Image Recognition with Convolutional Neural Network

Stian Aannerud, Marius Bjerke Børvind, Dennis $\hat{H}$ Fremstad and Andreas Wetzel

December 18, 2022

The aim of this project is to construct a Convolutional Neural Network (CNN) and train it to a dataset containing pictures of cards to see if we can make a network that recognizes the suit and number of a card. To do this we initialize our CNN with three convolutional layers and three pooling layers with a 3×3 kernel for the convolutional layer and max pooling over a 2×2 region for the pooling layer. We use the identity matrix as activation function for the hidden layers and the output layer, MSE as the cost function and the Xavier method to initialize the kernels and biases. We use the adaptive learning rate algorithm Adam with an initial learning rate of 10^{-3}. To reduce the computation time we reduce the size of the image from a 256×256 image to a 50×50 and use gray-scale. With this setup we test our network against a binary case, in which case we get an accuracy of 95%, and all cards with 1040 training cards and 520 testing cards, in which case our network yields an accuracy of 39%. In addition we make a decision tree with bagging and compare it to our network. Using decision trees we get an accuracy of 57%. This can be matched by our network by increasing the size of the input images, increasing the number of cards we train over, training over more epochs and including color.

1 Introduction

Convolutional Neural Network (CNN) is a class of artificial neural network in deep learning. CNN is the chief support of many modern computer system, and is most commonly used for image analysis. In recent years, CNN has evolved greatly, solving problems that would not have been thought possible for a computer system in the past, such as diagnostic methods for cancer.

In this project, we will study a classification problem which involves image recognition for a hand-selected dataset which contains various decks of cards. The main aim of this project is to develop our own CNN code to solve our classification problem and explore image recognition. We will also use Decision trees with bagging to solve our classification problem and compare the results.

The data is briefly described in the Data section. We go through the theory of CNN and decision trees in the theory section. We move on to explain our methods for building the CNN used in the method section. The quality of our CNN is tested and discussed in the results & discussion section.

All code used in this project can be found at our GitHub page `https://github.com/frdennis/FYS-STK4155`.

2 Data

Since we are dealing with image recognition in this project, we have chosen a data set consisting of images of playing cards from the Kaggle website [5]. The data set contains 53 classes, all possible cards and the joker. We do not include the joker in our analysis. The images are split into a training set with 7624 images, and a test and validation set, both with 265 images. An example image is shown in Figure 1.

Each image has size $224 \times 224 \times 3$, meaning 224×224 pixels and 3 RGB values. To decrease the amount of computation as much as possible, we chose to reduce the size of the images to 50×50 pixels before applying them to our CNN. We also converted the images to grayscale in order to decrease the dimensions of the data.

Figure 1: Example of a cool card!

3 Theory

The CNN is similar to the feed-forward neural network (FFNN) explored in project 2, but with some extra steps. The aim with these steps is to only take into account the most important features of the images. This is done by having multiple filters in a layer that scan through the input image and outputs a feature map for each filter. This is known as convolution. The size of the images can then be reduced to decrease the amount of computations needed, through either selecting the maximum or mean value over a small region of pixels. This is known as pooling.

3.1 Convolutional layer

The aim of the convolutional layer is to extract the most important features in the input image. This comes in the form of feature maps, where the feature in question is the most prominent part of the image.

Since the input is 2-dimensional we require 2-dimensional weights as a kernel. To take the input image and output several feature maps we add an additional dimension, making the kernel 3-dimensional. In this case we must let each 2-dimensional kernel scan the image independently.

The way the "scanning" of the image works is the following; For each $N \times N$ region in the original image we have a $N \times N$ kernel that reduces the original pixels to a single weighted pixel. The values of the new pixels $z^l_{i,j}$ for layer l are determined as the following weighted sum,

$$z^l_{i,j} = \sum_m \sum_n \text{rot}_{180^\circ} \left\{ w^l \right\}_{m,n} * a^{l-1}_{i+m,j+n} + b^l_{i,j}. \qquad (1)$$

Here w^l is the kernel with weights and a^{l-1} is the output from the previous layer. b^l is the bias added to the result from the convolution ($*$) between w^l and a^{l-1}. The reason we rotate the kernel w^l by 180 degrees is to counteract the inherit rotation in the convolution operation.

As an example of the "scanning", say we have a 8×8 input image with a 2×2 kernel. This would reduce the size of the image when it comes out of the convolution layer to a 7×7. An illustration can be seen in figure 2.

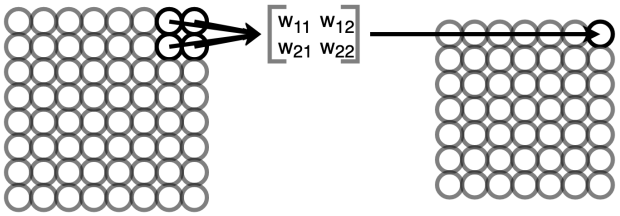

Figure 2: Illustration of a convolution. A 2×2 region in the input image is multiplied by weights which outputs a pixel.

