Project 3 - FYS-STK4155

Elias Roland Udnæs, Jacob Lie & Jonas Thoen Faber
(Dated: December 15, 2020)

In our previous project we created a Feed Forward Neural Network (FFNN) to recognise 8×8 images of hand-written digits. In this project we increase the complexity of 320×258 images by introducing seven different fruits (apple, banana, kiwi, mango, orange, pear and tomato). 15190 (2170 per fruit) images were reshaped into suitable resolution before entering the learning and testing process. Both colored and gray images were evaluated. Using our own produced and Keras' FFNN approach to solve the recognition problem had its limitation due to a vast number of adjustable parameters. We obtain a best accuracy score of 72 % for colored 100×100 images, which was computationally expensive to achieve. The accuracy of individual prediction varied between 50 % (apple) to 95 % (kiwi). We then introduce Convolutional Neural Network (CNN) that process the data differently. The prepossessing of images were done as before. For colored (gray) images with resolution of 50×50, the optimal parameters were found to be η (learning rate) = 0.01 (0.01), λ (regularisation l2) = 0.0005 (0.001) for 10 (10) epochs and batch size of 5 (5). Two convolutional layers and two max pooling layers followed by a dense layer and a output layer are used. Each convolution layer has a 3×3 receptive field and we use a pooling sise of 2×2 for reduction. The dense layer contains 100 neurons. The convolutional layers and the dense layer use ReLU as activation function and Softmax is used in the output layer. This setup ended with an optimal accuracy of 97.4 % (87.7 %). This model had the best prediction on orange (tomato) of 98.8 % (90.3 %) and worst on apple (apple) 0f 84.2 % (69.5 %).

I. INTRODUCTION

In our paper on Feed Forward Neural Networks (FFNN)[1], we analysed how a neural network could recognise handwritten digits from the MNIST data set. In this paper, we wish to explore more complex image recognition. From Kaggle, we found a dataset containing colored images of different types of fruit[2]. We wish to find if our network is capable of classifying these data.

For complex image recognition, fully connected neural networks like the FFNN face some challenges. A serious limitation on such networks is computational complexity. Fully connected neural networks do not scale well, as all neurons in layer $l-1$ are connected to all neurons in layer l. Therefore, a deep network is computationally costly. It also creates a black box, where it is hard to understand how the network makes its predictions. The solution to these challenges comes with introducing a different type of neural network.

The introduction of Convolutional Neural Networks (CNN) gave rise to a revolution within the field of machine learning. These types of networks are typically best suited for image recognition. The ability of giving machines "sight" and "hearing" has enabled self driving cars, better diagnostic methods for cancer treatment, direct speech translation etc. By using a CNN we can both detect image features with kernels/filters and downsample the images using pooling.

In this project, we will compare Feed Forward Neural Networks with Convolutional Neural networks. We start by training our own FFNN and a FFNN from Keras, and measure their performance on classification on the fruit dataset. Then, we compare the results with Keras' Convolutional Neural Network.

[1] Stochastic gradient descent and neural networks, Project 2 - FYS-STK4155 [1]

[2] Fruit Recognition, 44406 labelled fruit images [2]

II. THEORY

1. Convolutional Neural Network

In our paper about fully connected Feed Forward Neural Network [1], our goal was to predict handwritten digits in the MNIST data. The results were very promising with a peak accuracy of 98.1 % However, for this paper, the images we wish to recognise has both color and more detail. Therefore we will use a **Convolutional Neural Network**. The CNN assumes that its input is an image with pixel height, width and a channel axis. A normal channel axis for image recognition is RGB (Red, Green, Blue) channels and for each channel in the image there are three values between 0 and 255, which decides the strength of the individual colors.

The first step is to initialise kernels/filters. The filters in a CNN are the weights, which are updated to minimise the cost function. Their task is to pick up unique features in the image and represent them in a feature matrix 1. Often 3×3 or 5×5 filters are used. If the right features are extracted, then the network will predict the correct image, and cost is low. The filter slides over the image and takes the elementwise dot product between the pixel value and the filter value and returns a weighted sum:

$$\sum_i w_i x_i + b,$$

where x_i is not straightforward as the neurons share pa-

rameters (see [3] for detailed description). This is shown in figure 1. The figure shows the last step of making the feature matrix, when the filter has slid over the whole image. You see that each pixel element is dotted with its associated filter value, and each result is summed up to 4.
This example however, assumes stride = 1 and padding = 0. Where the filter only moves one step each iteration, and there is no padding with zeros, as the filter only moves one stride at the time, and will not pass the edges of the image. An example of such a filter is the vertical edge filter:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

The filters helps the neural network detect vertical differences in the image. An example of these filters can be seen in figure 2. If the image is more detailed, you would include more filters to detect enough unique features of said image. We cannot guarantee that we get a vertical edge filter, because the filters are initialized randomly. Therefore, we can not decide what features we wish to detect, but the randomness has an advantage. It makes it less likely that filters will detect the same feature, and a more complex image requires more filters. How the filters finds features is represented in figure 4 and 5.

An activation function is applied to the feature matrix to both limit the values allowed, and to account for non linearity (discussed in [1]).

To minimise computational cost, a pooling layer is applied, and its purpose is to downsample the feature matrix without loosing the features it represents. A 2×2 filter with stride = 2 is often used, it slides over the feature matrix and extract e.g. the maximum value within the 2×2 matrix (max pooling) as shown in figure 3. The pooling layer has another purpose than downsampling. E.g. if you were to recognise a 100 × 100 image using a fully connected neural network with two hidden layers (1000,100) and (100,10). The number of weights to tune would be approximately 10^7. Let us instead consider a Convolutional Network, where we pad the image in such a way that we keep the dimension of the image. If you apply a 2 × 2 pooling layer with stride = 2, then the filter will extract the largest of the four values it evaluates each stride. Then the image is reduced from 100 × 100 to 50 × 50. If the object you wish to detect in the image has been rotated, then pooling will still detect the deciding feature. Again, look at the example figure 3. If you were to rotate the values inside individual color matrix, the pooling filter would still extract the same maximum values. This ensures rotational invariance.

