SWMAL gruppe 21- O[3]

Navn	StudieNummer	AUID
Mohamed Hashem Abodu	2019100895	Au656408

26/09/2023

L07/ cnn.ipynb:

In this exercise I will dive into convolutional neural networks and build a CNN via keras API, while trying to achieve the highest accuracy score possible on the MNIST dataset.

Intro

CNNs aka Convolutional Neural Networks, emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980. [HOML].
CNNs are basically specialised deep networks designed for processing grid data, fx images.

The most important building block of a CNN is the convolutional layer, where neurons in the first layer are not connected to every single pixel in the input, but only to pixels in their receptive fields. This architecture allows the network to concentrate on small low-level features in the first layer, assemble them into larger higher-level features in the next hidden layer and so on. [HOML]. These layers are followed by pooling layers aka downsample layers that are responsible for reducing the spatial dimensions of the input data, in terms of width and height, and increasing computational efficiency.

Building the CNN

Let's start with the import statements:

```
import pandas as pd
import tensorflow as tf
from sklearn.datasets import fetch_openml
import keras
from keras import layers
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
import matplotlib.pyplot as plt
```

These libraries above are our way to implement our CNN, plus Matplotlib for graph plotting. Next I copied MNIST_getdataset from O[1] to load the MNIST dataset to train our model on.

```
 ##loading the mnist dataset from O[1]
def MNIST_GetDataSet():
  mnist = fetch_openml('mnist_784',version=1,
                       return_X_y=True, as_frame=False)
  return mnist
X,y = MNIST_GetDataSet()
```

Then i realised that the input images(X) must have a 4D shape to be suitable for the convolutional layers we are going to use, so i had to reshape and then normalise the pixel values to be in the range [0,1] [TensorFlow] :

```
 X = X.reshape((-1, 28, 28, 1)).astype('float32') / 255
```

Then using sklearn.model_selection, i split the dataset into training and testing sets :

```
X_train, X_test, y_train, y_test = train_test_split(X,y,
test_size=0.2,random_state = 42 )
```

Here comes the fun part, I am going to build this CNN via keras sequential API, as described in the book [HOML] , the CNN is built with convolutional layers, max-pooling layers, a flattening layer, and dense layers. Then compiling it with the necessary metrics.

I have tried to implement three different sequential models with different layers structures to observe the results of these implementations. The first model has one Conv2D layer, one MaxPooling2D layer and one flatten layer. The second model has double as many layers and a Dropout layer to help with the regularisation and reduce overfitting. The third model has the same structure as model 2 but with different numbers of filters and dropout rate. The table below is the accuracy score for the mentioned models:

Model 1	Model 2	Model 3
0.9791	0.9903	0.9905

In the following figure we can see the accuracy score for both validation and test subsets over epochs.