In addition to choosing the size of the kernel, which affects the size of the output image, we also have the option to select how many pixels we move between each multiplication, essentially skipping some pixels. This is known as the stride. In the above example, we have assumed a stride $S = 1$. A stride of 1 therefore means the kernel moves one pixel at a time. This affects the size of the output from the convolution layer. Say we have a stride of 2 in the example above. In that case our output image would be reduced to a size of 4×4.

Another parameter affecting the size of the output from the convolution layer is called zero padding P. It consists of adding a boundary of zeros, increasing the

size of the input. The reason behind this is to make the boundary pixels contribute to the output in the same way as the the other pixels. If we use no zero padding, the boundary pixels will only contribute to the boundary pixels of the output. The other pixels, however, also affects several pixels not on the boundary. Again, using the example above with a zero padding of 1 (meaning we add a border with width 1 of zeros to the input image) the original image would have its dimension increased to 10×10, which gives the output image a size of 9×9 (given we use stride $S = 1$).

Equation 2-4 shows explicitly how the stride S, zero padding P and size of the kernel F (F=3 if the kernel has size 3×3) affects the volume of the new image after the convolution layer,

$$W_{i+1} = \frac{W_i - F + 2P}{S} + 1, \tag{2}$$

$$H_{i+1} = \frac{H_i - F + 2P}{S} + 1, \tag{3}$$

$$D_{i+1} = K \tag{4}$$

Here W, H and D are the width, height and depth of the images. The subscripts i and $i+1$ denote the images before and after a convolution layer. These expressions show that the width and height of the resulting image are computed equally by symmetry. The depth of the new image only depends on the number of kernels, K.

Another thing to note is that the input image may be either grayscale or in RGB color. In the latter case the input image is already of dimension $W \times H \times 3$ since there are 3 color values. This complicates things further by requiring that the kernels be 4-dimensional, where the additional fourth dimension deals with the colour.

The backpropagation algorithm in the convolutional layer in a CNN is different from a traditional FFNN, as the dimension of the weights have changed. The change in loss when a change in the weights is induced can be calculated through

$$\frac{\partial C}{\partial w^l_{m,n}} = \text{rot}_{180^\circ}\{\delta^l\} * a^{l-1}_{m,n}, \tag{5}$$

where C is the cost function, δ^l is the error and $a^{l-1}_{m,n}$ is the output value of pixel m, n in layer $l-1$. We notice in this expression that we also rotate δ^l by 180 degrees. Again, this is done to counteract the inherit rotation in the convolution operation. Minimizing the change in the cost function to 0 gives us the equation

$$\left(w^l_{m,n}\right)_{new} = \left(w^l_{m,n}\right)_{old} + \text{rot}_{180^\circ}\{\delta^l\} * a^{l-1}_{m,n}, \tag{6}$$

which tells us how to update the weights. Here we must be careful to update each kernel corresponding to each feature map independently. $\delta^l_{i,j}$ is updated to the previous layer through

$$\delta^l_{i,j} = \frac{\partial E}{\partial x^l_{i,j}} = \delta^{l+1}_{i,j} * \text{rot}_{180^\circ}\{w^{l+1}\}_{m,n} f'(x^l_{i,j}), \tag{7}$$

After the convolution has been performed, the output values are sent through an activation function. This is often seen as part of the convolution layer.

3.2 Pooling layer

It is often convenient to reduce the size of the image in between the convolution layers. Additional layers are added to the CNN that does exactly this, known as pooling layers. Similarly to the convolution case, for each $N \times N$ region in the original picture we get exactly one pixel. Two examples of how pooling can be done is mean pooling and max pooling, where in the mean pooling we take the mean pixel within the region we are considering and with max pooling we simply carry over the value of the max pixel within the region. The most normal approach in this method is to not include overlap, meaning that a 8×8 input image would scale down to a 4×4 image if we use a 2×2 kernel. Essentially the size of the kernel also gives the stride in the pooling layer, meaning that for an $n \times n$ kernel we have a stride of n to avoid overlapping pixels in this layer.

In the case of the pooling layer, the backpropagation algorithm works like the following; In the case of mean pooling, we calculate the average of the gradient of the output pixels and apply this to each region in the input image. If max pooling is used, we calculate the gradient of the output from each layer and only apply it to the pixels that was carried over. The pixels that were not used (the pixels that are not the maximum pixel) remain unchanged. In other words, if we look at a 2×2 region and only let the maximum pixel move on to the next layer, we only update this pixel during backpropagation and not the other 3.

In the end, the output is flattened and passed through a standard FFNN (like in project 2).

3.3 Decision tree

A decision tree typically consists of a root node, interior nodes and leaf nodes connected by branches. The main idea is to split the tree in two at each node and eventually end up with a prediction.