After convolution and pooling layer, we have a final downsampled feature map, that we can send through a fully connected neural network. The final feature map is flattened (i.e. vectorised) before entering the multilayer perceptron which we described in our paper on neural network [1].

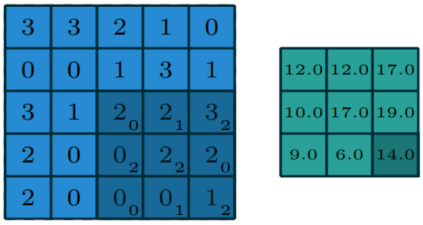

Figure 1. A visualisation of how the filter moves from one corner of the matrix in strides of 1, and takes the dot product between the filter value in the subscript and the image pixel value. The result of each dot product are summed together to form the feature matrix [4].

Figure 2. A visualisation of the filters in a Convolutional Neural Network, where their purpose is to find features in the image you wish to recognise. The filters are from our own CNN.

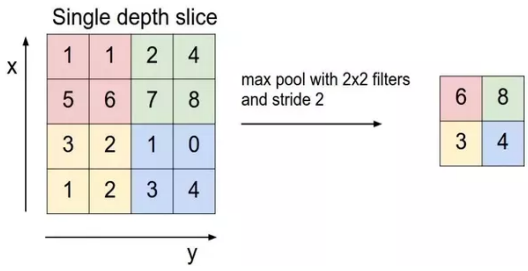

Figure 3. Example of a max pooling layer. The maximum value in each 2×2 matrix is extracted for image reduction. [5].

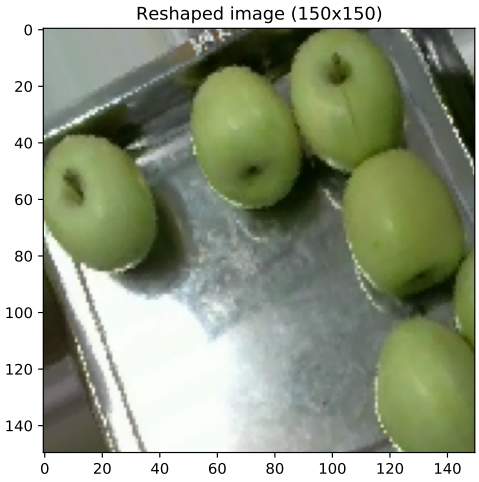

Figure 4. The original rescaled RGB image before being sent through the CNN.

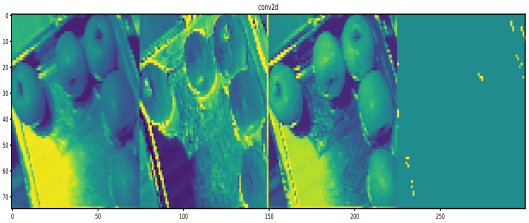

Figure 5. How the image looks when sent through a convolution layer, before being downsampled by the pooling layer. The convolution layer detects the edges of the image. The image is created using Keras' CNN (the code for visualisation was from [6])

III. METHOD

1. Data

The dataset we work with is pictures of fruit publically available through Kaggle [2]. This dataset consists of 44406 fruit images, captured on a clear background with resolution of 320×258.

The pictures were created under variable conditions to get a realistic dataset. These variables include light conditions, placement, and varying number of fruits.

We picked out seven of these fruits to work with, namely: apple, banana, kiwi, mango, orange, pear, and tomato. Figure 6 shows the image distribution of each fruit.

The size of the dataset made up of these seven fruits is 3.4 GB. This is a considerable size which might lead to memory issues depending on the computer you are working on. For a laptop, it is not possible to work with the entire dataset at once. The solution to this problem is described in the next section.

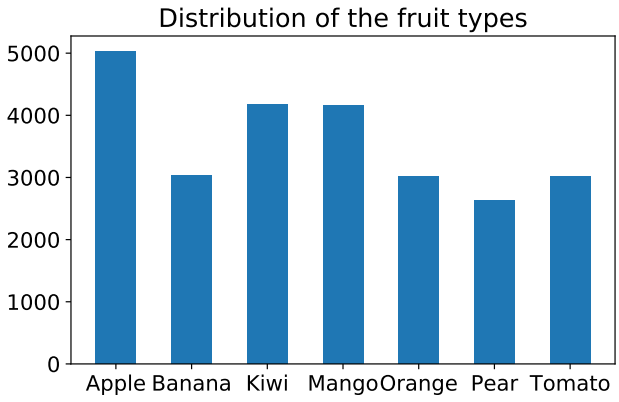

Figure 6. Number of images for each fruit.

2. Data Pre-processing

Before we can begin to train the models, the input data needs to be adjusted correctly. We created a class that configure the input data so that the models are trained, validated and tested on the same format of data.

The class is called `extract_data` and can be found in the `data_adjustment.py` script. The class takes primarily two inputs, namely `path_to_data` and `labels`. Both inputs must be arrays with strings as format. The first input contains the path to the folder containing the data sets. The second input contains the label according to those paths. It is important that `path_to_data` and `labels` are correctly ordered with respect to one another, else the models will name e.g. apple as banana. The program will then extract those files from the folders of interest and then store the data ready to be adjusted.

There are two additional inputs to the `extract_data` class which are `lim_data` and `from_data`. The first inputs sets the number of files to be extracted from each of the folders. The second input defines the numbering from where we want to retrieve data in each folder (e.g. from data #200 and the next `lim_data` data). These functionalities are made to prevent the memory to be fully allocated as they very often are limited.

In the `extract_data` class, another function is defined as `reshape` which is the main function in this script. It requires a single input `dat_size` which defines the new size of the data. When the function is called upon, it adjust the extracted data to an desirable resolution. Note that the data will converted to a quadratic form (e.g. 100×100). `dat_size` is set to 50 as standard.

