]
    table.add_row(["[1]", np.round(float(J3_score(X_train[0], Y_train)), 3)])
    table.add_row(["[2]", np.round(float(J3_score(X_train[1], Y_train)), 3)])
    table.add_row(["[3]", np.round(float(J3_score(X_train[2], Y_train)), 3)])
    table.add_row(["[1,2]", np.round(float(J3_score(np.delete(X_train, 2, 0),
    Y_train)), 3)])
    table.add_row(["[1,3]", np.round(float(J3_score(np.delete(X_train, 1, 0),
    Y_train)), 3)])
    table.add_row(["[2,3]", np.round(float(J3_score(np.delete(X_train, 0, 0),
    Y_train)), 3)])
    table.add_row(["[1,2,3]", np.round(float(J3_score(X_train, Y_train)), 5)])
    print(table)
```

```
103
104 def plot_data():
105     """
106     Plots all combinations of dimensions of the training set
107     """
108     plt.subplot(1,3,1)
109     plt.hist(X_train[0][Y_train == 1], bins = 20, alpha = 0.5, label = 'Class 1',
        color = 'red')
110     plt.hist(X_train[0][Y_train == 2], bins = 20, alpha = 0.5, label = 'Class 2',
        color = 'blue')
111     plt.hist(X_train[0][Y_train == 3], bins = 20, alpha = 0.5, label = 'Class 3',
        color = 'green')
112     plt.legend()
113     plt.title('Dimension: [1], $J_3$ = ' + str(np.round(float(J3_score(X_train[0],
        Y_train)), 5)))
114
115     plt.subplot(1,3,2)
116     plt.hist(X_train[1][Y_train == 1], bins = 20, alpha = 0.5, label = 'Class 1',
        color = 'red')
117     plt.hist(X_train[1][Y_train == 2], bins = 20, alpha = 0.5, label = 'Class 2',
        color = 'blue')
118     plt.hist(X_train[1][Y_train == 3], bins = 20, alpha = 0.5, label = 'Class 3',
        color = 'green')
119     plt.legend()
120     plt.title('Dimension: [2], $J_3$ = ' + str(np.round(float(J3_score(X_train[1],
        Y_train)), 5)))
121
122     plt.subplot(1,3,3)
123     plt.hist(X_train[2][Y_train == 1], bins = 20, alpha = 0.5, label = 'Class 1',
        color = 'red')
124     plt.hist(X_train[2][Y_train == 2], bins = 20, alpha = 0.5, label = 'Class 2',
        color = 'blue')
125     plt.hist(X_train[2][Y_train == 3], bins = 20, alpha = 0.5, label = 'Class 3',
        color = 'green')
126     plt.legend()
127     plt.title('Dimension: [3], $J_3$ = ' + str(np.round(float(J3_score(X_train[2],
        Y_train)), 5)))
128     plt.show()
129
130     # plotting the 2D data
131     plt.subplot(1,3,1)
132     plt.scatter(X_train[0][Y_train == 1], X_train[1][Y_train == 1], label = 'Class 1
        ', color = 'red')
133     plt.scatter(X_train[0][Y_train == 2], X_train[1][Y_train == 2], label = 'Class 2
        ', color = 'blue')
134     plt.scatter(X_train[0][Y_train == 3], X_train[1][Y_train == 3], label = 'Class 3
        ', color = 'green')
135     plt.legend()
136     plt.title('Dimensions: [1,2], $J_3$ = ' + str(np.round(float(J3_score(np.delete(
        X_train, 2, 0), Y_train)), 5)))
137
138     plt.subplot(1,3,2)
139     plt.scatter(X_train[0][Y_train == 1], X_train[2][Y_train == 1], label = 'Class 1
        ', color = 'red')
140     plt.scatter(X_train[0][Y_train == 2], X_train[2][Y_train == 2], label = 'Class 2
        ', color = 'blue')
141     plt.scatter(X_train[0][Y_train == 3], X_train[2][Y_train == 3], label = 'Class 3
        ', color = 'green')
```