For a classification problem, a decision tree predicts that all observations belong to the most frequent class of training observations in the region to which it belongs. We are not just interested in the class prediction for a specific region of the interior node when we do an interpretation on the results of the classification tree. We are also interested in the class proportions where the training observations that fall within that given region. To increase the size of the decision tree, we use recursive binary splitting.

Recursive binary splitting lines up the values and tries different split points with a cost function. We then select the best cost by minimizing it. For binary splitting, we cannot use the MSE, but we can use a classification error rate. The classification error rate is the fraction of training observations in that region that do not belong to the most frequent class and can be written as

$$p_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k). \tag{8}$$

$I(y_i = k)$ gives 1 if the output equals the target and 0 otherwise. N_m is the number of observations in region R_m. Equation 8 is then used to compute the cost function. Here we use the entropy cost function,

$$s = -\sum_{k=1}^{K} p_{mk} \log p_{mk}. \tag{9}$$

We then minimize this expression to decide how to split the tree at each node.

One way to improve the accuracy of a decision tree is to include bootstrap aggregation (bagging). Here, we build several trees (dubbed estimators) that all try to predict the outcome of a given data set. Since each estimator (tree) has low bias and high variance, having more estimators reduces the variance as the output is averaged over all estimators.

4 Method

In this section we describe the implementations of the convolutional neural network and decision trees.

4.1 Convolutional neural network

In the forward step, we used Equation 1 to compute the outputs from the convolution layers. The convolution was performed with the convolution method from scipy.signal. This method rotates one of the input matrices by 180°. That means the pre-rotation of the kernel w and the error δ, described in the theory section, was counteracted and thus the operation worked as desired. The output was then sent through an activation function.

Next, the pooling layer was performed as described in the theory section. We implemented both max and mean pooling. In the case of max pooling, we saved the indices of the maximum values in each region for the backpropagation. We also created the option of choosing more convolution and pooling layers.

After the convolution and pooling layers, we used fully-connected layers like in the FFNN in project 2 [2]. In the output layer, the output was then compared to a target one-hot vector which contains only a 1 at the position of the true input category and zeros otherwise.

Here we used the identity matrix as activation function for both the hidden layers and the output layer. Weights and biases were initialized using the Xavier method discussed in project 2 [2]. This was done both in the convolution layers and the fully connected layers.

During backpropagation we have to make sure that each pixel is accounted for equally when updating the kernel. This is done by padding the image, adding zeros along the border. This way, the pixels along the edge will have equal contribution to the rest of the pixels when we update the kernel. In short, padding is a way to ensure equal contribution from each pixel in the image.

In addition, we must ensure that none of the pixels get skipped during backpropgataion, as we need to train the network on all the pixels available to us. This can only be done during feed-forward, by choosing a stride larger than 1. The way we avoid skipping pixels during backpropagation is to interweave the input image, adding zeros in between the pixels as shown in equation 10. This way, all the pixels will be present during backpropagation. To summarize, interweaving is done to make sure that no matter what stride we choose we still consider all pixels during backpropagation.

$$\text{interweave}\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = \begin{bmatrix} a & 0 & b & 0 \\ 0 & 0 & 0 & 0 \\ c & 0 & d & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{10}$$

The classification with decision trees and bagging were performed with the methods from the Scikit-Learn library [11].

5 Results & Discussion

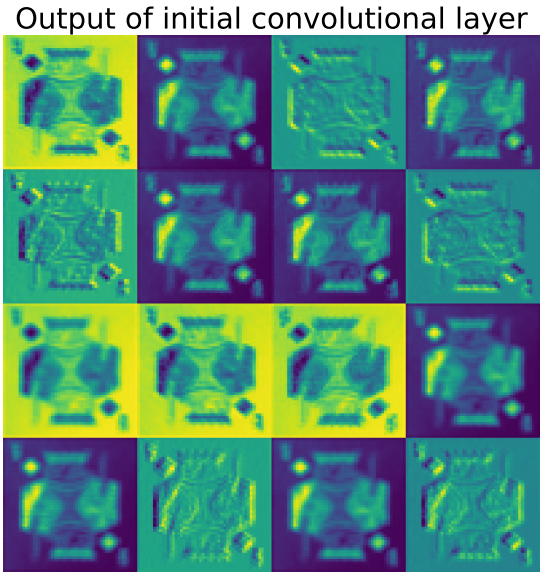

Figure 3: The output feature maps from the first convolutional layer

We initialize our network with a 50×50 input that runs through one convolution layer then a pooling layer 3 times. In the first convolution we create 16 feature maps (as seen in figure 3) and double it during the following convolution layers, meaning we end up with 64 feature maps (plotted in figure 4). In the dense layer we have 28 nodes, and in the output layer we have the option to choose 4 nodes, in which case the network tries to predict the suit of the card, or 52 nodes, in which case the network guesses the suit and number of the card. The activation function we found gave the best result was the identity matrix, meaning we simply carry over the values from one layer to the next. For cost function we chose the MSE and we used Adam adaptive method to improve the learning rate with an initial learning rate of 10^{-3}. We use a 3×3 kernel for each feature map in the convolutional layers and max pooling over a 2×2 region in the pooling layers. We train over 10 epochs.