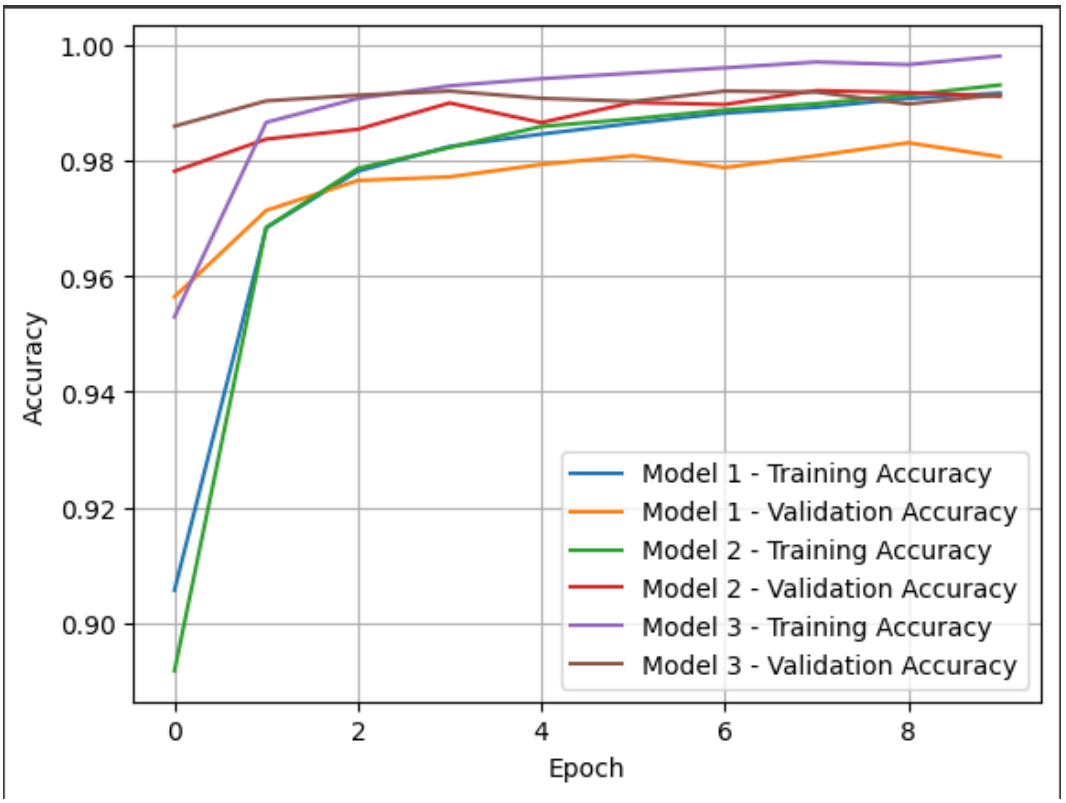

Figure 1. Accuracy scores for the models in test

We can see in the figure above that all the models are performing relatively similar, but also noticed that the model with most layers and filters does not perform best, which also means a possibility to overfitting, which we are trying to avoid, that's why i chose model 2:

```
 #Build the CNN
model = keras.Sequential([
    layers.Conv2D(32,(3,3),activation='relu', input_shape=(28,28,1)),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64,(3,3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

fitting = model.fit(X_train, y_train, epochs=10,
 batch_size= 64,validation_split = 0.2)
```

In the code snippet above we have this structure for chosen sequential model, in this model we have stack of Conv2D and MaxPooling2D layer, in this example i will configure my CNN to process inputs of shape(28,28,1) which is the format of MNIST images, assuming grayscale images, by passing the argument input shape to the first layer. The ReLU activation function is to learn the low-level features such as edges and textures. The MaxPooling2D layer is to reduce the spatial dimensions. Dense layers take vectors as input 1D, while the current output is a 3D tensor, to do that i will first flatten the 3D output to 1D, then add one or more Dense layers on top, and a final Dense layer with 10 outputs. [Tensorflow].

Conclusion

In my journey to reach the highest accuracy score for an image classification model, i learned about building CNNs via keras, with the purpose of image classification using the mnist dataset, i have reached the highest accuracy score for the model by following the instructions in the book HOML and tensorflow keras web guidance. The secret was to find balance in terms of layers, to also avoid complexity and overfitting in the model, I had to keep it simple. Convolutional layers and pooling layers are the backbone of CNNs, and adding different types of layers creates a deep neural network that can learn the levels of features from the input data.

L08/ generalization_error.ipynb

In this exercise, I will be introducing an explanation of all important overall concepts in training.

Qa) On Generalisation Error

As mentioned in the exercise, the figure 5.3 provides an insight over the relationship between capacity and error in ML models. Lets explain the Concepts in the figure:
We have two types of errors:

- **Training Error**, which is the measure of how well the model will perform on the training data.
- **Generalization Error**, which is the model performance on unseen data, and it is estimated by measuring the error on a test dataset.

And as we can see in the figure the graph area is splitted in 2 zones, the **Underfitting Zone** on the left end of the graph is where the model is too simple to capture the underlying patterns in the data, thus both generalization error and training error are high.
Then we have the **Overfitting Zone** is where the model will start becoming too complex, and start to fit the data too closely. This means that the model may not generalize well to new, unseen data because it has essentially memorised the training data instead of truly learning the underlying patterns or relationships. [EXXACT]

Machine Learning algorithms will perform well when their capacity of functions is appropriate for the true complexity of the task that they need to perform and the amount of training data they are provided with, which brings us to the next point from the figure 5.3.