```
    plt.legend()
    plt.title('Dimensions: [1,3], $J_3$ = ' + str(np.round(float(J3_score(np.delete(
    X_train, 1, 0), Y_train)), 5)))

    plt.subplot(1,3,3)
    plt.scatter(X_train[1][Y_train == 1], X_train[2][Y_train == 1], label = 'Class 1
    ', color = 'red')
    plt.scatter(X_train[1][Y_train == 2], X_train[2][Y_train == 2], label = 'Class 2
    ', color = 'blue')
    plt.scatter(X_train[1][Y_train == 3], X_train[2][Y_train == 3], label = 'Class 3
    ', color = 'green')
    plt.legend()
    plt.title('Dimensions: [2,3], $J_3$ = ' + str(np.round(float(J3_score(np.delete(
    X_train, 0, 0), Y_train)), 5)))
    plt.show()
    plt.clf()

    print("The difference between dimensions 1,2 and 3d is: ",
        np.round(float(J3_score(X_train, Y_train))-float(J3_score(np.delete(X_train,
     2, 0), Y_train)), 5),
        "which is so small that we can get rid of dimensin 3 without losing much
    information.")

class Gaussian:
    def __init__(self, X_train, Y_train):
        """Gaussian classifyer for 3 classes"""
        self.X_train = X_train
        self.Y_train = Y_train
        self.accuracy = 0

    def mean_vector(self):
        """Returns the mean vector for each class"""
        mean_1 = mean_vector(X_train, Y_train, 1)
        mean_2 = mean_vector(X_train, Y_train, 2)
        mean_3 = mean_vector(X_train, Y_train, 3)
        return mean_1, mean_2, mean_3

    def covariance_matrix(self):
        """Returns the covariance matrix for each class"""
        cov_1 = np.cov(np.array((X_train[:,Y_train == 1])))
        cov_2 = np.cov(np.array((X_train[:,Y_train == 2])))
        cov_3 = np.cov(np.array((X_train[:,Y_train == 3])))
        return cov_1, cov_2, cov_3

    def pdf(self, x):
        """
        Calculates the probability density function for each class

        Parameters
        ----------
        - x: The set of feature vectors

        Returns
        -------
        - p1, p2, p3: The probability density function for each class
        """

        # calculating the mean vector and covariance matrix for each class
```

```
        mean_1, mean_2, mean_3 = self.mean_vector()
        cov_1, cov_2, cov_3 = self.covariance_matrix()

        # calculating the probability for each class (using the multivariate gaussian distribution)
        denominator = (2*np.pi)**(len(mean_1)/2)*np.linalg.det(cov_1)**(1/2)
        exponent = -0.5*(x-mean_1).T@np.linalg.inv(cov_1)@(x-mean_1)
        p1 = np.diag(np.exp(exponent))/denominator # note: we take the diagonal since the other values don't matter for us

        denominator = (2*np.pi)**(len(mean_2)/2)*np.linalg.det(cov_2)**(1/2)
        exponent = -0.5*(x-mean_2).T@np.linalg.inv(cov_2)@(x-mean_2)
        p2 = np.diag(np.exp(exponent))/denominator

        denominator = (2*np.pi)**(len(mean_3)/2)*np.linalg.det(cov_3)**(1/2)
        exponent = -0.5*(x-mean_3).T@np.linalg.inv(cov_3)@(x-mean_3)
        p3 = np.diag(np.exp(exponent))/denominator

        return p1, p2, p3

    def predict(self, X, Y, accuracy = True):
        """
        Predicts the class for each feature vector in X

        Parameters
        ----------
        - X: The set of feature vectors
        - Y: Known labels for the feature vectors
        - accuracy: If True, the accuracy of the prediction is calculated and stored in self.accuracy

        Returns
        -------
        - pred: The predicted class for each feature vector in X
        """
        p1, p2, p3 = self.pdf(X)
        pred = np.argmax(np.array([p1, p2, p3]), axis = 0) + 1
        if accuracy == True:
            self.accuracy = np.sum(pred == Y)/len(Y)
        return pred

    def plot(self, X, Y, title, resolution = 100):
        """
        Plots the dataset and the decision boundary