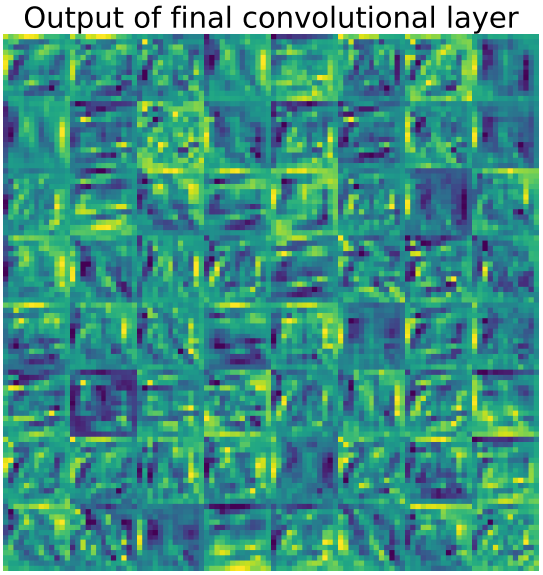

Figure 4: The feature maps from the last convolutional layer

We test our setup first against a binary problem where the CNN tries to predict if an input card is either an ace of spades and an eight of hearts. In this case, using a sigmoid activation function for the dense layers, we get an accuracy of 95% for the test data and 90% for the training set. We use 1040 training cards and 520 testing cards and test our networks against all types and suits of cards. In this case our CNN gets an accuracy score of 39% for the testing set and 88% for the training set. We get a much lower accuracy when using the whole data set, which makes sense because it is a much more complex problem. Still, 39% is a significant upgrade than if the CNN were to guess a card randomly, in which case the chance of guessing correct for one card would be about 2%. This accuracy can also be improved by using the full 256×256

image instead of a scaled down 50×50, by using all the 7624 available training cards and by training over more epochs, but this greatly increases the time the CNN takes to train. Using the full RGB image could also increase the accuracy of our network.

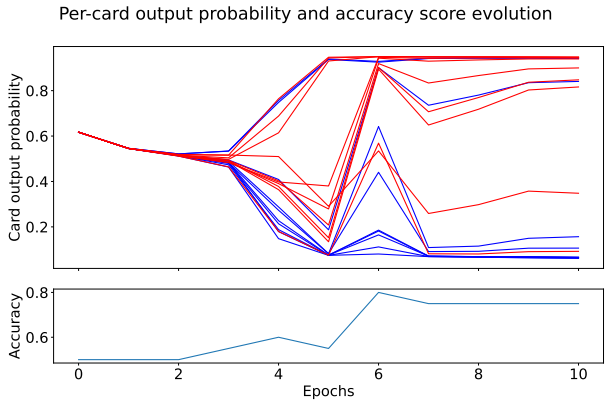

Figure 5: The top panel shows the probability output from a node, where each line represent a different card that is fed to the CNN. The bottom panel shows the overall accuracy of the CNN as a function of the number of epochs.

For decision trees with bagging we get an accuracy of 57% using the whole data set excluding the joker with images of 50×50 size in gray-scale. Having used the whole data set it makes sense that we achieve a higher accuracy score. We are confident that with enough training we can beat the trees with our network. This can be done by implementing the improvements mentioned at the end of the last paragraph.

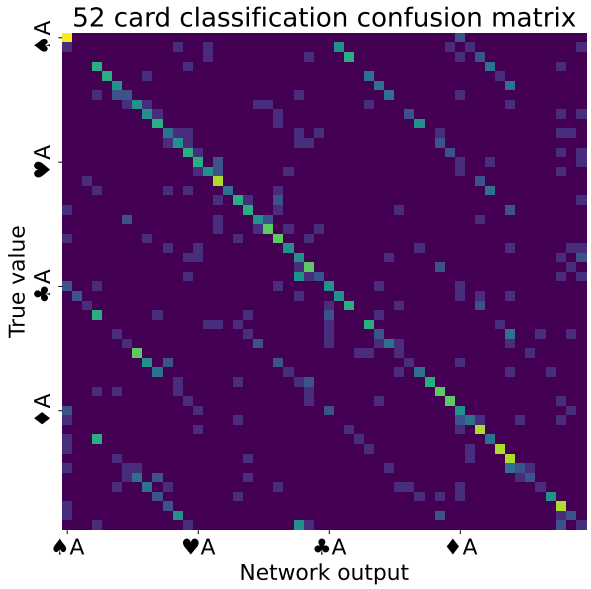

Figure 6: The confusion matrix for the CNN showing showing how many times the CNN guesses an output card given an input card. The diagonal shows how many correct guesses the network has made. The colours represent number of occurrences, with zero being represented by a dark blue and brighter green and yellow showing multiple occurrences.

