O4: MAL End Project

Determine shapes

Mikkel Bleeg Carstensen	au557112
Vinh Trung Thai	au554243
Maja L S Andersen	au485255

ITMAL Group 07

12-02-2019

Authors

	Trung	**Mikkel**	**Maja**
Preprossing			**x**
ML-Algorithm			**x**
ML pipeline		**x**	
Performance metrics			**x**
Optimization and improvements	**x**		
Under- og overfitting		**x**	
Discussion	**x**		

Introduction

Machine learning is a hot topic at the moment, and developers and data scientist are experimenting with interesting ways to make use of it. Whether it is for commercial use, industrial use or just for fun, it is quite hard not to encounter it. The earliest encounter of machine learning may be Bayes thereom which describes the probability of an event. Through statistical analyses and experiments, we are able to predict an outcome of an event. Broadly it could be said there are three use types in machine learning. Those would be recognition, fx image, voice, etc., prediction, fx finance, social media platforms and surveillance, fx spam and malware, fraud, etc.
For this project, our learning and understanding of machine learning will be applied to image recognition, which will be further explained in the upcoming sections.

Problem description

This project is about detecting and classifying four different shapes. We want our algorithm to detect differences in the shapes and be able to show the user what kind of shape it's detecting. The four shapes that are analyzed are a circle, a square, a triangle and a star.

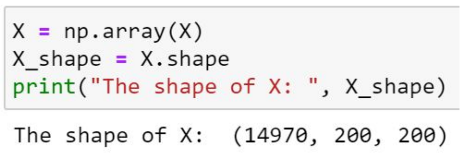
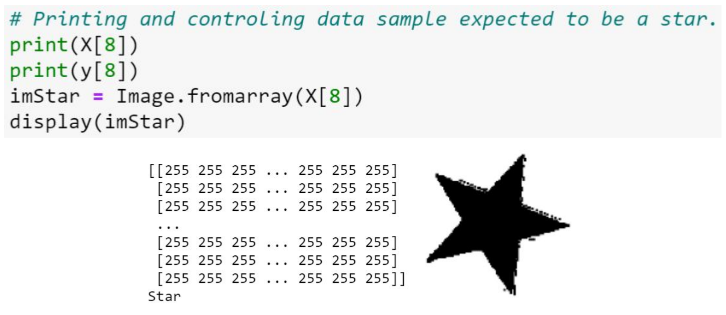
The algorithm should be able to consistently detect which shape is which, even though they are a bit crooked or if there is noise in the background.

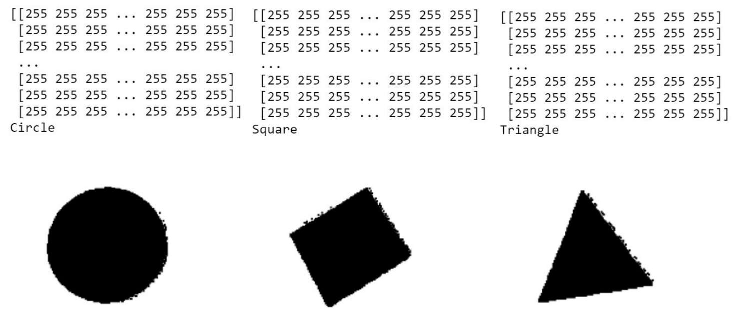
The dataset

The dataset consists of black and white images picturing different shapes. The images are extracted and preprocessed from a video where the shapes have been moved and rotated into different positions. In the preprocessing process the pictures have been reshaped, filtered and transformed from RGB to grayscale. The dataset consisted of four folders with approximately 3720 .pgn pictures of the shapes.

The dataset was found on Kaggle at the link[https://www.kaggle.com/smeschke/four-shapes]

Features	200x200 pixels equal to 40.000 features.

Samples	14.970 sample pictures of the four shapes divided into approximately 4.740 sample pictures pr. Shape.
	Each image is of the size pixel size 200x200 and in grayscale, as seen on the image 01.
	
	`X = np.array(X)`
	`X_shape = X.shape`
	`print("The shape of X: ", X_shape)`
	`The shape of X:  (14970, 200, 200)`
	Image 01: shape of data array
	This means that each pixel contains only one value between 0 and 255 instead of an array with tree RGB values, as seen on image 02.
	
	`# Printing and controling data sample expected to be a star.`
	`print(X[8])`
	`print(y[8])`
	`imStar = Image.fromarray(X[8])`
	`display(imStar)`
	`[[255 255 255 ... 255 255 255]`
	` [255 255 255 ... 255 255 255]`
	` [255 255 255 ... 255 255 255]`
	` ...`
	` [255 255 255 ... 255 255 255]`
	` [255 255 255 ... 255 255 255]`
	` [255 255 255 ... 255 255 255]]`
	`Star`
	Image 02: print of content in data array, label and star shape.
	When we examine the other images in the same way we see that they have similar structure as on image04:



```
[[255 255 255 ... 255 255 255]   [[255 255 255 ... 255 255 255]   [[255 255 255 ... 255 255 255]
 [255 255 255 ... 255 255 255]    [255 255 255 ... 255 255 255]    [255 255 255 ... 255 255 255]
 [255 255 255 ... 255 255 255]    [255 255 255 ... 255 255 255]    [255 255 255 ... 255 255 255]
 ...                              ...                              ...
 [255 255 255 ... 255 255 255]    [255 255 255 ... 255 255 255]    [255 255 255 ... 255 255 255]
 [255 255 255 ... 255 255 255]    [255 255 255 ... 255 255 255]    [255 255 255 ... 255 255 255]
 [255 255 255 ... 255 255 255]]   [255 255 255 ... 255 255 255]]   [255 255 255 ... 255 255 255]]
Circle                           Square                           Triangle
```

Image 03: print of the rest of the shapes.