Another function `gray` is created to re-scale the data by some dimension into a dimension of one. It is specifically used to convert colored images into a single colored image by taking the depth average of every pixels. The `shuffle` function simply shuffles the extracted data and the `flatten` function reduces the dimension of the data to vectorised form. The reason for flattening is because the data will enter a multi-layer perecptron that does not

accept tensors as input. It is crucial for the Feed Forward Neural Network models.

The last function `delete_all_data` handles the problem we get from the limitation of memory on the computer. Whenever the models are finished with the training process on some portion of the entire dataset, we delete the data allocated on the memory so that new data may be extracted.

A typical configuration of an image is shown in figure 7. The image is reshaped into a suitable size and figure 8 visualise the same image but as single colored.

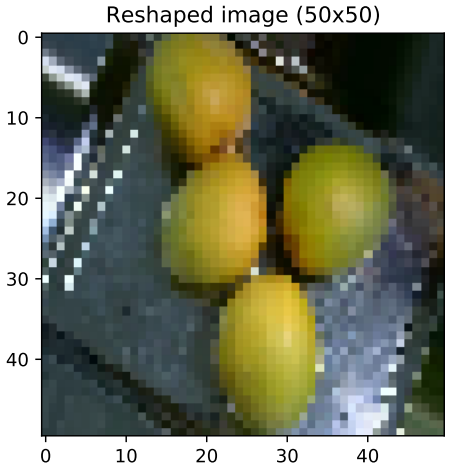

Figure 7. Image data of mango's reshaped to a resolution of 50 by 50 pixels with three colors.

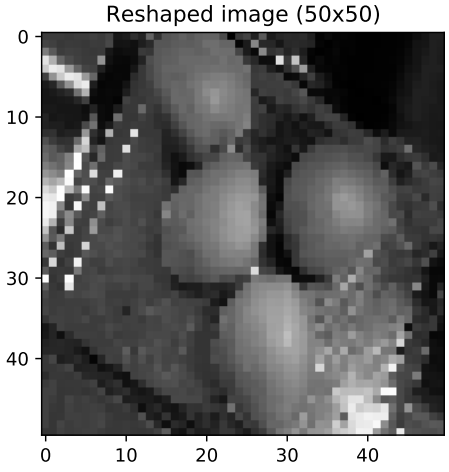

Figure 8. Same as figure 7 but reduced to gray-scale.

To make sure that the models do not train more on one item than the other, the algorithm counts the number of file in each folder. Then, it extract the number of files from each folder corresponding the folder with the lowest number of files.

3. *Feed Forward Neural Network*

First, we train a feed-forward neural network on images of the seven fruits of our choosing. The network we work with is the one presented in our paper on Neural Networks [1]. This time, we implement Glorot initialisation [7] of weights in the hidden layers of the network, and we initialise biases to zero. The weights are initialised by

$$\texttt{weights} = \mathcal{N}(\sigma, 0)$$

with shape (`inputs`, `outputs`) to the layer. If there are many inputs and outputs, it is advantageous to have a small σ. Glorot initialisation solves this by defining the standard deviation as

$$\sigma = \frac{2}{\texttt{inputs} + \texttt{outputs}} .$$

We use rectified linear unit (ReLU) as activation function in the hidden layers, and softmax activation function in the output layer. As a test, we also try to train the network without Glorot initialisation in the hidden layers.

Hidden layers are initialised with `DenseLayer` from the `network` module. A hidden layer with n `inputs` and m `outputs` (nodes) is initialised by

```
hidden_layer = DenseLayer(n, m, relu(),
                          Glorot=True)
```

We feedforward pictures from the fruit data set, and use back-propagation to train the network. In back-propagation, we use a constant learning rate $\eta = 0.0001$ and a regularisation parameter $\lambda = 0.001$.

The images are reshaped to a lower resolution with $p \times p$ pixels. This greatly reduces the time to train the network, but can have a negative impact on learning. If images are converted to too low resolution, it can become impossible to extract information from them.

The reshaped images consist of three matrices containing RGB values. The shape of each color-image is $(p, p, 3)$. To further reduce training time of the network, we can convert the images to gray-scale, meaning that each pixel only has one value. This reduces data-points in each picture by a factor 3. We explore how this impacts performance of the network by comparing how well the network recognises different fruits when it is trained with gray-scale or color image input.

Due to memory limitations, we cannot train the network on the whole dataset at once. Therefore we have to split the dataset into smaller bathces. We pick out each batch with the same number of each of the seven fruits. Then we train the network on this data for an appropriate number of epochs. We split each batch into training and validation data. This way we can observe if overfitting is happening by testing the network with validation data. After we have trained the network on a

large portion of the pictures, we test the network on the remaining pictures.

We repeat the same analysis by using a Feed Forward Neural Network set up with Keras [8] with Tensorflow backend [9]. Both networks have the same number of hidden layers and nodes. We have four hidden layers with nodes [3000, 1000, 200, 10].

The images are reshaped to 50×50 pixels with and without color. This corresponds to 7500 and 2500 input features to the network. We also train the network on gray-scale images with 85×85 pixels. This translates to 7225 features, roughly the same as the 50×50 RGB images. We also test the network with a much larger resolution, 100×100 pixels with color. We do not expect this to be efficient.

We pick out 1420 images of each fruit and split this set into smaller batches. Then we train the network for 5 epochs on every batch of pictures. In total, we train the network on 1136 pictures of each fruit type, splitting 80% to train and 20% to validation for each batch. Then we test the network on the remaining fruit images.

4. Convolutional Neural Network

The second model we introduced above is Convolutional Neural Network. This method has the advantage of reducing parameters to be adjusted when training the models independent of input size. We use Keras [8] to set up the Convolutional Neural Network with the same objective to solve the image recognition problem.

To set up the CNN, we start by creating a class `CNN_keras` which can be found in the `CNN_class.py` script. Before we add layers to the CNN, the model must be initialised by using the `tf.keras.Sequential` functionality. This function will let us add additional layers to the models easily by using the `add` utility that Keras provide.