We explore the accuracy of our CNN as a function of the number of epochs in the binary case, which is plotted in the bottom panel in figure 5. In this case we only have one output node, where a 1 represents an ace of spaces and a 0 represents an eight of hearts. This allows us to explore the probability output of that node for a given input card as a function of the number of epochs, which is plotted in the top panel in figure 5. We can see that initially, at the first epoch, the probability is 50%, meaning that the CNN essentially guesses which card we input. As we train it over more epochs we can see that the probability scatters. At the same time we see in the bottom panel that once the probability scatters the accuracy of our CNN starts to increase. At this stage the CNN is beginning to learn how to discern one card from the other. Once we hit epoch 10 our CNN has a probability of about 80%, showing that it can predict what card we input.

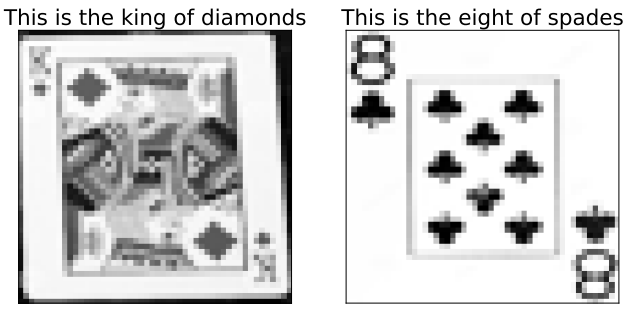

Figure 7: Example of both a correct and incorrect guess from the CNN.

We plot the confusion matrix of our CNN in figure 6. Say we input an ace of diamonds. The confusion matrix tells us how many times the CNN guesses (outputs) an ace of diamonds along with any other card. This means that the diagonal of this matrix tells us how many correct guesses the CNN has made for a given input card. We can see that there are some cards that the CNN gets confused. As an example we can see that the network often confuses the eights of clovers for the eight of spades, which makes sense since a pixelated clover can easily be mistaken for a spade. We see an example of this confusion in figure 7, where the network successfully guesses a king of diamonds but mistakes the eight of clovers for the eight of spades.

6 Conclusion

Using a data set of cards we built a CNN to predict what card we give the network. This was done by first convolving the input image to create 16 feature maps, then pooling before convoling and pooling 2 more times, doubling the amount of feature maps for each convolution until we end up with 64 maps. For the convolution layer we used a 3×3 kernel for each feature map and a 2×2 kernel for the pooling layer with max pooling. We use the identity matrix as activation function for all layers and the MSE as the cost function. Testing this setup against a binary case between the ace of spades or an eight of hearts, in which case we use a sigmoid activation function in the dense layers, we get an accuracy of 95% on the testing data. Testing it with 1040 training cards and 520 testing cards yields an accuracy of 39%.

To speed up the computation we scaled down the initial 256×256 card image to a 50×50 image. Choosing to test it agains a smaller data set, instead of the full set when trying to predict between all the cards in the data set excluding the joker also sped up the computation. Choosing to use gray-scale instead of the full RGB image further reduced the time.

We also use decision trees with bagging, in which case we got an accuracy of 57%. Here, we used the whole data set with each image scaled down to a 50×50 image excluding the joker. Increasing the size of each input image, using more cards to train the network, increasing the number of epochs to train over or using color could increase the accuracy of our network to match this.

Bias-Variance Tradeoff Analysis

Stian Aannerud, Marius Bjerke Børvind, Dennis $\hat{H}$ Fremstad and Andreas Wetzel

December 18, 2022

The aim with this report is to perform a bias-variance tradeoff analysis for the methods ordinary least squares (OLS), decision trees and feed-forward neural networks (FFNN) as function of complexity. We use polynomial order, maximum depth and number of hidden nodes as a measure of the complexity for the methods. For the neural network we only use one hidden layer. We consider a regression problem and use a continuous function as our data set. Bootstrap is used as resampling method. Our results showed that bias-variance tradeoff happens for a more complex model when the number of data points increases for OLS and decision trees. In addition, we found that it is possible to obtain both low bias and low variance with decision trees. For neural networks it was difficult to see at which model complexity the bias-variance tradeoff happens.

7 Introduction

In this bonus project we study the bias-variance tradeoff of ordinary least squares (OLS), decision trees and feed forward neural networks (FFNN) and compare them. We will here only consider regression problems and we use bootstrap as resampling method.

In the theory section, we first give a short recap of the bias-variance tradeoff also discussed in project 1. Then we go through the theory of decision trees. For a theory of OLS see project 1, and for FFNN see project 2. In the method section, we mention how we performed regression with the chosen methods and how the bias and variance were found. We then move on to present the results from the analysis and discuss the differences between the methods in addition to their pros and cons.