Target values	The Labels : - Cirkel - Square - Triangle - Star
Errors/uncertainty	We don't expect any errors or uncertainties to emerge because of faults in the dataset, as the images are made from resized pictures.

Preprocessing

For SourceCode look to the document Preprossing_CNN_Metrics

The data preprocessing have been through two faces, the preprocessing done by the dataset provider and the preprocessing done by us. As there have been done a preprocessing prior to our work with the dataset we will also discuss this since it's interesting in a ML perspective.

Preprocessing by the dataprovider

The dataprovider states in the description of the dataset that they have been using OpenCV colorspaces in python to process the raw data from the video. Colorspaces is a way of representing the different RGB colorspaces in a picture.[1] And since the shapes in the video are bright green we assume that the data providers must have taken the G-channel and filtered with a high threshold value before the images have been reshaped.

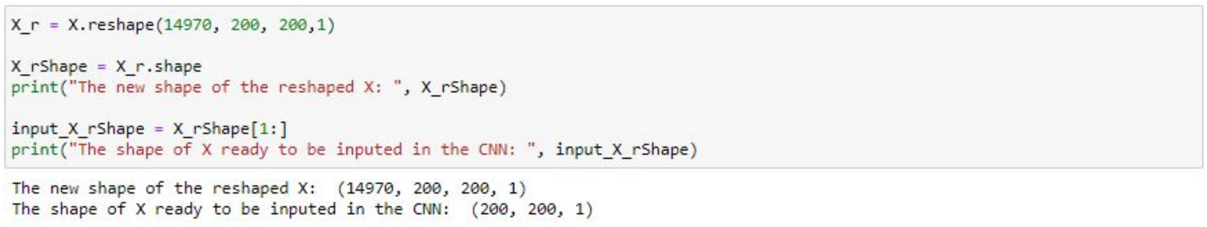
Preprocessing by us

The preprocessing that we have done have been to label the data, to further reshape the data and to one-hot-encode it.

Since our data came in folders with the pictures we needed to load all pictures from the four folders into one array X and create a array y with corresponding labels. The function we created for this can be seen in the source code under ./ProjectFunctions/LoadShapes.py.

[1] https://www.geeksforgeeks.org/color-spaces-in-opencv-python/

We knew that we wanted to use CNN and neural networks to train on our data. For this reason we reshaped the data and gave it another dimension to let the CNN know that we're working with grayscale data, as seen on image 5.

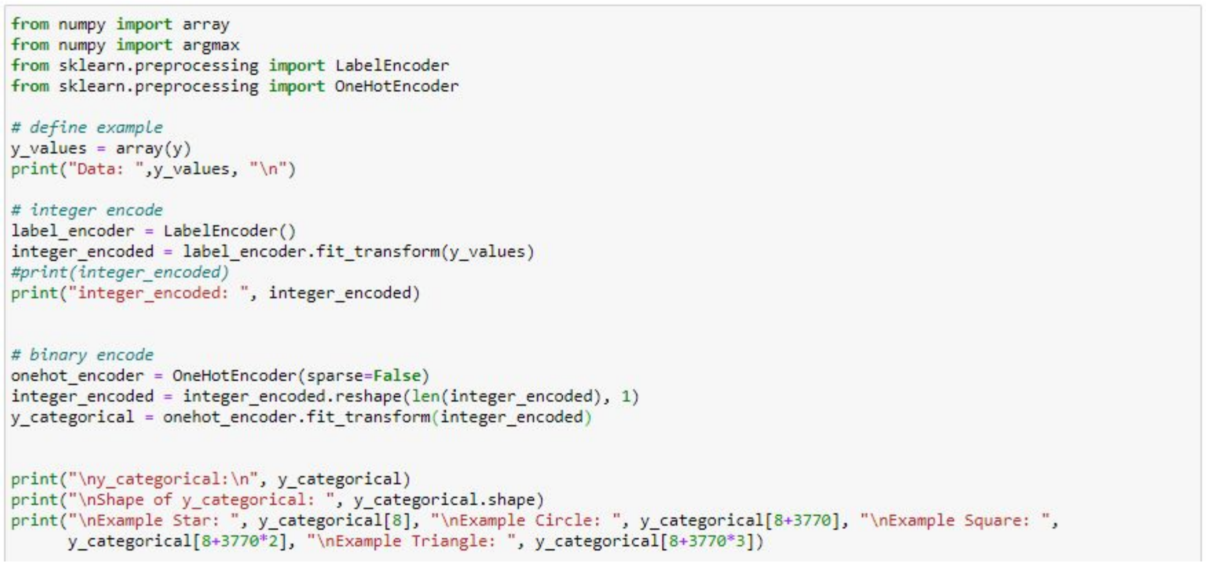


```
X_r = X.reshape(14970, 200, 200,1)

X_rShape = X_r.shape
print("The new shape of the reshaped X: ", X_rShape)

input_X_rShape = X_rShape[1:]
print("The shape of X ready to be inputed in the CNN: ", input_X_rShape)
```

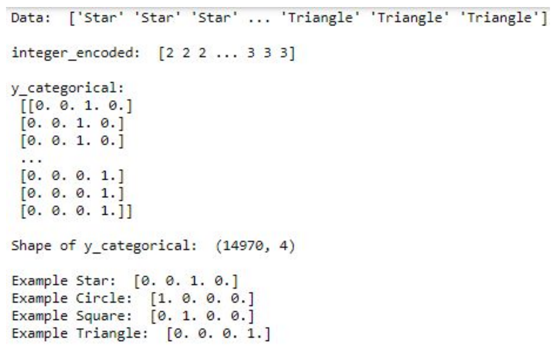
```
The new shape of the reshaped X:  (14970, 200, 200, 1)
The shape of X ready to be inputed in the CNN:  (200, 200, 1)
```

Image 05: reshaping of the data array.