The initialisation inputs to the class are `input_shape`, `receptive_field`, `n_filters`, `n_neurons_connected`, `labels`, `eta` and `lmbd`. The first input defines the shape of each image that the model will be trained and tested on. It is important that every image has the same dimensions as the input in the model stays fixed. Each image go through a data pre-processing as detailed described in III 2. Next input is the size of the receptive field, also known as the filter. We will consistently use the filter size of 3×3 but it might be changed if desirable. Next input is the number of neurons in the dense layers that comes after the reduction of convolution an pooling. The dense layers requires input on a vectorised form meaning that the data from the images must be flattened after the convolution and pooling. Keras provide this functionality by calling `Flatten()` inside utilised on `tf.keras.Sequential`. The `labels` input contains an array with the unique names of the different data that the models will be trained to recognise. Lastly, `eta` is the learning rate parameter which will be constant during the learning process and `lmbd` is the regularisation parameter. We will use the L2 penalty as the regularisation parameter during this project.

We will try various models with different sets of layers to be trained but common in all models is that the input layer is a 2D convolutional layer and the output layer is a dense layer with Softmax as activation as we are considering a classification problem. The first convolutional layer computes output values by using rectified linear units (ReLU) as activation function. The loss function in the output layer used are the categorical cross-entropy loss meaning that we also need to set up a one-hot vector. The one-hot vector is created when the images are preprocessed. We use Stochastic Gradient Descent method (SGD) as the gradient descent method in all models.

All weights in the CNN will be initialised with the Glorot initialisation as standard.

The number of epochs of training as well as the batch size will be considered during this project when training the models. Our goal is to prevent overfitting but still make sure the models are trained in a satisfactory manner.

5. Accuracy Map and Confusion Matrix

Our results are best shown when implementing accuracy maps and confusion matrices. The accuracy maps are created to configure parameters when training either of the models. We will mainly look into parameters such as learning rate (η) and the regularisation parameter (λ) as well as the number of epochs and batch size to evaluate the optimal parameters.

By creating a heatmap of the accuracy scores as function of two parameters of choice, we may analyse the plots to make sure models gives satisfied predictions. Our objective is to train models of high excellence by achieving an accuracy score close to 1.

To visualise our optimal results, we rely heavily on a plot called the confusion matrix. The confusion matrix is a way of visualising predicted versus true labels. Predicted labels are plotted in the columns of the matrix, while the rows hold the true labels. Number of predictions on a fruit divided by total fruits are plotted along the rows.

This way, correct predictions are reside on the diagonal of the matrix. The diagonal then holds accuracy for each fruit. The confusion matrix is very useful, as we can observe not only correct predictions, but also what is predicted when a prediction is false.

IV. RESULTS

1. Feed Forward Neural Network

Using our own Feed Forward Neural Network, we trained the network with different input shapes of the

images. First off, we tried training the network without Glorot initialisation in the hidden layers and with ReLU as activation function. This quickly resulted in overflow, known as exploding gradient. For the rest of the analyses, we trained the network with Glorot initialisation.

We recorded the accuracy on the validation data, as shown in figure 9. We see that accuracy increases with the number of images trained on, and stabilises around 1000 images. Feeding the network with gray-scale 50×50 images gave poor accuracy, while color 50×50 and gray-scale 85×85 performed better with similar trend in accuracy. After finished training, we tested the network on 71 images of each fruit. Accuracy on test data was 67% for color 50×50, 50% for gray-scale 50×50, and 69% for gray-scale 50×50.

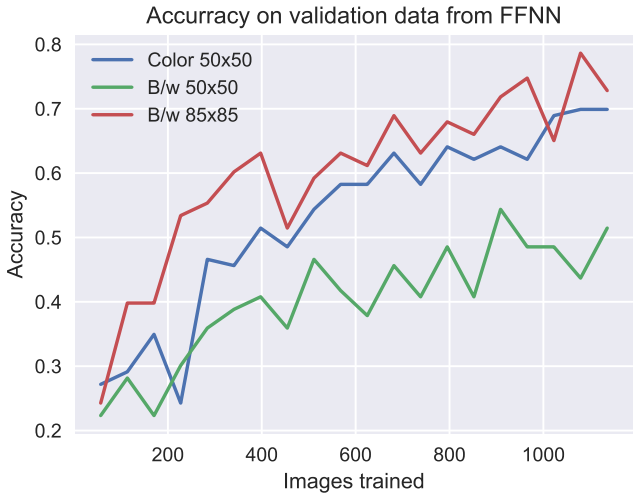

Figure 9. Accuracy on validation images using our own network. Input data are images with 50×50 pixels color and gray-scale (B/w for black and white), and gray-scale images 85×85 pixels.

We plot a confusion matrix for the input format 85×85 pixels gray-scale, which gave best accuracy on the test data. The confusion matrix can be seen in figure 10, where we can see correct predictions along the diagonal. Here, we see the network performing well on the fruits kiwi and pear.

Moving on, we repeated the same analysis with a Feed Forward Neural Network from Keras. Accuracy on validation data versus amount of images trained on is shown in figure 11. We see the same trends as with the training of our own network. Accuracy increases with number of images trained on until around 1000. Feeding the network with gray-scale 50×50 images gave poor accuracy here as well, while color 50×50 and gray-scale 50×50 performed better with similar trend in accuracy. Accuracy on test data was 78% for color 50×50, 54% for gray-scale 50×50, and 75% for gray-scale 85×85. Keras gave approximately 10% better accuracy on test data than our own neural network for color 50×50 and gray-scale 85×85

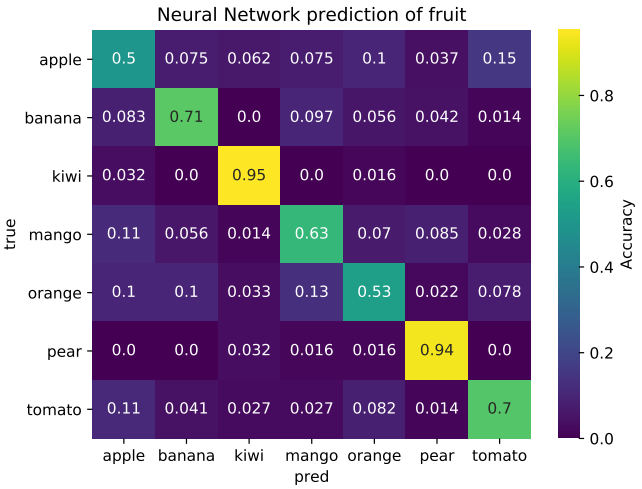

Figure 10. Confusion matrix from prediction accuracy with our own neural network.

input.