The code can be found at our GitHub page `https://github.com/frdennis/FYS-STK4155`.

8 Theory

8.1 Bias and variance

We start this section by recalling bias and variance which arises from a rewrite of the expression for the mean squared error (MSE). The MSE can in general be written

$$\text{MSE} = \mathbb{E}[(\mathbf{y} - \tilde{\mathbf{y}})^2], \tag{11}$$

where $\mathbf{y}$ is the target and $\tilde{\mathbf{y}}$ is our model. In project 1, we showed that this can be rewritten as

$$\mathbb{E}\left[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2\right] = (\text{Bias}[\tilde{\boldsymbol{y}}])^2 + \text{Var}[\tilde{\boldsymbol{y}}] + \sigma^2. \tag{12}$$

with

$$(\text{Bias}[\tilde{\boldsymbol{y}}])^2 = \mathbb{E}\left[(\boldsymbol{y} - \mathbb{E}\left[\tilde{\boldsymbol{y}}\right])^2\right], \tag{13}$$

$$\text{Var}[\tilde{\boldsymbol{y}}] = \mathbb{E}\left[(\tilde{\boldsymbol{y}} - \mathbb{E}\left[\tilde{\boldsymbol{y}}\right])^2\right]. \tag{14}$$

σ^2 is the error due to the variance of the data. $\mathbb{E}$ denotes expectation value.

Equation 12 can be used to easier show where the errors of the model comes from. The error due to the noise of the model is irreducible which means there is nothing we can do to avoid this. The error from the bias and variance, however, depends on the complexity of the model.

Variance is a type of error that occurs in a regression model due the model being too complex. This results in overfitting to the data, meaning that any noise present in the data also is included in the model. The error arises when the model is then tested against a test data set. Since the model is overfitted to the training data set, the test data set thus results in a larger error.

Bias, as opposed to variance, is an error which occurs because the model is too simple. The expectation value of the model will then be different from the target values. From equation 13, this leads to a high bias. We then get what is called underfitting, meaning the model is badly fitted to the data.

The goal in machine learning methods is to find a model which gives both low bias and low variance. A sufficient model is found at bias-variance trade-off, where the model is both complex enough to fit nicely to the data, but not too complex to where it also fits to the noise.

8.2 Decision trees

The theory of OLS and feed-forward neural networks have been covered in the previous projects. Therefore we only introduce the method of decision trees here. A decision tree usually consists of what is called a root node, interior nodes and leaf nodes which are connected by branches.

The main idea behind the decision tree is to split the data into parts and use the target values corresponding to the data in these parts to make predictions. It is common to only split the data into two parts at each step.

The splitting procedure for a regression problem with one-dimensional data, starts with choosing a cut-off value s. The data is then divided into two parts with data points $x_j < s$ and $x_j > s$. We then compute the sum of the MSE for both regions,

$$\text{MSE} = \sum_{i:x_i \in R_j} (y_i - \overline{y}_{R_1})^2 + \sum_{i:x_i \in R_2} (y_i - \overline{y}_{R_2})^2. \quad (15)$$

Here y_i is the target corresponding to the data point x_i. $\overline{y}_{R_1}$ and $\overline{y}_{R_2}$ are the average of the target values in the two regions. The predicted value for each region is set to the mean value of the target values corresponding to the data points in that region.

The data is split at all possible values of s, and for each s, the sum of the MSE for both regions is found. The s, and the two regions, from the split with the lowest MSE is then kept. Next, the same splitting procedure is performed for one of these regions. This can continue until a stopping criterion is met, like when there are only a particular number of data points left in the region with the lowest MSE. It is also possible to set a particular depth of the tree, meaning how many times we want to split the data. The latter option is used in this project since we want to study the bias-variance tradeoff as function of the model complexity. For decision trees we use the depth of the tree as the complexity.

For OLS and the neural network we use polynomial order and number of nodes in the first hidden layer as a measure of the complexity of the model.

9 Method

For all three algorithms, OLS, decision trees and feed-forward neural networks, we have used the methods from the Scikit-Learn library. The way these methods were used can be seen in our code on GitHub. For each method we also implemented the bootstrap resampling method. For a number of bootstraps, we draw values from the data set randomly with replacement.

The bias and variance for each algorithm were found by using equation 13 and the *var* function from numpy respectively. The expectation value $\mathbb{E}[\tilde{y}]$ were found by averaging over the predictions from all bootstrap samples.

Before computing the bias and variance for the neural network we found the learning rate and hyperparameter λ which gave the best results. The sigmoid was used as activation function, and adam was used as the adaptive learning rate algorithm. We only used one hidden layer and varied the number of nodes in this layer.

To test the methods we use a simple function with added noise, normally distributed with mean 0 and variance 0.1, over an interval $x \in [0.1, 10]$;

$$f(x) = \frac{sin(x)}{x} + cos(x) + \mathcal{N}(0, 0.1) \quad (16)$$

We include 50 bootstraps for all the methods and split the data set into 80% train data and 20% test data.