Furthermore we use one-hot encoding on the categorical data, which our expected outcome. One-hot-encoding is a ML method that is used to transform categorical data into numerical and binary data. This is because most ML algorithms works best with numerical input and output data such as integers or a 1d array with numerical data[2]. This encoding is seen on image 06.



```
from numpy import array
from numpy import argmax
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder

# define example
y_values = array(y)
print("Data: ",y_values, "\n")

# integer encode
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(y_values)
#print(integer_encoded)
print("integer_encoded: ", integer_encoded)


# binary encode
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
y_categorical = onehot_encoder.fit_transform(integer_encoded)


print("\ny_categorical:\n", y_categorical)
print("\nShape of y_categorical: ", y_categorical.shape)
print("\nExample Star: ", y_categorical[8], "\nExample Circle: ", y_categorical[8+3770], "\nExample Square: ",
      y_categorical[8+3770*2], "\nExample Triangle: ", y_categorical[8+3770*3])
```

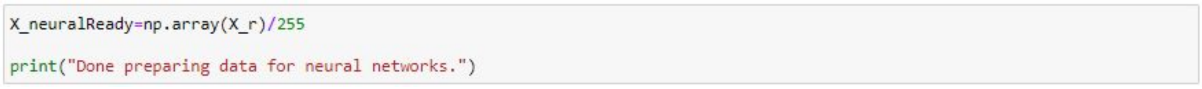


```
Data:  ['Star' 'Star' 'Star' ... 'Triangle' 'Triangle' 'Triangle']

integer_encoded:  [2 2 2 ... 3 3 3]

y_categorical:
 [[0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 ...
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]

Shape of y_categorical:  (14970, 4)

Example Star:  [0. 0. 1. 0.]
Example Circle:  [1. 0. 0. 0.]
Example Square:  [0. 1. 0. 0.]
Example Triangle:  [0. 0. 0. 1.]
```

Image 06: one-hot-encoding.

Afterwards we changed the pixel values from 0-255 to 0-1. This is because the neurons works within the range from 0 to 1 in the step functions that is in the activation function. For our data to be processed correctly in the neural network we need to feed the network with data in the same range. On image 07 below this processing step is shown.

[2] https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/



```
X_neuralReady=np.array(X_r)/255

print("Done preparing data for neural networks.")
```

Image 07: preparing the data for CNN.

ML-Algorithm

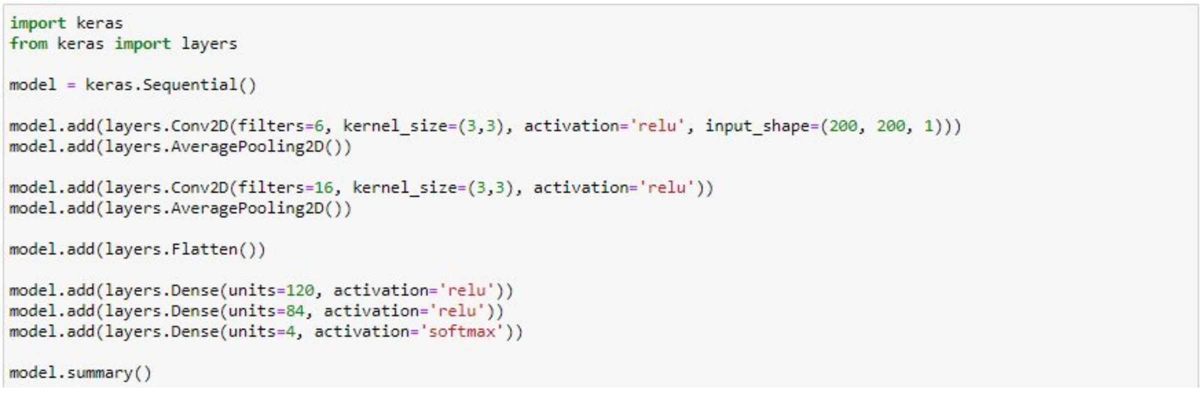
For SourceCode look to the document Preprossing_CNN_Metrics
We have chosen to use CNN (Convolutional Neural Network) to classify our data. We chose CNN because of the following criteria; because CNN its a strong and often used tool for classifying image data.

Our use of CNN is supervised learning[3] because we feed our algorithm with data consisting of images with labels describing the desired outcome of our image classification which is either a Circle, Triangle, Star or Square.

One of the challenges with using CNN is that it takes a lot of computing power and takes a lot of time to train on datasets. This we also saw while training the algorithm on our data we found that our CNN took approximately 20 minutes to train on our train dataset as seen on image 10. Which is a fairly long time. This could have been optimized by running it on a GPU server since GPU's have more dedicated computing power and higher bandwidth. Another way to optimize could be to do a grid search for best parameters.