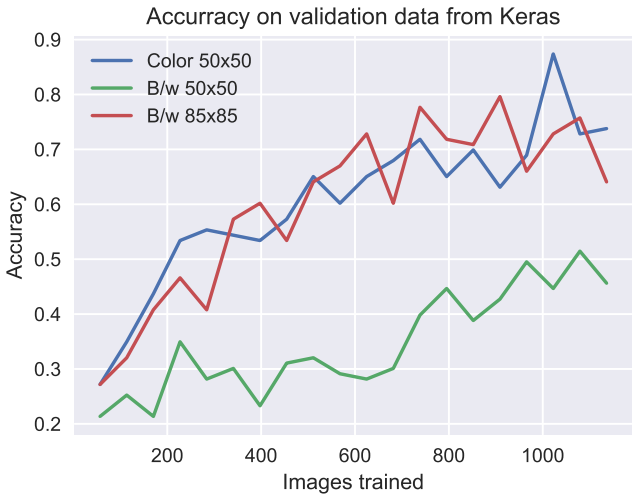

Figure 11. Accuracy on validation images using a Keras Feed Forward Neural network. Input data are images with 50×50 pixels color and gray-scale (B/w for black and white), and gray-scale images 85×85 pixels.

Again we plot a confusion matrix for the input format which gave best accuracy on test data, which was 50×50 pixels color for the Keras neural network. This confusion matrix is presented in figure 12. This network also performed surprisingly well on kiwis, at 98% correct predictions.

Lastly, we tried running our own network with 100×100 color images on a 16 Gb, Intel i7-8565U (8) @ 4.600GHz computer, and the training took ≈ 3.4 hours. This increased accuracy on test data by 3% with our own network, to an accuracy of 72%. Trying 100×100 color images as input on the Keras network did not work.

We plotted a confusion matrix for the 100×100 color

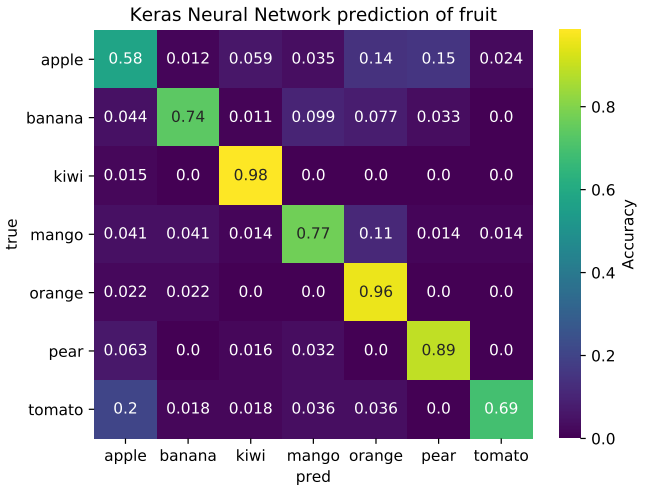

Figure 12. Confusion matrix from prediction accuracy with Keras neural network.

input data in figure 13. Here we can see an overall improvement from the smaller input data to our own network. These results are still not as good as the 50×50 RGB input to the Keras network however.

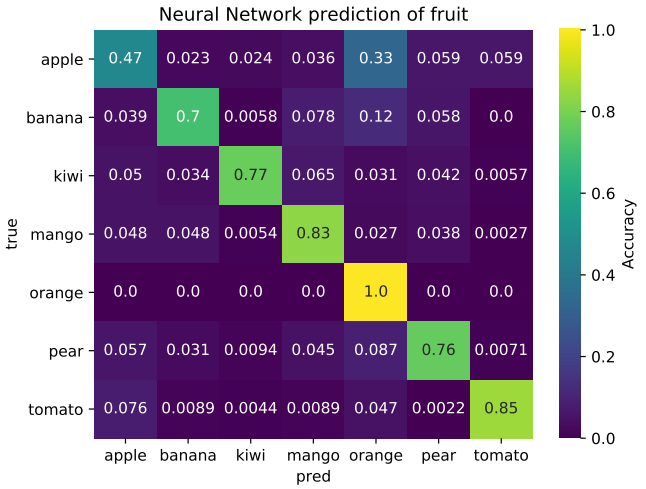

Figure 13. Confusion matrix from prediction accuracy with our own neural network using 100×100 color input.

2. *Convolutional Neural Network using Keras*

After running a Keras CNN with two 3 × 3 kernels, two 2 × 2 max pooling filters and one hidden layer, we achieved the optimal accuracy of 97% for $\eta = 0.01$ and $\lambda = 0.0005$ as seen from figure 14. It was achieved using 20 kernels, 100 neurons in the hidden layer with ReLU as activation function and Glorot weight initialisation. The network used ≈ 4 minutes on training.