Optimal Capacity, which basically is the optimal polynomial degree, is where the model achieves a good balance between fitting the training data and generalizing to new data effectively. Below that degree in the polynomial regression is when the model is underfitting and higher than that degree is where the model starts fitting the noise in the training data, which leads to overfitting. Next is the **Generalization Gap**. As the model capacity increases, the gap between training and generalization widens, which leads the model to potential overfitting. And the two axes we have are **X/Capacity** for the model capacity (complexity) and **Y/Error** for the performance of the model.

Qb) A MSE-Epoch/Error plot

Let's analyse the code provided in the exercise.
In **part 1**, it generates data for polynomial regression using 90-degree polynomials. It creates a dataset with 100 samples, with some added noise. Then the code splits the data into training and validation sets.
Part 2 of the code is where we train the model. As mentioned in the exercise description, a 90-degree polynomial is used for polynomial regression because of its extremely high capacity model. The code uses Stochastic Gradient Descent(SDG) for training the model on given amounts of epochs.
Epochs are the number of passes a training dataset takes around an algorithm. The SGDRegressor is set up with warm_start, which allows the model to continue training from the current state, and makes sure the model retains its parameters from the previous iteration. The SGD model is also set with a constant learning rate to simplify the optimization process, then we have the training loop that iterates through the specified number of epochs, in each epoch, the model is fitted to the training data. We also have the Mean Squared Error

calculation for both the training and validation sets in each epoch, which measure how well the model fits the training data, and how well the model generalises to new, unseen data.
In **part 3**, the code plots the **Root Mean Squared Error** for both the training and validation sets, across epochs.**(learning curve)**, which helps evaluate the models performance over epochs, so we can identify trends like in Qa), such as underfit/overfit zones, generalisation gap etc.

Qc) Early Stopping

Early stopping is a regularisation technique for deep neural networks that stops training when parameter updates no longer begin to yield improvements on a validation set. [PaperWithCode]
To implement early stopping in the code, we need to do implement these few lines:

```
if mse_val < best_val_error:
            best_val_error = mse_val
            best_model = sgd_reg
            patience = tolerance
        else:
            patience -= 1
            if patience == 0:
                print(f"Stopping early at epoch {epoch}")
                break
```

The code here will be continuously evaluating the validation error after each epoch during training, and it will be triggered if the validation error does not show improvement. Which helps prevent overfitting.When implemented in the code provided in the exercise:

```
epoch= 247, mse_train=0.67, mse_val=1.35
epoch= 248, mse_train=0.67, mse_val=1.35
Stopping early at epoch 249
OK
```

Figure 2. Early stopping triggered in training

Qd) Polynomial RMSE-Capacity plot

The plot mentioned in the exercise shows the relationship between the model capacity and the root mean squared error on both the training and validation sets.
The X-axis is a representation of the model capacity, and Y-axis is for the RMSE, a measure of how well the model's predictions align with actual values.
We can also see the blue dashed line(training error) keep dropping as the model capacity increases, because a high-capacity model can fit the training data more closely. While the green solid line (RMSE) drops only until around capacity 3 and begins to rise again which indicates that the model starts overfitting the training data, leading to a rise in the validation error.

In figure 3 we can see the RMSE- Capacity after I changed the capacity to 10, we can still observe the same results as before, in terms of capacity 3 the validation RMSE begins to rise. But we can also see that the rise after 8 is big, which indicates the model would not have performed well.