        Parameters
        ----------
        - X: The set of feature vectors
        - Y: Known labels for the feature vectors
        - title: The title of the plot (Gaussian classifier for "title")
        - resolution: The resolution of the plot
        """
        x, y = np.linspace(-4.0, 4.2, num=resolution), np.linspace(-3.7, 4.2, num=resolution)
        XY = np.asarray(np.meshgrid(x,y)).reshape(2, -1)

        pred = self.predict(XY, Y, accuracy = False)

```

```
        plt.pcolormesh(x, y, pred.reshape(resolution,resolution), cmap = 'Set1', alpha=0.6, vmin=0, vmax=12)
        plt.scatter(X[0][Y == 1], X[1][Y == 1],marker='+', label = 'Class 1', color = 'red')
        plt.scatter(X[0][Y == 2], X[1][Y == 2],marker='+', label = 'Class 2', color = 'blue')
        plt.scatter(X[0][Y == 3], X[1][Y == 3],marker='+', label = 'Class 3', color = 'green')
        plt.xlabel('$x_1$')
        plt.ylabel('$x_2$')
        plt.legend()
        plt.title('Gaussian classifier:' + title)
        plt.show()

    def plot_confusionmatrix(self, X, Y, title):
        pred = self.predict(X, Y, accuracy = False)
        confusion_matrix = metrics.confusion_matrix(Y, pred)
        confusion_matrix_display = metrics.ConfusionMatrixDisplay(confusion_matrix, display_labels = [1,2,3])
        confusion_matrix_display.plot(cmap="Reds")
        plt.title("Gaussian classifier: " + title)
        plt.grid(False)
        plt.show()
        plt.clf()

class Parzen:
    def __init__(self, X_train, Y_train):
        """Parzen window density estimator for 3 classes"""
        self.X_train = X_train
        self.Y_train = Y_train
        self.accuracy = 0

    def pdf(self, x, h):
        """
        Calculates the probability density function for each class

        Parameters
        ----------
        - x: The set of feature vectors
        - h: The window size

        Returns
        -------
        - p1, p2, p3: The probability density function for each class
        """

        # separating the training data into the respective classes
        X1 = self.X_train[:,Y_train == 1]
        X2 = self.X_train[:,Y_train == 2]
        X3 = self.X_train[:,Y_train == 3]

        N = len(X1[0]) # number of feature vectors in each class
        l = len(X1) # number of dimensions

        denominator = ((2*np.pi)**(l/2))*h**l # denominator of the probability density function
        p1, p2, p3 = np.zeros(len(x[0])), np.zeros(len(x[0])), np.zeros(len(x[0])) # initializing empty arrays
```

```
        for i in range(N):
            exponent1 = np.diag(-((x.T-X1[:,i]) @ (x.T-X1[:,i]).T))/(2*h**2) # exponent of the probability density function for class 1
            p1 += np.exp(exponent1)/(denominator*N) # adding the probability density function for each feature vector

            exponent2 = np.diag(-((x.T-X2[:,i]) @ (x.T-X2[:,i]).T)/(2*h**2))
            p2 += np.exp(exponent2)/(denominator*N)

            exponent3 = np.diag(-((x.T-X3[:,i]) @ (x.T-X3[:,i]).T)/(2*h**2))
            p3 += np.exp(exponent3)/(denominator*N)

        return p1, p2, p3

    def predict(self, X, Y, h, accuracy = True):
        """
        Predicts the class for each feature vector in X

        Parameters
        ----------
        - X: The set of feature vectors
        - Y: Known labels for the feature vectors
        - h: The window size
        - accuracy: If True, the accuracy of the prediction is calculated and stored in self.accuracy

        Returns
        -------
        - pred: The predicted class for each feature vector in X
        """

        p1, p2, p3 = self.pdf(X, h) # getting the pdfs
        pred = np.argmax(np.array([p1, p2, p3]), axis = 0) + 1 # predicting the class for each feature vector
        if accuracy == True:
            self.accuracy = np.sum(pred == Y)/len(Y) # calculating the accuracy
        return pred

    def plot(self, X, Y, h, title, resolution = 50):
        """
        Plots the dataset and the decision boundary

        Parameters
        ----------
        - X: The set of feature vectors
        - Y: Known labels for the feature vectors
        - h: The window size
        - title: The title of the plot (Parzen window density estimator for "title", h = "h")
        - resolution: The resolution of the plot

        Returns
        -------
        - pred: The predicted class for each feature vector in X
        """

        x, y = np.linspace(-4.0, 4.2, num=resolution), np.linspace(-3.7, 4.2, num=resolution) # creating the grid
```