After trained the model with a various sets of epochs

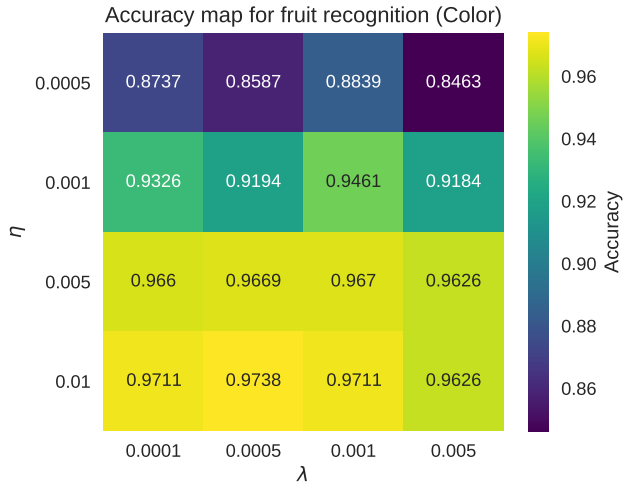

Figure 14. Accuracy for different learning rates and regularisation for Keras' Convolutional Neural Network with colored images.

and batch size with the optimal η and λ, it was found that 10 epochs with batch size 1 gave optimal results as seen from figure 15. However, for reducing computational cost, we varied the batch size. The best accuracy of the test data were 97 %. We note that increasing the batch size to 3 gave almost identical accuracy. Increasing the batch size decreases the computational time as more data is trained at once.

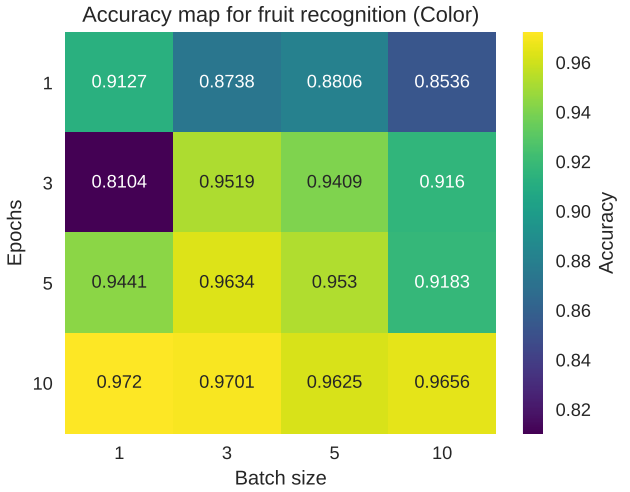

Figure 15. Accuracy for different number of epochs and batch size for Keras' Convolutional Neural Network with colored images.

Using the same set of parameters, we produced equal accuracy map but with single color images as input. The results are shown in figure 16 and 17. With $\eta = 0.01$, $\lambda = 0.001$, epochs = 10 and batch size = 3, we obtain

best accuracy scores of 88 % and 87 % for the two maps respectively.

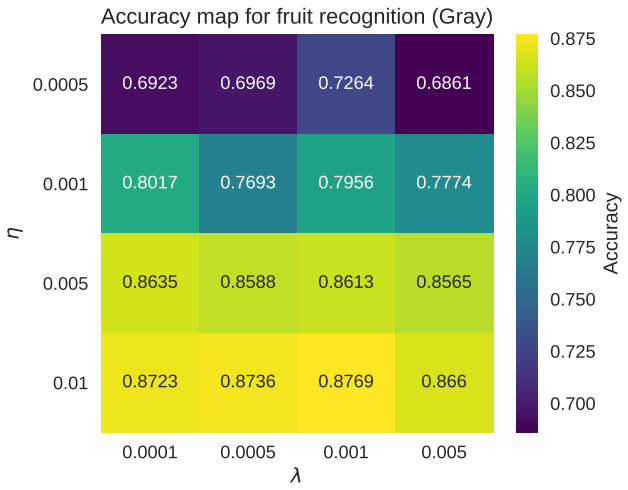

Figure 16. Accuracy for different learning rates and regularisation for Keras' Convolutional Neural Network using single colored images.

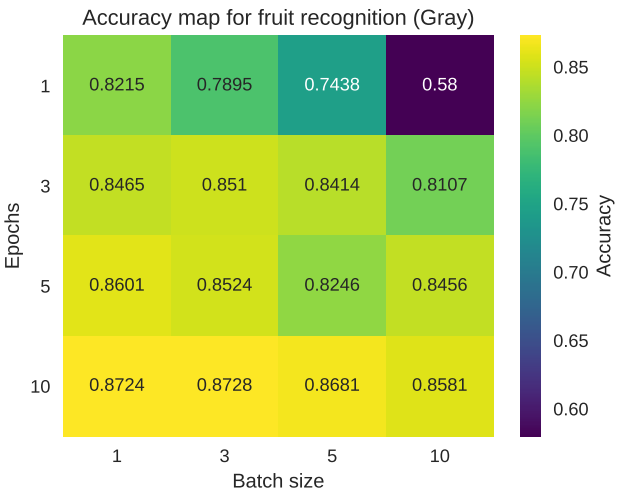

Figure 17. Accuracy for different epochs and batch size for Keras' Convolutional Neural Network using single colored images.

With optimal η, λ, epochs and batch size, we produced the confusion matrix for both colored and gray images shown in figure 18 and 19 respectively. The true labels were compared to predicted labels. The prediction were generally above 91 % in accuracy but the only deviation were apples that only peaked at 84 %. The best prediction were achieved with oranges at 99 %.

To justify the use of only 10 epochs, we plotted training accuracy against number of epochs shown in figure 20. The data was obtained using the same CNN with 20 kernels, 100 neurons in hidden layer and Glorot initialisation. The different lines represent number of images used for training. We see that the accuracy saturates around 7.5 epochs. Notice the red line, as it represents 1000 images trained. We also plotted validation data accuracy against number of images trained in figure 21. The peak accuracy is reached around 1000 images trained, which indicates that 10 epochs gives satisfied results.

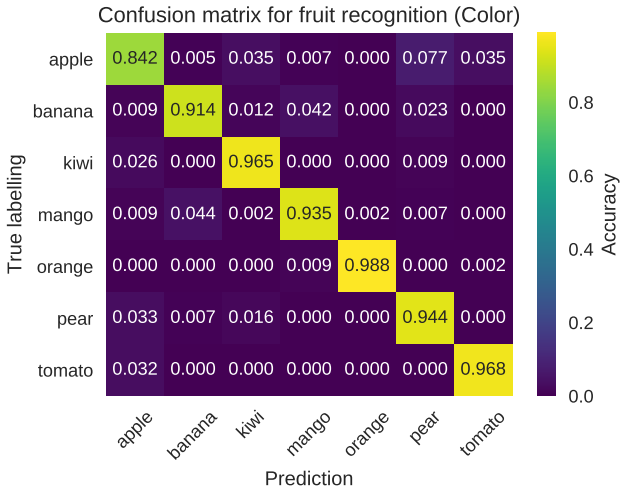

Figure 18. Confusion matrix for colored images. The diagonal represents the accuracy of the predictions of fruit. All other values should be close to zero.