Building the model

We have built our CNN sequential, which means that that we can build it step by step and add one layer at the time, as shown on image 08.



```
import keras
from keras import layers

model = keras.Sequential()

model.add(layers.Conv2D(filters=6, kernel_size=(3,3), activation='relu', input_shape=(200, 200, 1)))
model.add(layers.AveragePooling2D())

model.add(layers.Conv2D(filters=16, kernel_size=(3,3), activation='relu'))
model.add(layers.AveragePooling2D())

model.add(layers.Flatten())

model.add(layers.Dense(units=120, activation='relu'))
model.add(layers.Dense(units=84, activation='relu'))
model.add(layers.Dense(units=4, activation='softmax'))

model.summary()
```

Image 08: building the CNN sequentially.

Our Neural Network consists of 7 layers. The first four input layers are two convolutional layers (layer 1 and 3) and two pooling layers (layer 2 and 4). We use the convolutional layer *Conv2D* because we are working with two dimensional data in our pictures. This we also tell our first layer by with the *input_shape=(200,200,1)* since our picture is 200 times 200 pixels and are in grayscale which we teels the algorithm with the *1*. Then we set *kernel_size=(3,3)*

[3] Page 8 Chapter 1 Hands-On Machine learning by Aurélien Géron

which means that the filter matrix of our convolution is 3x3. Furthermore we have the size of output filters to 6 and 16 describing how many filters we have in depth.

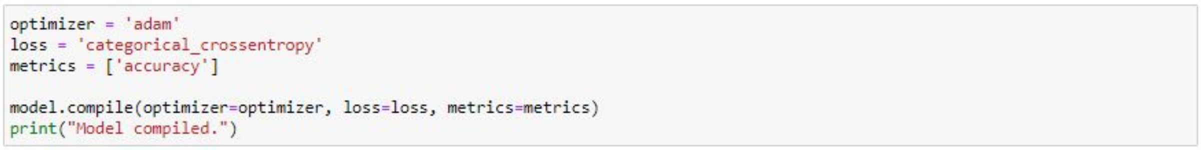
Our activation function is relu (Rectified Linear Unit)[4] and we use it because it is one of the most used activation functions in CNN and therefore is a good algorithm to start with[5]. The reason for this is because the ReLU function is quite simple $y = max(0, x)$ and therefore won't use too much computing power, which is good since CNN already is computation heavy.

In the middle we have a flattened layer between the input and output layers that connects the input and output layers.

Thereafter we tree Dense[6] output layers that connects the input neurons to output neurons, we define the number of output neurons as units and decreases the output from 120 to 4 so that we ends up with the classification of the four categories. In the last layer we uses softmax that takes the input and transforms it to a probability distribution[7], and this way the categorisation is determined on the highest probability.

Training the model

When compiling and training the model we uses the optimizer *adam*, the loss function *categorical_crossentropy*, and the metrics we validate our function in is accuracy, as seen on image 09.



```
optimizer = 'adam'
loss = 'categorical_crossentropy'
metrics = ['accuracy']

model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
print("Model compiled.")
```

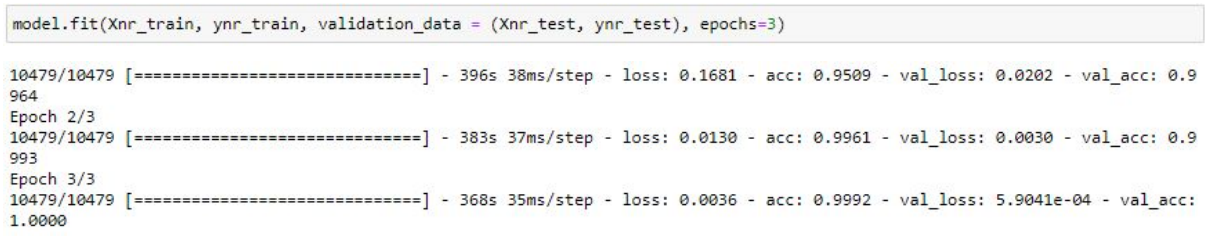
Image 09: compiling the model.

Adam is an adaptive optimizer that performs local optimization based on the history of iteration[8] that way it adjusts the learning rate while training. This means that adam is a faster and less heavy optimizer than other optimizers such as SGD.

Categorical_crossentropy is the loss function that we're using because it's often used with one-hot-encoded categorical data[9]. The *Categorical_crossentropy* is also called softmax loss because it is a combination of a softmax activation and cross loss entropy this works good together with our softmax output layer and with CNN in general.
Categorical_crossentropy returns the probability for what category the input data is in. The lower the loss the better the model is performing.