```
        XY = np.asarray(np.meshgrid(x,y)).reshape(2, -1)

        pred = self.predict(XY, Y, h, accuracy = False) # predicting the class for each point on the grid

        plt.pcolormesh(x, y, pred.reshape(resolution,resolution), cmap = 'Set1', alpha=0.6, vmin=0, vmax=12)
        plt.scatter(X[0][Y == 1], X[1][Y == 1], marker = '+', label = 'Class 1', color = 'red')
        plt.scatter(X[0][Y == 2], X[1][Y == 2], marker = '+', label = 'Class 2', color = 'blue')
        plt.scatter(X[0][Y == 3], X[1][Y == 3], marker = '+', label = 'Class 3', color = 'green')
        plt.xlabel('$x_1$')
        plt.ylabel('$x_2$')
        plt.legend()
        plt.title('Parzen classifier:' + title + ', h = ' + str(h))
        plt.show()

    def plot_confusionmatrix(self, X, Y, h, title):
        """
        Plots the confusion matrix

        Parameters
        ----------
        - X: The set of feature vectors
        - Y: Known labels for the feature vectors
        - h: The window size
        - title: The title of the plot (Confusion matrix for "title", h = "h")
        """
        pred = self.predict(X, Y, h, accuracy = False)
        confusion_matrix = metrics.confusion_matrix(Y, pred)
        confusion_matrix_display = metrics.ConfusionMatrixDisplay(confusion_matrix, display_labels = [1,2,3])
        confusion_matrix_display.plot(cmap="Reds")
        plt.title("Parzen classifier: " + title + ", h = " + str(h))
        plt.grid(False)
        plt.show()

if __name__ == "__main__":
    # task 1
    print_J3() # printing the J3 scores
    plot_data() # plotting the training data

    # removing dimension 3 from the training and test sets
    X_train = np.delete(X_train, 2, 0)
    X_test = np.delete(X_test, 2, 0)

    # task 2
    print("------------Task 2------------")

    # Gaussian classifier
    # plotting the results
    gaussian = Gaussian(X_train, Y_train)
    gaussian.plot(X_train, Y_train, 'Training set')
    gaussian.plot(X_test, Y_test, ' Test set')
    gaussian.plot_confusionmatrix(X_test, Y_test, 'Test set')

```

```
    # printing the results
    table = PrettyTable()
    table.title = "Gaussian classifier"
    table.field_names = ["Data set", "Accuracy", "Misclassifications"]
    gaussian.predict(X_train, Y_train, accuracy = True)
    table.add_row(["Training set", np.round(gaussian.accuracy, 3)*100, np.round(len(
    Y_train)*(1-gaussian.accuracy), 3)])
    gaussian.predict(X_test, Y_test, accuracy = True)
    table.add_row(["Test set", np.round(gaussian.accuracy, 3)*100, np.round(len(
    Y_test)*(1-gaussian.accuracy), 3)])
    print(table)

    # Parzen classifyer
    h = 0.095 # optimal (found by trial and error)
    #h = 0.001 # overfitting (extreme case)

    # plotting the results
    parzen = Parzen(X_train, Y_train)
    parzen.plot(X_test, Y_test, h, ' Test set')
    parzen.plot_confusionmatrix(X_test, Y_test, h, 'Test set')

    # printing the results
    table = PrettyTable()
    table.title = "Parzen classifier (h = " + str(h) + ")"
    table.field_names = ["Data set", "Accuracy", "Misclassifications"]
    parzen.predict(X_train, Y_train, h, accuracy = True)
    table.add_row(["Training set", np.round(parzen.accuracy, 3)*100, np.round(len(
    Y_train)*(1-parzen.accuracy), 3)])
    parzen.predict(X_test, Y_test, h, accuracy = True)
    table.add_row(["Test set", np.round(parzen.accuracy, 3)*100, np.round(len(Y_test
    )*(1-parzen.accuracy), 3)])
    print(table)
```

Task 3

```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
import scipy as sc
from sklearn import metrics

# Importing the data
dataset = sc.io.loadmat('Home-exam\ExamData3D.mat')

# Splitting the data into training and test sets
X_train = dataset['X_train']
X_test = dataset['X_test']
X_train, X_test = np.delete(X_train, 2, 0), np.delete(X_test, 2, 0)
Y_train = dataset['Y_train'][0]
Y_test = dataset['Y_test'][0]

# adding the extra 'dimension' to X_train and X_test
X_train = np.insert(X_train, 0, 1, axis=0)
X_test = np.insert(X_test, 0, 1, axis=0)

class perceptron:
    def __init__(self, c, y, N=len(X_train[0])):
        """
```