10 Results & Discussion

We explore the bias-variance trade-off for ordinary least squares as a function of polynomial degree and with a varying size of data set (10, 50 and 100 points). The result is plotted in figure 8. We can see that the bias-variance tradeoff occur around polynomial degree 2, 6 and 8 for 20, 50 and 100 data points respectively. We also see a trend in the MSE; we can see that

(in the middle and right panels) the MSE starts by decreasing before increasing after trade-off.

For OLS we get overfitting after bias-variance trade-off. This is indicated by the bias being low and the variance being high. We can see that for increasing number of data points we need a higher polynomial degree in order to get trade-off. This makes sense since the more data points we include the more complex our model must be in order to make an adequate fit given the noise present. When we only have 10 data points, for example, the underlying function is almost washed out by the noise. This means that when we try to fit a polynomial to it we quickly get overfitting.

One of the pros with using OLS is that it is generally fast and yields a low MSE for simpler data sets. However, if were to model the terrain data from Project 2, for example, we would find that OLS would not be a sufficient model unless we include many more polynomal degrees, in which case this method would begin to take significantly longer time to run. Another disadvantage is that the model easily overfits the data if we use high polynomial degrees which means we get a large variance. The variance can be reduced by bootstraping, but as we see from figure 8, this does not have a significant impact on the result.

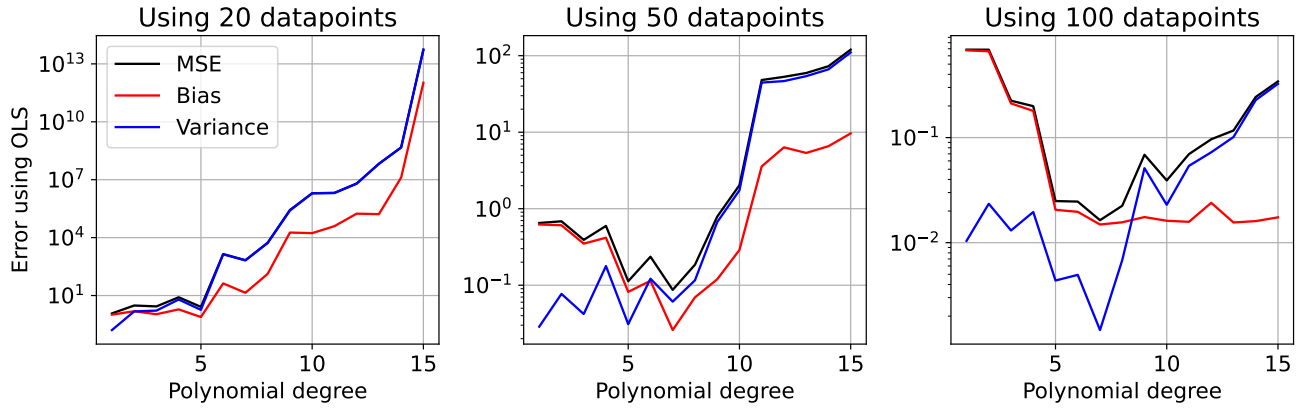

Figure 8: Error including MSE, bias and variance showing bias-variance tradeoff for OLS with varying size of data set.

We plot the MSE, variance and bias of a decision tree as a function of the max depth of the tree using different sized data sets. The plots are shown in figure 9. We can see that in this case the MSE starts off high, then decreases until it reaches somewhat of a constant value.

We can see that the decision tree follows a similar pattern to OLS where the size of the data set shifts the trade-off to the right. In addition we also see that the bias begins as the dominant source of error and after trade-off the variance takes over. This makes sense because if the tree is too shallow it will be unable to fit to the data sufficiently. However, if the tree is too deep, it is overcompensating for the training set which results in more error for the testing set and thus a larger variance.

One difference between the plots in figure 8 and 9 is that the variance for decision trees stays low even for larger maximum depths while it increases for higher order polynomials in the case of OLS. The reason for this could be that we use bootstrapping which reduces the variance. An advantage with decision trees is therefore that it is possible to obtain both low bias and low variance even for more complex models.

However, a disadvantage with decision trees is that they in general have high variance. To see why this is the case, we can consider the number of times the tree is split for a particular maximum depth. For each depth the tree, and the data, is split in two. Thus, for a maximum depth of 4, we have already split the data into $2^4 = 16$ regions. If the data then contains less than, or around, 20 data points, we might easily get overfitting which means we have high variance.