[4] https://keras.io/activations/
[5] https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7 and https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6
[6] https://stackoverflow.com/questions/56005323/what-is-dense-layer
[7] https://en.wikipedia.org/wiki/Softmax_function
[8] https://arxiv.org/abs/1705.08292 and exercise to lecture 06 keras_mlp_moon
[9] https://gombru.github.io/2018/05/23/cross_entropy_loss/

We use the accuracy as the metrics for which we're measuring the performance of our model, and the result of the training is seen on image 10.



```
model.fit(Xnr_train, ynr_train, validation_data = (Xnr_test, ynr_test), epochs=3)
```

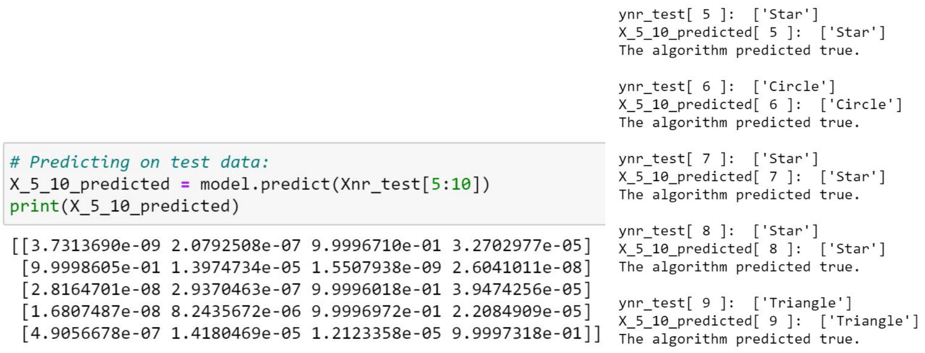
```
10479/10479 [==============================] - 396s 38ms/step - loss: 0.1681 - acc: 0.9509 - val_loss: 0.0202 - val_acc: 0.9
964
Epoch 2/3
10479/10479 [==============================] - 383s 37ms/step - loss: 0.0130 - acc: 0.9961 - val_loss: 0.0030 - val_acc: 0.9
993
Epoch 3/3
10479/10479 [==============================] - 368s 35ms/step - loss: 0.0036 - acc: 0.9992 - val_loss: 5.9041e-04 - val_acc:
1.0000
```

Image 10: fitting the model to our data, with results from the different epochs.

If we take a look at the results above on image 10 we see a very low validation loss close to 0 and a high accuracy on 1 (100%) in our third iteration(epoch) of training the algorithm. This what we hoped to achieve but also raises the question whether we have overfitted on the data. We will discuss overfitting in a later section.

Testing on the test set

If we look at some random data in the test set we can try and see if our algorithm predicts the true outcome.



```
# Predicting on test data:
X_5_10_predicted = model.predict(Xnr_test[5:10])
print(X_5_10_predicted)
```

```
[[3.7313690e-09 2.0792508e-07 9.9996710e-01 3.2702977e-05]
 [9.9998605e-01 1.3974734e-05 1.5507938e-09 2.6041011e-08]
 [2.8164701e-08 2.9370463e-07 9.9996018e-01 3.9474256e-05]
 [1.6807487e-08 8.2435672e-06 9.9996972e-01 2.2084909e-05]
 [4.9056678e-07 1.4180469e-05 1.2123358e-05 9.9997318e-01]]
```

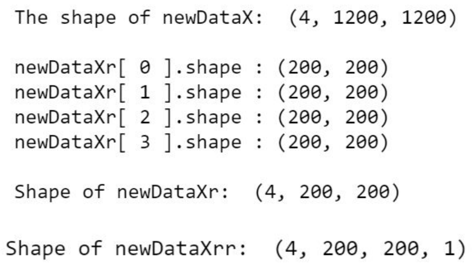
```
ynr_test[ 5 ]:  ['Star']
X_5_10_predicted[ 5 ]:  ['Star']
The algorithm predicted true.

ynr_test[ 6 ]:  ['Circle']
X_5_10_predicted[ 6 ]:  ['Circle']
The algorithm predicted true.

ynr_test[ 7 ]:  ['Star']
X_5_10_predicted[ 7 ]:  ['Star']
The algorithm predicted true.

ynr_test[ 8 ]:  ['Star']
X_5_10_predicted[ 8 ]:  ['Star']
The algorithm predicted true.

ynr_test[ 9 ]:  ['Triangle']
X_5_10_predicted[ 9 ]:  ['Triangle']
The algorithm predicted true.
```

Image 11: Predicting the data from the test set.

We can see that for the five samples that we have chosen that the function predicts true for all samples.

Testing on new data

We created new data to see if the algorithm was able to determine new data not from the original dataset. We created this pictures in *paint.net* with their generic shapes, loaded and reshaped the data, as seen on image 12.



```
The shape of newDataX:  (4, 1200, 1200)

newDataXr[ 0 ].shape : (200, 200)
newDataXr[ 1 ].shape : (200, 200)
newDataXr[ 2 ].shape : (200, 200)
newDataXr[ 3 ].shape : (200, 200)

Shape of newDataXr:  (4, 200, 200)

Shape of newDataXrr:  (4, 200, 200, 1)
```

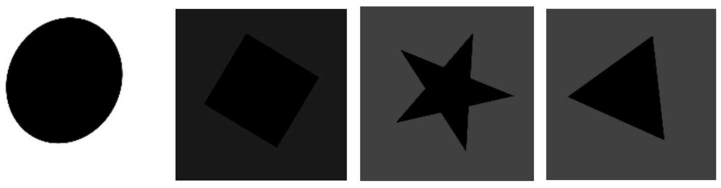


Image 12: hape and images of the new created shapes.

The result of the data is as seen in the text string below, where the algorithm detects all the shapes correctly.

```
[array(['Circle'], dtype='<U8'), array(['Square'], dtype='<U8'), array(['Star'], dtype='<U8'), array(['Triangle'], dtype='<U
8')]
```

For future work and to challenge the algorithm we could have hand drawn some pictures and scanned them to see the result.