```
        Perceptron
        --------

        Parameters
        ----
        - c: class which the perceptron is trained to recognize
        - y: training set
        - N: number of training feature vectors
        """

        self.w = np.random.rand(3,1) # random initial weights
        self.y_hat = np.zeros(N)
        self.c = c
        self.y = np.zeros(N)
        self.y[y == self.c] = 1 # y = 1 if the feature vector belongs to class c, 0 else

    def forward(self, x):
        """
        Forward pass (calculates the output of the perceptron)

        Parameters
        ----
        - x: array of feature vectors
        """

        g = self.w.T @ x # discriminant function
        f = 1/(1+np.exp(-g)) # activation function
        self.y_hat = f

    def backward(self, x):
        """
        Backward pass (updates the weights of the perceptron)

        Parameters
        ----
        - x: array of feature vectors
        """

        df = 1/(1+np.exp(-self.w.T @ x))*(1-(1/(1+np.exp(-self.w.T @ x)))) # derivative of the activation function
        dw = (rho*(self.y-self.y_hat)*df @ x.T).T # weight update
        self.w += dw

class neural_network:
    def __init__(self, X_train, Y_train, X_test, Y_test):
        """
        Neural network (single layer with 3 nodes)

        Parameters
        ----
        - X_train: training set
        - Y_train: training labels
        - X_test: test set
        - Y_test: test labels
        """

        self.accuracy = 0
```

```
        self.predictions = np.zeros(len(Y_test))
        self.X_train = X_train
        self.Y_train = Y_train
        self.X_test = X_test
        self.Y_test = Y_test

        # setting up the perceptrons
        self.neurons = [perceptron(1, self.Y_train), perceptron(2, self.Y_train), perceptron(3, self.Y_train)]

    def train(self, T):
        """
        Training the neural network (updates the weights T times)

        Parameters
        ----
        - T: number of epochs
        """

        for t in range(T):
            for neuron in self.neurons:
                neuron.forward(self.X_train)
                neuron.backward(self.X_train)

    def predict(self):
        """
        Predicting the labels of the test set

        Returns
        ----
        - accuracy: accuracy of the predictions
        """

        for neuron in self.neurons:
            neuron.forward(self.X_test) # forward pass to get the predictions

        # taking the most probable class
        self.predictions = np.argmax([self.neurons[0].y_hat, self.neurons[1].y_hat, self.neurons[2].y_hat], axis=0)+1
        self.accuracy = np.sum(self.predictions == self.Y_test)/len(self.Y_test)

        return self.accuracy

    def decision_boundary(self, txt, resolution=120):
        """
        Plotting the decision boundary

        Parameters
        ----
        - txt: title of the plot
        - resolution: resolution of the decision boundary
        """

        x, y = np.linspace(-4.0, 4.2, num=resolution), np.linspace(-4.0, 4.2, num=resolution)
        XY = np.asarray(np.meshgrid(x,y)).reshape(2, -1)
        XY = np.insert(XY, 0, 1, axis=0)
        for neuron in self.neurons:
```

```
            neuron.forward(XY)
        predictions = np.argmax([self.neurons[0].y_hat, self.neurons[1].y_hat, self.neurons[2].y_hat], axis=0)+1
        plt.pcolormesh(x, y, predictions.reshape(resolution, resolution), cmap = 'Set1', alpha=0.6, vmin=0, vmax=12)
        plt.scatter(self.X_test[1, self.Y_train == 1], self.X_test[2, self.Y_train == 1], marker = '+', color='red', label='Class 1')
        plt.scatter(self.X_test[1, self.Y_train == 2], self.X_test[2, self.Y_train == 2], marker = '+', color='blue', label='Class 2')
        plt.scatter(self.X_test[1, self.Y_train == 3], self.X_test[2, self.Y_train == 3], marker = '+', color='green', label='Class 3')
        plt.title("Single layer Perceptron: " + txt)
        plt.xlabel("$x_1$")
        plt.ylabel("$x_2$")

    def plot_confusion_matrix(self, title):
        """
        Plotting the confusion matrix

        Parameters
        ----
        - title: title of the plot
        """

        confusion_matrix = metrics.confusion_matrix(self.Y_test, self.predictions[0])
        confusion_matrix_display = metrics.ConfusionMatrixDisplay(confusion_matrix)
        confusion_matrix_display.plot(cmap="Reds")
        plt.title("Single layer Perceptron: " + title)
        plt.xlabel("Predicted label")
        plt.ylabel("True label")
        confusion_matrix_display.ax_.set_xticks([])
        confusion_matrix_display.ax_.set_yticks([])
        plt.show()

if __name__ == "__main__":
    """---Testing the neural network---"""
    # defining parameters
    rho = 0.01 # learning rate
    T = np.linspace(0, 200, 201, dtype=int) # epochs