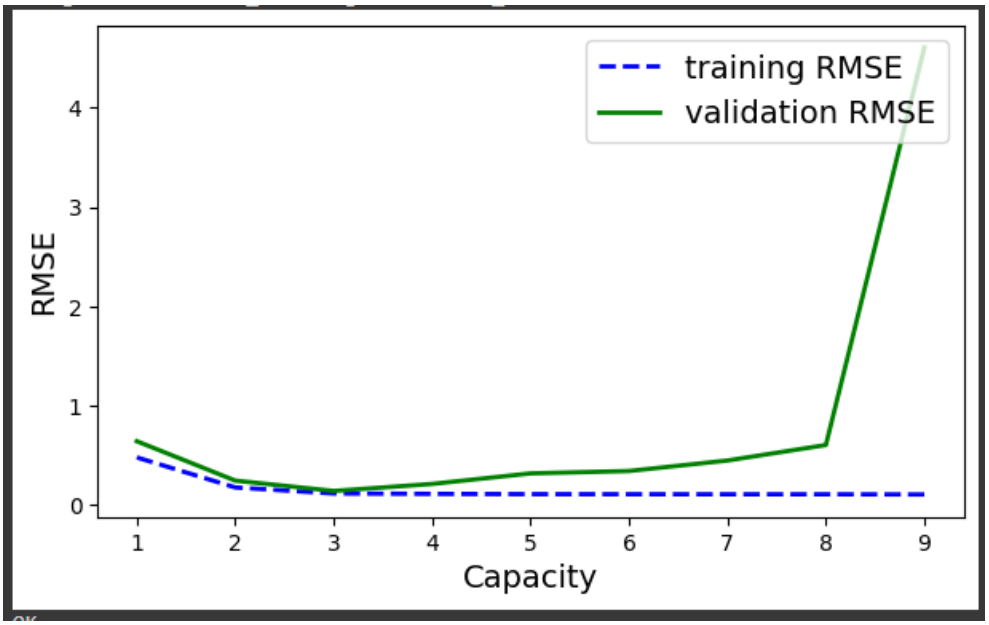

Figure 3. Capacity = 10

L08/ Capacity_under_overfitting.ipynb

In this exercise I am going to examine model capacity and under/overfitting, using linear regression with polynomial features.

Qa) Polynomial fitting code review

The code given in the exercise is for understanding the models behaviour during cross-validation. As usual it starts with generating data and adding random noise to simulate real life scenarios. Then polynomial regression models of different degrees (1,4,15) are being fitted using Scikit-learn's pipeline. The code creates a pipeline for each specified degree. The pipeline includes a polynomialFeatures transformer and a lineaRegression model. The fit model is called on the pipeline to train the model on the generated data. The models then are evaluated using cross-evaluation, with MSE used as the evaluation metric. Finally we have plots that are generated to visualise the fitted models, true function and the sample points for each polynomial degree. From the output for this code we can conclude that the model with degree 1 is underfitting, degree 4 is balanced fitting and degree 15 is overfitting due to its sensitivity to the training data's noise.

Qb) Capacity and Under/Over fitting concept

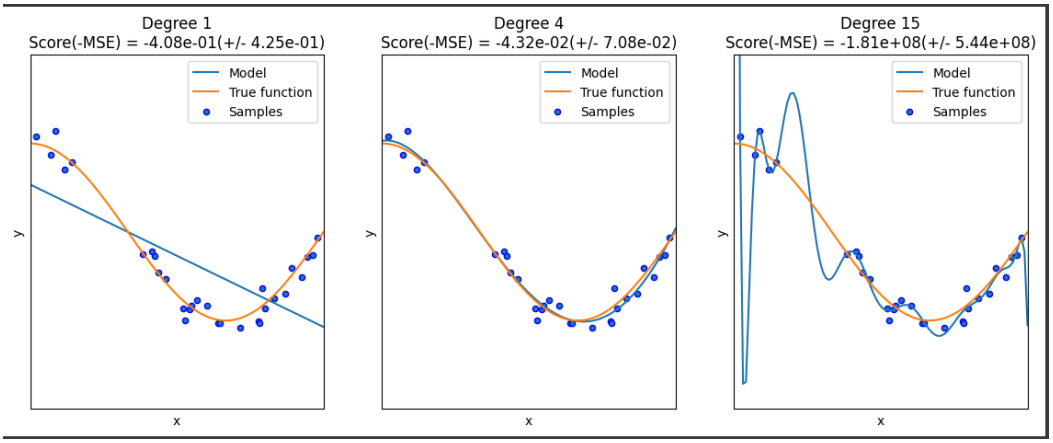

Figure 4. Plots from earlier exercise

The figure above illustrates the difference in the fitted models with different degrees.
In the subplot on the left end, we can see that the model is underfitting, which means that the model is too simple and unable to learn the underlying patterns in the data.
The subplot in the middle represents the model with medium complexity, it strikes a balance between simple and complex, and fits the data well. THe third model with the highest capacity (degree15) is highly complex, capturing noise and fluctuations, and will fail to generalise to new, unseen data.