ML pipeline

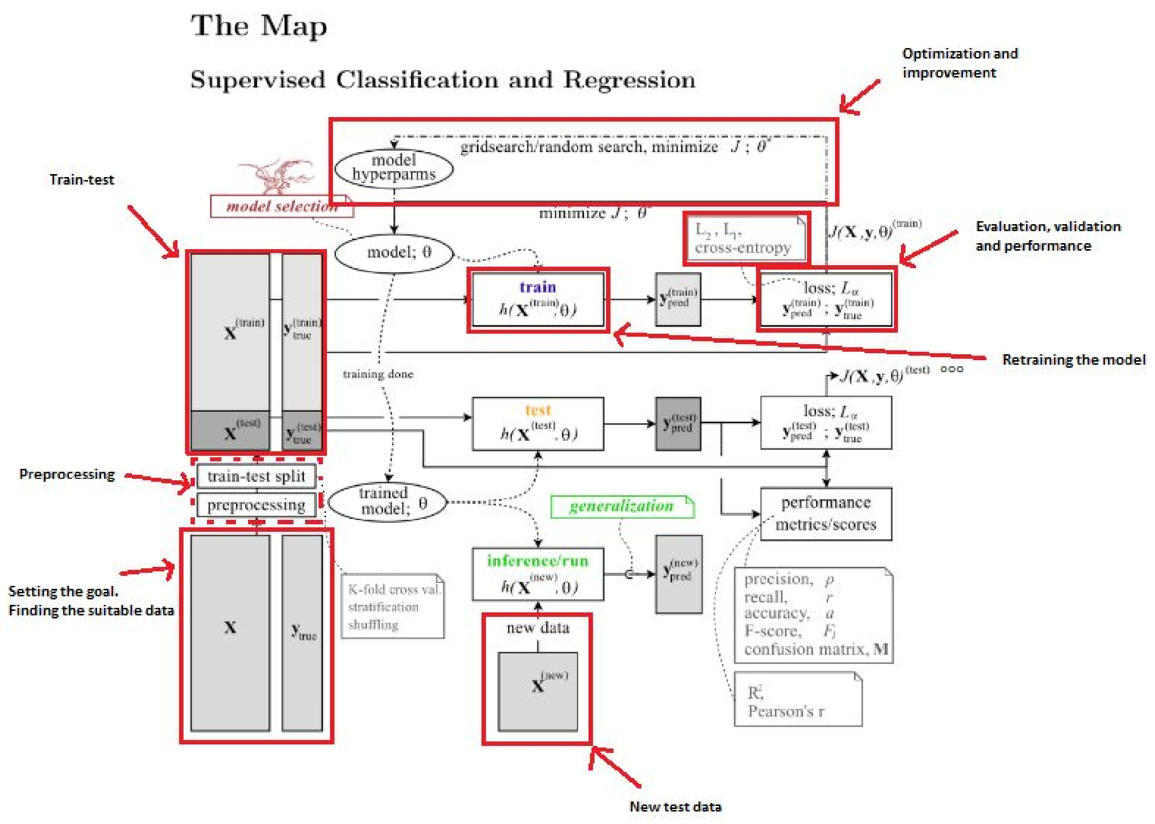


Image 13: Our pipeline through supervised classification.

Preprocessing

For preprocessing of the data, we reshape the data to make it fit the CNN model. Afterwards we use one-hot encoding, which is a method used to transform categorical data into numerical and binary data.
The data is then prepared for neural networks. This means that the pixel data is changed from 0 - 255 to a value between 0 and 1. This is done by dividing the X_r array with 255.

The reason for this, is because the neurons works with step functions in the activation function step. For the data to be processed correctly in a neural network, it’s needed to be prepared with data within the same range.

Train-test

For this project we use the Sklearn train_test_split library at first, to split the dataset into random train and test subsets called X and y. The test_size decides how large of a percentage of the dataset is worked on.

The data is also split up in train and test set, which then will work with a neural network, theses are called Xnr and ynr.

Performance metrics

For SourceCode look to the document Preprossing_CNN_Metrics
When validating how good the algorithm performed we have looked at two different performance metrics classification accuracy and confusion matrix.

Classification Accuracy

As shown in an earlier section we an accuracy on 1 or 100% which was the desired outcome. In our case with almost equal amount of samples to every category the accuracy is a reliable metrics to validate the algorithm with. In some cases accuracy can give a false measure of performance, for example in datasets with big difference in the amount of samples to different categories. Here one could have a high accuracy score when detecting a category with many samples but in not being able to detect categorize the categories belonging to the smaller categories.

Confusion Matrix

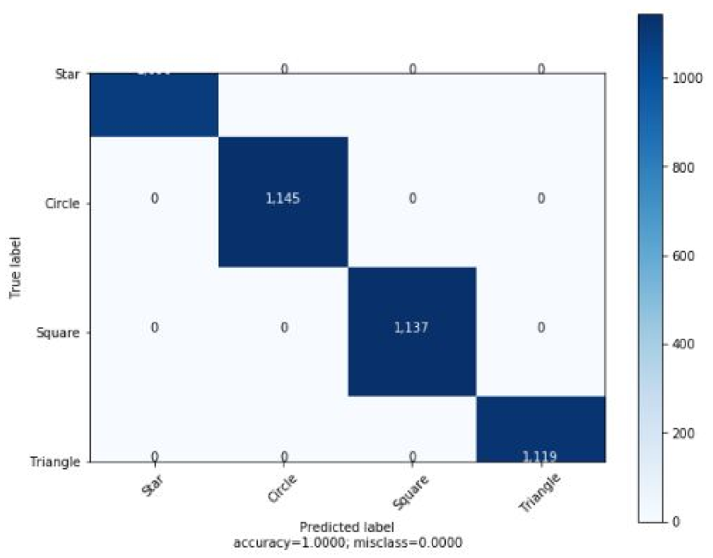
Another way we looked at the performance of our algorithm is to plot a confusion matrix this also showed that we have 100% precise algorithm. This we can calculate by the formula[10]:

$$precision = \frac{TP}{TP+FP} = \frac{4491}{4491+0} = 1$$

The confusion matrix (seen on image 14 below) shows the true positive(TP) and enables one to calculate the true negative(TN), false positive(FP) and true negative(FN)[11].