    # initiating the neural network, for both training and test sets
    NN_test = neural_network(X_train, Y_train, X_test, Y_test)
    NN_train = neural_network(X_train, Y_train, X_train, Y_train)

    # get the accuracy as a function of T
    test_accuracies = np.zeros(len(T))
    train_accuracies = np.zeros(len(T))
    for t in range(len(T)):
        #print(t)
        NN_test.train(T[t])
        NN_train.train(T[t])
        test_accuracies[t] = NN_test.predict()
        train_accuracies[t] = NN_train.predict()

    # plotting the decision boundary
    plt.subplot(1,2,1)
    NN_test.decision_boundary("Test set" + " (" + str(chr(961)) + "=" + str(rho)+ ")
```

```
    " + ", T = " + str(T[-1]))
    plt.subplot(1,2,2)
    NN_train.decision_boundary("Training set")
    plt.show()
    print("Training set: ", np.round(train_accuracies[-1], 3), ", Test set: ", np.
    round(test_accuracies[-1], 3))

    # plotting the accuracy as a function of T
    plt.plot(T, test_accuracies, label="Test set")
    plt.plot(T, train_accuracies, label="Training set")
    plt.title("Accuracy as a function of T (" + str(chr(961)) + "=" + str(rho)+ ")")
    plt.xlabel("T")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.show()
    print("Highest accuracy for test set: ", np.round(np.max(test_accuracies), 3), "
    , at T = ", np.argmax(test_accuracies)+1)

    # plotting the confusion matrix
    NN_test.plot_confusion_matrix("Test set")
    NN_train.plot_confusion_matrix("Training set")
```

Task 4

```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
import scipy as sc
from sklearn import metrics

# Importing the data
dataset = sc.io.loadmat('Home-exam\ExamData3D.mat')

# Splitting the data into training and test sets
X_train = dataset['X_train']
X_test = dataset['X_test']
X_train, X_test = np.delete(X_train, 2, 0), np.delete(X_test, 2, 0)
Y_train = dataset['Y_train'][0]
Y_test = dataset['Y_test'][0]

# adding the extra 'dimension' to X_train
X_train = np.insert(X_train, 0, 1, axis=0)
X_test = np.insert(X_test, 0, 1, axis=0)

class perceptron:
    def __init__(self, dimension, N=len(X_train[0])):
        """
        Perceptron
        --------
        Parameters
        ----
        - dimension: dimension of the feature vectors
        - N: number of training feature vectors
        """
        # initializing random weights, and all other attributes to zero
        self.w = np.random.rand(dimension,1)
        self.dw = np.zeros(self.w.shape)
        self.y_hat = np.zeros(N)
```

```
        self.v = np.zeros(N)
        self.e = np.zeros(N)
        self.delta = 0

    def forward(self, x):
        """
        Forward pass (calculates the output of the perceptron)

        Parameters
        ----
        - x: array of feature vectors
        """
        self.v = self.w.T @ x
        self.y_hat = 1/(1+np.exp(-self.v))

    def calculate_e(self, deltas, weights):
        """
        Calculate the error of the perceptron

        Parameters
        ----------
        - deltas: the deltas of the next layer
        - weights: the weights of the next layer (that are connected to the perceptron)
        """
        self.e = (deltas @ weights.T).T

    def calculate_delta(self):
        """
        Calculates the value for delta
        """
        f = 1/(1+np.exp(-self.v))
        df = f*(1-f)
        self.delta = np.multiply(self.e,df)

    def backward(self, y):
        """
        Backward pass (updates the weights of the perceptron)

        Parameters
        ----
        - y: array of outputs from the previous layer
        """
        self.dw = alpha*self.dw - (rho*y@self.delta.T) # (momentum term + cost function term)
        self.w += self.dw

class output_perceptron(perceptron):
    def __init__(self, label, dimension, y, N=len(X_train[0])):
        """
        Output perceptron (only difference from regular perceptron if calculating e and delta)

        Parameters
        ----
        - label: the label of the output perceptron
        - dimension: dimension of the feature vectors
        - y: training set
```