Qc) Score method

Scikit-learn interprets the scoring function as a utility function more than a cost function. All scorer objects follow the convention that higher return values are better than lower return values, thus metrics which measure the distance between the model and the data, are available as neg_mean_squared_error which return the negated value of the metric [Scikit-learn]
The conceptual shift for the MSE from Cost function to score function is because, when using the cross_val_score function in scikit-learn, it expects a scoring function where higher values are better. And the MSE function becomes a score function by negating it. This allows using the same framework for both scoring and optimization.
If we try to set the scoring method to mean_squared_error, it will raise an exception as shown in the figure below. Because scikit-learn expects the function to return higher values.

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_scorer.py in get_scorer(scoring)
    430             scorer = copy.deepcopy(_SCORERS[scoring])
    431         except KeyError:
--> 432             raise ValueError(
    433                 "%r is not a valid scoring value. "
    434                 "Use sklearn.metrics.get_scorer_names() "

ValueError: 'mean_squared_error' is not a valid scoring value. Use sklearn.metrics.get_scor
```

Figure 5. Value Error when setting scoring method to mean_squared_error

The Theoretical Minimum Score value is 0, indicating a perfect match between the predicted and the true values. On the other hand, The theoretical Maximum is not defined, so it would

be the opposite to R2 score, because it can be any large positive number, which obviously indicates a very bad model. The high negative score for the degree 15 model tells us that the predictions are way off. So basically it expresses how far off the predictions are.

L09/ gridsearch.ipynb

In this exercise we will dive into ML algorithms and model selection via searching.

Qa) GridsearchCV

Let's break down Cell 2 in the exercise.
First the data setup with LoadAndSetupData('iris') to load the dataset for training and testing. Then we have the model initialization, model =svm.SVC(gamma=0.001), which initialises a support vector Classifier.Then the code utilises GridsearchCv to perform hyperparameter tuning for the SVC model.

```
grid_tuned = GridSearchCV(model,tuning_parameters,cv=CV,
                          scoring='f1_micro',verbose=VERBOSE, n_jobs=-1)
```

As mentioned earlier the model parameter is the svm.SVC. the tuning_parameters is a dictionary specifying the hyperparameter to tune in the provided range. The scoring metric used to evaluate the performance of the model during tuning. The n_jobs is the number of jobs to run in parallel. -1 means using all the CPU available cores.
Finally we fit the GridSearchCV with the training data and generate a detailed report on the best model found, its parameter and the score achieved during the grid search.

Qb) Hyperparameter grid search using SDG classifier

Now I am going to replace the svm.SVC model with SGDClassifier with a suitable set of hyperparameters for the model.

```
model = SGDClassifier()

tuning_parameters = {
    'loss': ['hinge', 'log', 'modified_huber', 'squared_hinge',
'perceptron'],
    'penalty': ['l1', 'l2', 'elasticnet'],
    'alpha': [0.0001, 0.001, 0.01],
    'learning_rate': ['constant', 'optimal', 'invscaling', 'adaptive'],
    'max_iter': [100, 1000, 10000]
}
```

The tuning parameters here are different from the SVC model, the SGD model requires four or five hyperparameters in the search space. These parameters are used to find the combination of hyperparameter values that maximises the performance metric f1_score.
In the figure below we can see the results for the GridSearchCv with the SGD classifier.