[10] https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62
[11] https://youtu.be/FAr2GmWNbT0?t=185

```
Normalized Confusion matrix with heat plot:
```



```
Raw Confusion Matrix:

 [[1090    0    0    0]
 [   0 1145    0    0]
 [   0    0 1137    0]
 [   0    0    0 1119]]
```

Image 14: confusion matrix and normalized confusion matrix shown as heat plot.

From the dark blue diagonal one can read the true positive of the different categories. The True negative can be calculated from the sum of all columns and rows except those for which category one wishes to calculate TN for. False positives are calculated from the column of the category minus the TP. False negatives are calculated from the rows of the category minus the TP. For our project this means that FP and FN are false for both 0.

From looking at the accuracy and confusion matrix we can conclude that the algorithm has a high accuracy and precision.

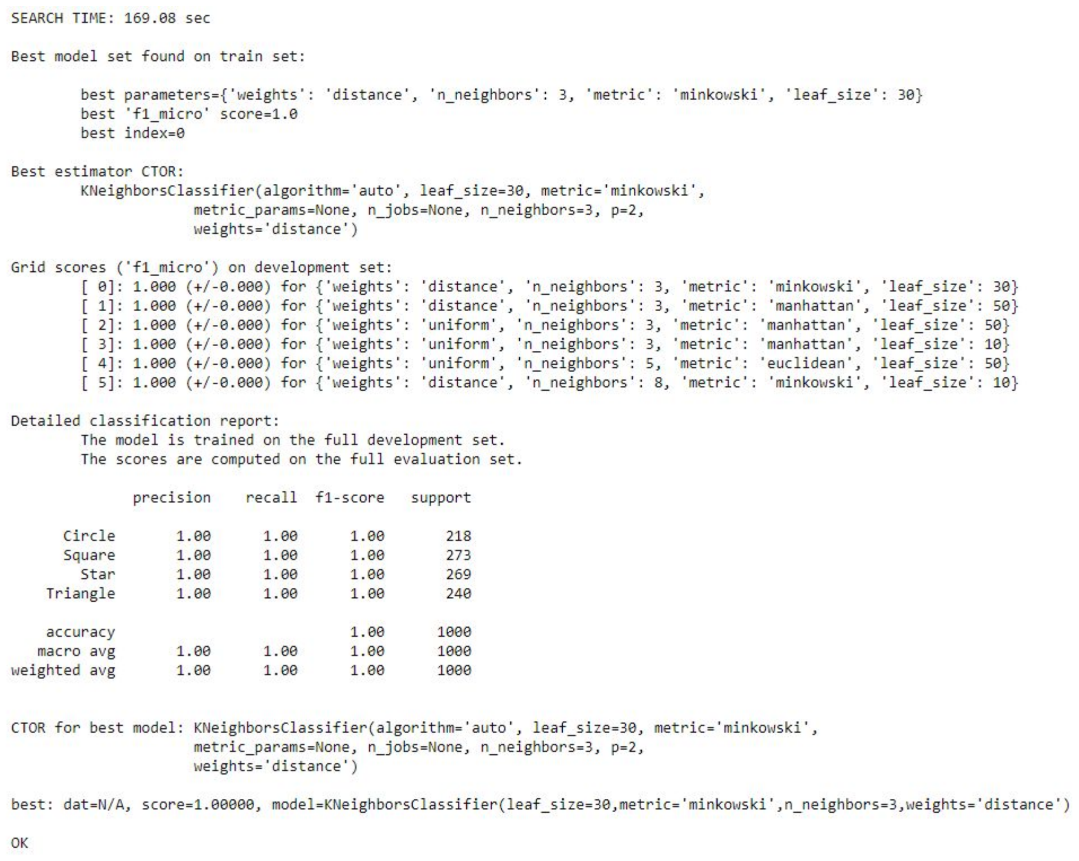
Optimization and improvements:

If a model doesn't fulfill the expectations as desired, an attempt in optimizing and improvements can be made. These optimizations can be made with optimization parameters or search for hyperparameters.

For optimizing of chosen dataset, the hyperparameter searching method was chosen. When using hyperparameter searching, different approaches can be taken. For our system the two approaches chosen were GridSearchCV and RandomSearchCV for given models. Both optimization method originates from scikit-learn. To find the best optimization, the search methods were used with different combinations of tuning parameters, implemented in a parameter_grid. The search methods will try to optimize the models using the parameters given, which then results in scores to given parameters. The scores indicates which tuning parameters are best for optimizing. The best scoring parameters is known as the best parameters for the given dataset.