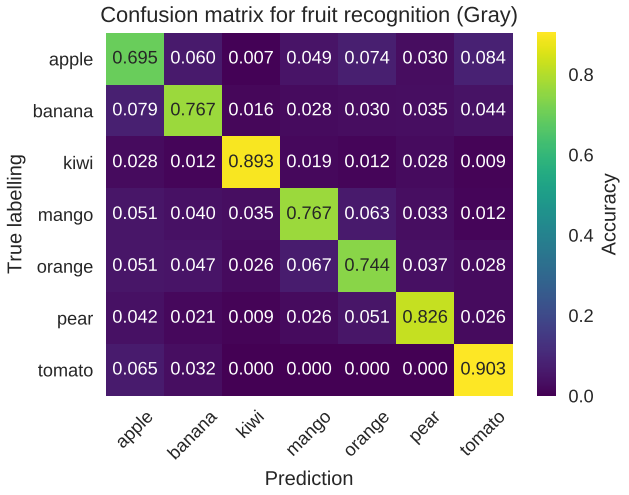

Figure 19. Confusion matrix for gray images. The diagonal represents the accuracy of the predictions of fruit. All other values should be close to zero.

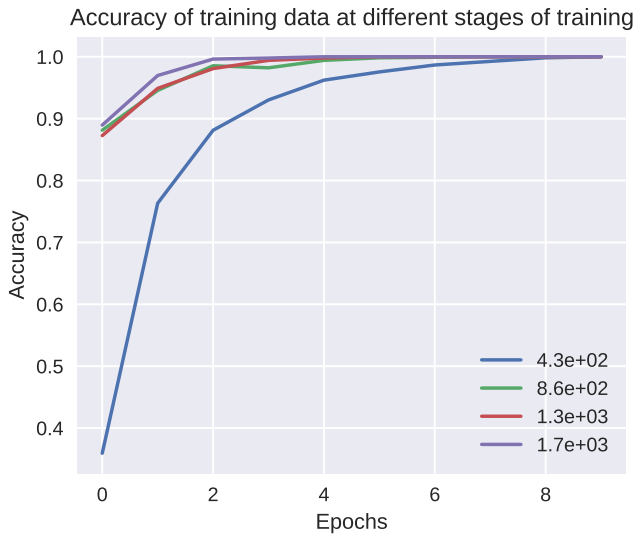

Figure 20. Accuracy of training data at different stages of training. The blue line represents 430 images trained, while the purple line shows accuracy when 1700 images have been trained. The lines build on each other and as more images have been trained, the accuracy saturates faster.

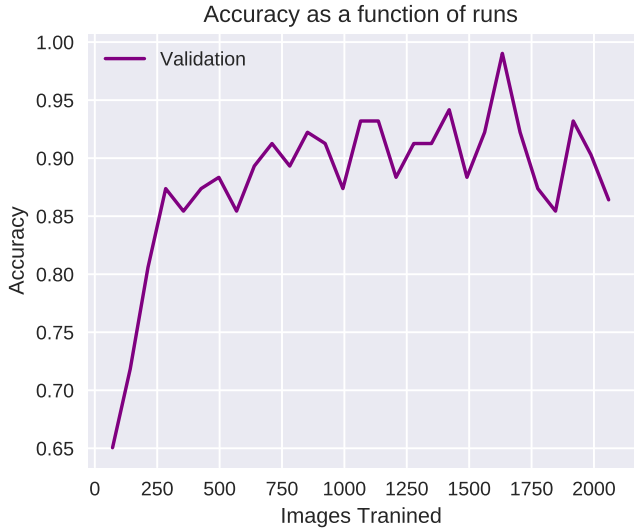

Figure 21. Accuracy on validation data plotted against images trained. About 70% of the images were used for training. The figure helps indicate if we get overfitting during training.

V. DISCUSSION

1. *Feed Forward Neural Network*

The models that were trained using our own Feed Forward Neural Network did not produce an accuracy greater than 72% on test data. Using Keras, we achieved an accuracy 78%. For practical purposes, these networks' performance are unacceptable. The only fruit which the networks consistently predicted well was kiwi. This may be due to kiwi being a much darker than the other fruits. Therefore the network may predict this fruit based on the average brightness of all pixels.

From trial and error, we experienced that complex networks with many hidden layers and neurons were required to get acceptable predictions on the fruits. The great number of inputs that corresponds to the total pixels in the images may cause difficulties for the network to learn. Every pixel is connected to every neuron in the first hidden layer where each connection has a unique adjustable weight. If the model has $50\times50\times3 = 7500$ pixels as inputs, a single neuron in the first hidden layer will adjust 7500 of its connected weights.

For high image resolutions a standard laptop quickly runs out of RAM, making it incapable of training the network. We found that an input resolution of 50×50 RGB values were suitable in terms of memory allocation and computation time. However, reducing image size was a limiting factor in terms of network accuracy. Increasing the input to 100×100 RGB pixels improved accuracy on test data to 72% for our own network. This was a small improvement at a high computational cost. Trying to perform the same analysis with Keras proved unfruitful as the computer crashed.

To increase accuracy without increasing computational cost, we employed a different model. Namely, the Convolutional Neural Network.