```
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

           0       1.00      1.00      1.00        16
           1       0.94      0.89      0.91        18
           2       0.83      0.91      0.87        11

    accuracy                           0.93        45
   macro avg       0.92      0.93      0.93        45
weighted avg       0.94      0.93      0.93        45


CTOR for best model: SGDClassifier(alpha=0.001, loss='squared_hinge', max_iter=10000, penalty='l1')

best: dat=iris, score=0.98095, model=SGDClassifier(alpha=0.001,learning_rate='optimal',loss='squared_hinge',max_iter=10000,penalty='l1')
```

Figure 6. Results of GridSearchCV with SGD classifier

Qc) Hyperparameter random search using SDG classifier

Now i am going to examine the RandomSearchCV instead of the GridSearchCV with the help of the code provided in the exercise.

```
random_tuned = RandomizedSearchCV(
    model,tuning_parameters,n_iter=20,
    random_state=42, cv=CV, scoring='f1_micro',
    verbose=VERBOSE, n_jobs=-1
)
```

And the results in the figure below, shows a lower score value than the GridSearchCV, which makes sense, that is because the GridSearchCv is more precise and slower in runtime, while RandomSearchCv is faster and not so precise.

```
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

           0       1.00      1.00      1.00        34
           1       1.00      0.94      0.97        32
           2       0.95      1.00      0.97        39

    accuracy                           0.98       105
   macro avg       0.98      0.98      0.98       105
weighted avg       0.98      0.98      0.98       105


CTOR for best model: SGDClassifier(alpha=0.01, loss='log', penalty='l1')

best: dat=iris, score=0.97143, model=SGDClassifier(alpha=0.01,learning_rate='optimal',loss='log',max_iter=1000,penalty='l1')
```

Figure 7. RandomSearchCV results

Qd) MNIST search Quest ||

In this exercise I will be experimenting with the MNIST dataset instead of iris. I will also examine the score values for the dataset.

```
X_train, X_test, y_train, y_test = LoadAndSetupData('mnist')

# Setup search parameters
model_mnist = svm.SVC(
    gamma= 0.001)
```

```
tuning_parameters_mnist = {
    'kernel': ('linear', 'rbf','poly'),
    'C': [0.1, 1, 10]
}

CV = 5
VERBOSE = 0

random_tuned_mnist = RandomizedSearchCV(
    model_mnist,
    tuning_parameters_mnist,
    cv=CV,
    scoring='f1_micro',
    verbose=VERBOSE,
    n_jobs=-1,
)

start_mnist = time()
random_tuned_mnist.fit(X_train[:10000], y_train[:10000])
t = time() - start_mnist
# Report result
b_mnist, m_mnist = SearchReport(random_tuned_mnist,X_train,y_train,t)
print(f"OK(randomized-search MNIST with Search time :{t:0.2f} sec)")
```

In the code above I chose the simple approach of the svm.SVC model, with a randomized search. I am also experimenting with different kernels and regularisation parameter C. And since the MNIST is a large dataset, I am using a subset for the quest X_train[10000], it will help speed up the search. Here are the results in the figure below.

```
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

           0       0.97      0.99      0.98      4826
           1       0.96      0.98      0.97      5492
           2       0.96      0.96      0.96      4875
           3       0.97      0.95      0.96      5024
           4       0.97      0.97      0.97      4820
           5       0.96      0.96      0.96      4413
           6       0.98      0.98      0.98      4831
           7       0.97      0.97      0.97      5104
           8       0.97      0.95      0.96      4783
           9       0.96      0.95      0.96      4832

    accuracy                           0.97     49000
   macro avg       0.97      0.97      0.97     49000
weighted avg       0.97      0.97      0.97     49000