Various models was used for optimizing. Sadly, it was not possible to optimize the self made model, as it could not optimize multi-labelled targets. Despite attempts on trying to optimize the model, unsuccessfully, model still managed to get a score of 0.99 . One would therefore argue whether optimization is needed for this model, as the score is 0.99 and can't reach above 1. For this reason, no further attempts were made to optimize the system. Instead, the model was compared to other models after optimization which was able to handle multi-labelled or multi-class targets.

When comparing the different models with both GridSearch and RandomSearch, the resulting output gave various different scores. As seen in the results above, the search found the best estimator for optimizing by trying combinations with the given tuning parameters for the given model. The various combinations saw scores of 0.997 and 0.998 for the SVM model. With the best estimator for optimization a score of 1, which is a perfect score, was achieved. The tuning parameter for the best estimator is found to be: Kernel=rbf and C-gamma=1 for the SVM model. The SGDClassifier gave various results with for optimization. As seen in the results, the score could swing 0.294 up to 0.999. If given more iterations, it could be assumed the hyperparameter tuning would find the best estimator for the given dataset. As for KNeighbor model, each parameter gave a score of 1, which can be very questionable, whether it is a valid tuning model for these types of data, see image15.



```
SEARCH TIME: 169.08 sec

Best model set found on train set:

        best parameters={'weights': 'distance', 'n_neighbors': 3, 'metric': 'minkowski', 'leaf_size': 30}
        best 'f1_micro' score=1.0
        best index=0

Best estimator CTOR:
        KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                     weights='distance')

Grid scores ('f1_micro') on development set:
        [ 0]: 1.000 (+/-0.000) for {'weights': 'distance', 'n_neighbors': 3, 'metric': 'minkowski', 'leaf_size': 30}
        [ 1]: 1.000 (+/-0.000) for {'weights': 'distance', 'n_neighbors': 3, 'metric': 'manhattan', 'leaf_size': 50}
        [ 2]: 1.000 (+/-0.000) for {'weights': 'uniform', 'n_neighbors': 3, 'metric': 'manhattan', 'leaf_size': 50}
        [ 3]: 1.000 (+/-0.000) for {'weights': 'uniform', 'n_neighbors': 3, 'metric': 'manhattan', 'leaf_size': 10}
        [ 4]: 1.000 (+/-0.000) for {'weights': 'uniform', 'n_neighbors': 5, 'metric': 'euclidean', 'leaf_size': 50}
        [ 5]: 1.000 (+/-0.000) for {'weights': 'distance', 'n_neighbors': 8, 'metric': 'minkowski', 'leaf_size': 10}

Detailed classification report:
        The model is trained on the full development set.
        The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

      Circle       1.00      1.00      1.00       218
      Square       1.00      1.00      1.00       273
        Star       1.00      1.00      1.00       269
    Triangle       1.00      1.00      1.00       240

    accuracy                           1.00      1000
   macro avg       1.00      1.00      1.00      1000
weighted avg       1.00      1.00      1.00      1000


CTOR for best model: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                     weights='distance')

best: dat=N/A, score=1.00000, model=KNeighborsClassifier(leaf_size=30,metric='minkowski',n_neighbors=3,weights='distance')

OK
```
[12]

Image 15: report from grid search.

No further optimization method was tested as the grid and random search, with KNN, achieved a perfect score. Other methods could be used to speed up the optimization, such as different types of regression, but may cost effectiveness on the score.

But with a score of 1, it could be questionable whether the system is overfitted.

[12] Source code for optimization models comparison: Gridsearch_onEnd.ipynb

Under- og overfitting:

Under- and overfitting, is when the algorithm is given either too little or too much data to train with. If our algorithm is underfitted, it will not be able to recognize the shapes and categorize them accordingly.

Usually when an algorithm is overfitted, it would know the data too well, it would usually deal with all the data points, instead of making a prediction.

With our dataset, it's hard to see if there is overfitting, since the algorithm is supposed to recognize different shapes, so there is no clear indication of this. Since it would still be categorizing the shapes correctly,

To avoid underfitting, we can keep testing the algorithm, to see if it has a stable, acceptable score. If the score is too low, the different variables can be tweaked. There can be added more or fewer layers or it can be optimized with search, which is described in the previous section.

For avoiding overfitting we use cross-validation, in some of the tests in the grid-search. When the data is split into multiple smaller train, test splits.
During the tests, there is also used the sklearn train_test_split function, which automatically makes cross-validation on the test data.

Discussion

Optimization and improvement

Due to time schedule, the group was not able to find an optimal way to optimize the model. When the self made model was compared to other models, it was not able to reach a perfect score of 1. One could imagine a way to optimize the self made model, but not without changing the data modelling.
If an attempt to optimize the system was desired, the model has to be changed. The layers must be able to change parameters as well as either handle multi-label or multi-class variable targets or modulate the targets to single label or class.

Conclusion

For this project, we were able to achieve an accuracy and precision score of 1 or in some cases 0.99, which is close to one. This could be the result of the algorithm being overfit with training data, but we've tried to take our measurements by cross-validating the data.
The CNN algorithm was used for the dataset, to achieve the best possible performance. The algorithm was chosen early in the process. The CNN algorithm is used to take an input image and assign importance to various aspects of the image, which makes it able to differentiate one picture from another.
Since the goal for the project was to differentiate between four shapes, the algorithm was the best choice.
The results for our model has been compared to other models which saw similar score results. It could therefore be assumed we have found a fitting model. Through various test with self made images, the system was able to classify and categorize the new test data, which was shown in the Testing on new data section.
The overall result for the end project satisfied our expectations as we were able to classify the shapes to the split test data as well as the new self made data.