2. *Convolutional Neural Network using Keras*

Training a CNN, required tweaking many parameters. We tested different number of kernels, number of convolutional layers, number of neurons in hidden layer, learning rates, penalties, weight initialisation and activation functions. The optimal results however, were achieved using 20 randomly initialised 3×3 kernels organised in two layers, two 2×2 max pooling filters, $\eta = 0.01$, $\lambda = 0.0005$, Glorot initialisation and ReLu activation function. Finding the right combination of hyper parameters is the crutch of machine learning, and the results shown in figure 14 and 15, indicates satisfactory hyper parameters.

Training a CNN on gray images were generally poor in relation to training on colored images as compared between figure 18 and 19. The greatest accuracy of gray images never exceed 90 % on test data. A possible explanation to this may be due to lack of information of

the data. The gray-scaled images limits the model to see pattern in edges and not in the color contrast. This may be a challenge when the models are trained on lower resolution as well as the number of total images are limited.

3. FFNN versus CNN

After running our own FFNN, Keras' FFNN and Keras' CNN, we noticed the importance of two key factors. To obtain peak accuracy on test data with our own network (67%) it needed 4 hidden layers. For a 50 × 50 colored image of 7 fruits, the layers was (7500,3000), (3000,1000), (1000,200), (200,10). In total ≈ 26 million weights and biases needs to be adjusted. Compared to the CNN network, we achieved a peak accuracy of 97%. The set up were two convolutional layers each with 20 unique 3 × 3 kernels with zero padding. Additionally two 2×2 max pooling layers with stride = 2 and one hidden layer of 100 neurons were implemented. The total number of weights in a CNN is independent of the resolution of the image. It is dependent on the depth of the image tensor as well as the size and amount of kernels. We will compute the total parameters for our setup taking parameter sharing into consideration [3]. From the first convolutional layer, we get:

$$\text{Parameters}_{\text{ First layer}} = (3 \cdot 3) \cdot 20 \cdot 3 + 20 = 560.$$

In a more detailed manner, the calculation becomes (kernel width × kernel height) × kernels × image depth + kernels. We add an additional number of kernels to the equation as each kernel has their own unique bias. The max pooling layer does not have any adjustable weights on their own. When entering the second convolutional layer, we now have a depth of 20 feature maps which results in:

$$\text{Parameters}_{\text{ Second layer}} = (3 \cdot 3) \cdot 20 \cdot 20 + 20 = 3620.$$

After the convolutional and pooling layers, we now have adjusted the image data to a tensor of 12×12×20. Flattening these data result in a total of 2880 neurons connected to 100 neurons in the hidden dense layer. That gives additional 288000 (weights) + 100 (bias) parameters to be adjusted. The output layer has 7 neurons giving 700 (weights) + 7 (bias) parameters. In total, we thus have:

$$\text{Total Parameters} = 560 + 3620 + 288100 + 707 = 292987$$

adjustable parameters which is a great decrease in relation to the FFNN by a factor of approximately 88.

The second noticeable advantage is time consumption. Keras CNN was 205 min/4 min = 51.3 times faster than our own network, which is a massive improvement.

VI. CONCLUSION

We have implemented two different methods to solve the image recognition problem. These are Feed Forward Neural Network and Convolutional Neural network. Through analysis of the different networks applied on fruit recognition, we observed noticeable differences in performance. We have set up the Feed Forward Neural Network by using script from previous project as well as using the functionality that Keras provide. Keras were also used to set up the Convolutional Neural Network.

The learning schedule from the Feed Forward Neural Network were rather complex as the number of inputs were great, hence requires a lot of parameters to be adjusted. The vast number of inputs soon lead to complication in the learning process as the network ended with poor predictions. This is where the Convolutional Neural Network comes handy. This method is independent of the resolution of the input images and therefore reduce the total adjustable parameters greatly.

A Feed Forward Neural Network is sufficient for predicting smaller, less detailed images (8 by 8 single colored images [1]) but fails as the complexity of an image increases.

The Convolutional Neural Network generally ended with good prediction on the image recognition problem but also had certain drawbacks. For instance, 10 % of the images of apples were mistaken as either kiwis or pears. Depending on the need of great accuracy for different environments, these result may in many occasions not be sufficient.

We soon noted that our everyday personal computer had its own limitation as the memory quickly were fully allocated. The solution to this problem were to consider only a small amount of the total dataset at a time. This is similar to how the models is trained in practice as we define the neural nets to process a limited portions of data at a time, also known as batches. A possible future objective could be to analyse images of higher detail (resolution, more features on image, etc.) by using machines with greater specs that e.g. the University of Oslo provides at various institutes.

In summation the models trained using convolutional neural network has the benefits of decreasing computational cost, improve accuracy and reduce time consumption.

During this fruitful project we strengthen our knowledge in machine learning even further by introducing a new methodical approach. We noticed that a different setup than FFNN are highly recommended if we were to consider data with great complexity as e.g. images of fruits. If the number of feature inputs are small ($<$ 100), then FFNN is sufficient [1]. If the feature inputs are great ($>$ 100), we need to consider other methods i.e. CNN.

[1] E. R. Udnaes, J. Lie, and J. T. Faber, "Project 2 - fys-stk4155," 2020.
[2] C. Gorgolewski, "Fruit recognition." http://kaggle.com/chrisfilo/fruit-recognition/, Feb 2020. Accessed: 2020-12-03.
[3] F. F. Li, "Convolutional neural networks (cnns / convnets)." https://cs231n.github.io/convolutional-networks/.
[4] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," 2016. cite arxiv:1603.07285.
[5] A. Karpathy *et al.*, "Max pooling." https://github.com/cs231n/cs231n.github.io/blob/master/assets/cnn/maxpool.jpeg, 2015.
[6] "Convolutional neural network: Feature map and filter visualization."
[7] J. D. McCaffrey, "Neural network glorot initialization." https://jamesmccaffrey.wordpress.com/2017/06/21/neural-network-glorot-initialization/, 2017.
[8] F. Chollet *et al.*, "Keras." https://github.com/fchollet/keras, 2015.
[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems." http://tensorflow.org/, 2015. Software available from tensorflow.org.

VII. SOURCE CODE

Source code and additional results can be found in the github repository https://github.com/jacobllie/FYS-STK4155.