CTOR for best model: SVC(C=0.1, gamma=0.001, kernel='poly')

best: dat=mnist, score=0.95150, model=SVC(C=0.1,kernel='poly')
```

Figure 8. GridSearch on the MNIST dataset

We can see in the figure that the best score for the SVC model is 0.951.

L10/ cnn2.ipynb

In this exercise, I am going to build an advanced CNN using roboflow with an existing dataset from the website. I will also be training a YOLOv8 object detection model on the custom dataset. I have chosen a Bicycle dataset from roboflow and trained the model on the dataset. Here is the results :

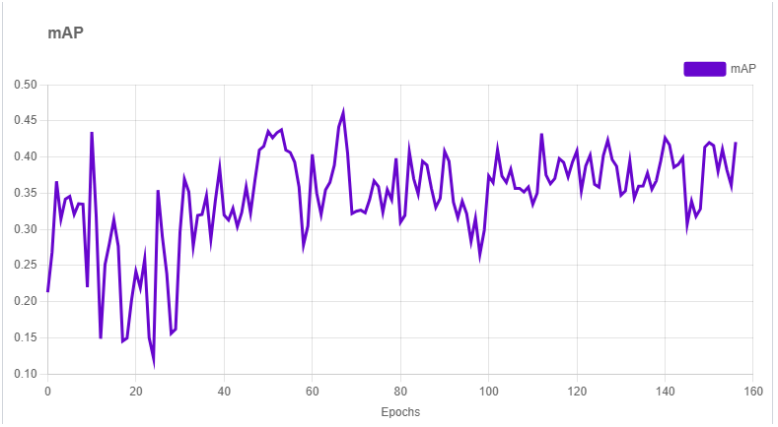

Figure 9. MAP graph for the model.

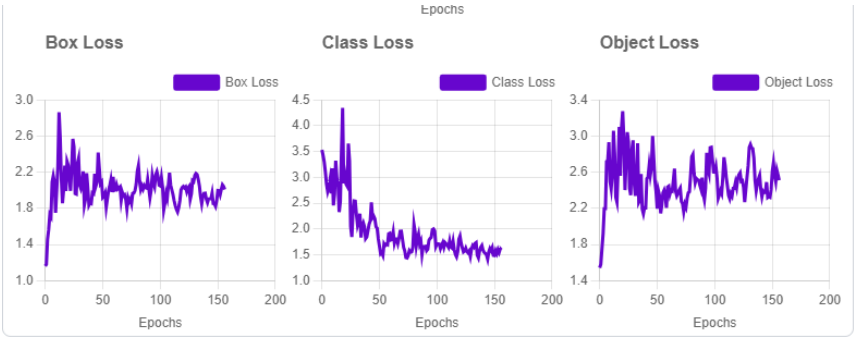

Figure 10. Box loss, class loss and object loss

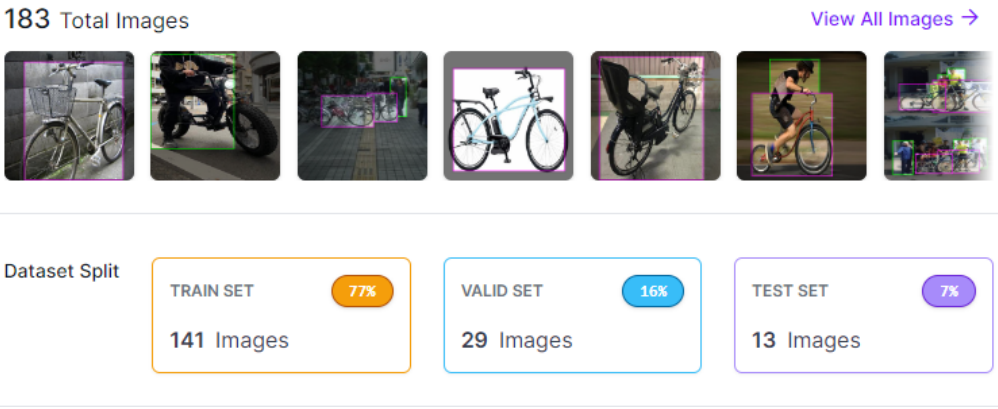

Figure 11. Dataset Details.

We can see in figure 9 the mean average precision is more unstable in the beginning, it is because the algorithm establishes the models concurrently. In Figure 10 we can see the object loss, which measures how well the model is deciding whether an object is present in the predicted area. Class loss measures the error in predicting the object class. Box loss refers to the error in the prediction of the object box parameter, such as size and position of the detected object. Lastly we have Figure 11 to show the dataset details with train/test/valid split