```
        - N: number of training feature vectors
        """
        super().__init__(dimension, N=len(X_train[0]))
        self.y = np.zeros(N)
        self.y[y == label] = 1

    def calculate_e(self):
        """
        Calculates the error
        """
        self.e = self.y_hat - self.y

    def calculate_delta(self):
        """
        Calculates the value for delta
        """
        f = 1/(1+np.exp(-self.v))
        df = f*(1-f)
        self.delta = self.e*df

class MLP:
    def __init__(self, shape, X_train, Y_train):
        """
        Multi-layer perceptron
        --------
        Parameters
        ----
        - shape: the shape of the hidden layers of the network, input layer is 2
    since the dataset is 2d and output layer is 3 nodes from the task description
        - X_train: training set
        - Y_train: training labels
        """

        # initializing attributes
        self.shape = shape
        self.X_train = X_train
        self.Y_train = Y_train
        self.layers = []
        layer = []

        # creating the layers
        # first hidden layer
        for i in range(shape[0]):
            layer.append(perceptron(3))
        self.layers.append(layer)

        # other hidden layers
        for i in range(1, len(shape)):
            layer = []
            for j in range(shape[i]):
                layer.append(perceptron(shape[i-1]+1))
            self.layers.append(layer)

        # output layer
        output_layer = [output_perceptron(1, shape[-1]+1, self.Y_train),
    output_perceptron(2, shape[-1]+1, self.Y_train), output_perceptron(3, shape
    [-1]+1, self.Y_train)]
        self.layers.append(output_layer)
```

```
145
146     def forward_propagation(self, X_train):
147         """
148         Propogates the network forwards (calculates the output of the network)
149
150         Parameters
151         ----------
152         - X_train: training set (input to the network)
153         """
154
155         # inpput the training set into the first layer
156         for neuron in self.layers[0]:
157             neuron.forward(X_train)
158
159         # propogate the output of the first layer to the next layer, and so on
160         for i in range(1, len(self.layers)):
161             input = []
162
163             for n in self.layers[i-1]: # for all neurons in the previous layer
164                 input.append(n.y_hat.tolist()[0]) # add the output to the input list
165             input = np.insert(input, 0, 1, axis=0) # add the bias term
166
167             for neuron in self.layers[i]: # propogate the input to the next layer
168                 neuron.forward(np.asarray(input))
169
170     def backward_propagation(self):
171         """
172         Propogates the network backwards (updates the weights of the network)
173         """
174
175         # calculate e and delta for the last layer
176         for neuron in self.layers[-1]:
177             neuron.calculate_e()
178             neuron.calculate_delta()
179
180         for i in reversed(range(0, len(self.layers)-1)): # iterate backwards through
     the layers
181             for j in range(len(self.layers[i])): # iterate through the neurons in
    the layer
182                 deltas = []
183                 weights = []
184                 for neuron in self.layers[i+1]: # for all neurons in the next layer
185
186                     deltas.append(neuron.delta) # add the delta to the list
187                     weights.append(neuron.w[j+1]) # add the weights to the list
188                 deltas, weights = np.matrix(np.asarray(deltas)).T, np.matrix(np.
    asarray(weights)).T # convert to matrices
189
190                 self.layers[i][j].calculate_e(deltas, weights) # calculate the new
    error for the neuron
191                 self.layers[i][j].calculate_delta() # calculate the new delta for
    the neuron
192
193         # update the weights for the first layer
194         for neuron in self.layers[0]:
195             neuron.backward(self.X_train)
196
197         # update the weights for the other layers
```

```
        for i in range(1, len(self.layers)):
            for j in range(len(self.layers[i])):
                y = []
                for neuron in self.layers[i-1]:
                    y.append(neuron.y_hat)
                y = np.asarray(y)
                y = np.insert(y, 0, 1, axis=0) # bias term
                self.layers[i][j].backward(np.matrix(y))

    def train(self, T, X_train, Y_train,X_test, Y_test, predict=False):
        """
        Training the network (updates the weights T times)

        Parameters
        ----
        - T: number of epochs
        - X_train: training set
        - Y_train: training labels
        - X_test: test set
        - Y_test: test labels
        - predict: (bool) if we want to get the accuracy for each epoch
        """

        # if we want to get the accuracy for each epoch
        if predict == True:
            predictions_train = np.zeros(T)
            predictions_test = np.zeros(T)
            for t in range(T):
                # Just for checking the progress
                if t % 500 == 0:
                    print(t)
                self.forward_propagation(self.X_train)
                self.backward_propagation()
                self.predict(X_train, Y_train)
                predictions_train[t] = self.accuracy
                self.predict(X_test, Y_test)
                predictions_test[t] = self.accuracy
            return predictions_train, predictions_test

        # if we only want the final accuracy
        else:
            for t in range(T):
                self.forward_propagation(self.X_train)
                self.backward_propagation()

    def predict(self, X_test, Y_test):
        """
        Predicting the labels of the test set

        Parameters
        ----
        - X_test: test set
        - Y_test: test labels
        """
        # calculate the output of the network
        self.forward_propagation(X_test)

        # taking the most probable class
```

```
        self.prediction = np.argmax([self.layers[-1][0].y_hat, self.layers[-1][1].y_hat, self.layers[-1][2].y_hat], axis=0)+1

        # calculating the accuracy
        self.accuracy = np.sum(self.prediction == Y_test)/len(Y_test)

    def decision_boundary(self, Y_test,resolution=100):
        """
        Plotting the decision boundary

        Parameters
        ----------
        - Y_test: The true labels of the test set
        - resolution: The resolution of the decision boundary
        """

        # creating the meshgrid
        x, y = np.linspace(-4.0, 4.2, num=resolution), np.linspace(-4.0, 4.2, num=resolution)
        XY = np.asarray(np.meshgrid(x,y)).reshape(2, -1)
        XY = np.insert(XY, 0, 1, axis=0)

        # calculating the output of the network
        self.forward_propagation(XY)

        # predicting the labels
        predictions = np.argmax([self.layers[-1][0].y_hat, self.layers[-1][1].y_hat, self.layers[-1][2].y_hat], axis=0)+1

        # plotting the decision boundary
        plt.pcolormesh(x, y, predictions.reshape(resolution, resolution), cmap = 'Set1', alpha=0.6, vmin=0, vmax=12)
        plt.scatter(X_test[1, Y_test == 1], X_test[2, Y_test == 1], marker = '+', color='red', label='Class 1')
        plt.scatter(X_test[1, Y_test == 2], X_test[2, Y_test == 2], marker = '+', color='blue', label='Class 2')
        plt.scatter(X_test[1, Y_test == 3], X_test[2, Y_test == 3], marker = '+', color='green', label='Class 3')
        plt.xlabel("$x_1$")
        plt.ylabel("$x_2$")

    def plot_confusion_matrix(self, Y_test, title):
        """
        Plotting the confusion matrix for the prediction

        Parameters
        ----------
        - Y_test: The true labels of the test set
        - title: The title of the plot
        """

        confusion_matrix = metrics.confusion_matrix(Y_test, self.prediction[0])
        confusion_matrix_display = metrics.ConfusionMatrixDisplay(confusion_matrix)
        confusion_matrix_display.plot(cmap="Reds")
        plt.title("Confusion matrix for " + title)
        confusion_matrix_display.ax_.set_xticks([])
        confusion_matrix_display.ax_.set_yticks([])
        plt.show()
```

```
307
308 if __name__ == "__main__":
309     # defining the parameters
310     rho = 0.008
311     alpha = 0.65
312     shape = [10, 8, 6]
313     T = 2500
314
315     # creating and training the network
316     nn = MLP(shape, X_train, Y_train)
317     accuracies_train, accuracies_test = nn.train(T, X_train, Y_train, X_test, Y_test
        , predict=True)
318
319     # plotting the decision boundary
320     print("Test set: ", np.round(accuracies_test[-1], 3), ", Training set: ", np.
        round(accuracies_train[-1], 3))
321     plt.title(chr(961) + "=" + str(rho) + " " + chr(945) + "=" + str(alpha))
322     nn.decision_boundary(Y_test, resolution = 200)
323     plt.show()
324
325     # plotting the accuracy as a function of T
326     plt.plot(np.arange(T), accuracies_train, label="Training set", color = 'blue')
327     plt.plot(np.arange(T), accuracies_test, label="Test set", color = 'red')
328     plt.xscale("log")
329     plt.title("Accuracy as a function of T")
330     plt.xlabel("T")
331     plt.ylabel("Accuracy")
332     plt.legend()
333     plt.show()
334
335     # plotting the confusion matrix
336     nn.plot_confusion_matrix(Y_test, " Test set")
```