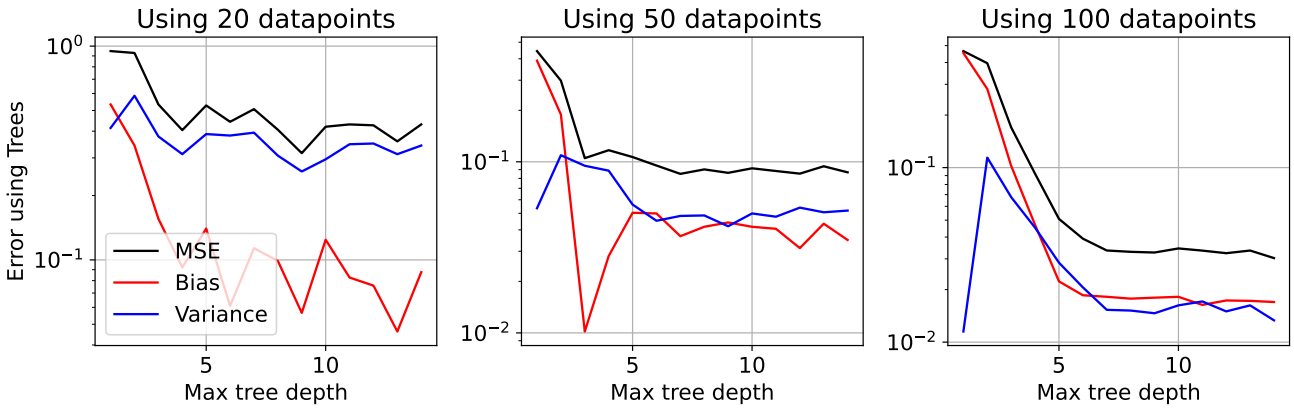

Figure 9: Error using decision trees as a function of max tree depth with varying size of data set.

From figure 10 we see that FFNN also yields similar results to what is mentioned above; as the data increases in points we require a more complex model to get trade-off. Using 20 data points we get trade-off between 1 and 2 hidden nodes, with 50 data points trade-off is found at around 6 hidden noes and for 100 data points we find trade-off at around 7 hidden nodes.

One of the disadvantages with using a FFNN is the amount of hyperparameters we have available. In order to get the best model with the available nodes and hidden layers it may sometimes be necessary to scan through these hyperparameters to see which combination gives us the best results. In addition to this, the MSE in the plots in figure 10 is unpredictable and does not follow a well defined trend like it does for OLS and decision trees. This can make it hard to find trade-off, meaning it is harder to say which complexity yields the best model.

One advantage with FFNN is the fact that we can model essentially any data sufficiently given that we find a nice combination of hyperparameters and number of hidden layers and nodes.

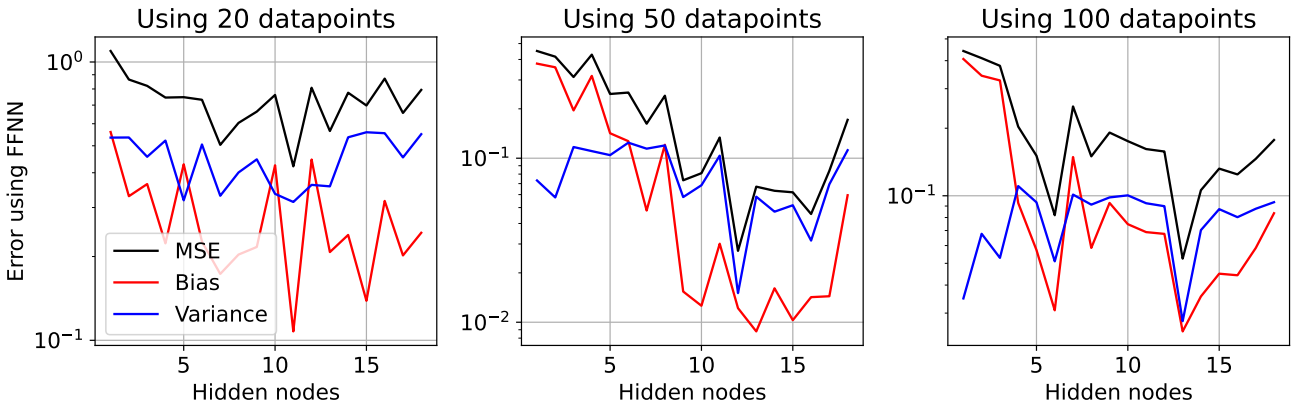

Figure 10: Error using FFNN as a function of hidden nodes with 1 hidden layer with varying size of data set.

11 Conclusion

We have studied the bias-variance trade-off as function of model complexity for three different methods, OLS, decision trees and FFNNs. We have found that each method exhibit some form of trade-off where the model goes from having a high bias and low variance to low bias and high variance. We find that increasing the size of our data set requires a more complex model to achieve trade-off.

We have found that OLS is a good approach when the data can be fitted by a simple polynomial, but fails for more complex data. Decision trees can be an advantageous model as it is possible to obtain both low bias and variance even for more complex models, however they generally yield higher variance. Neural networks can be tweaked to fit to any data set independent of its complexity, however it has the drawback of having unpredictable MSE, which can make it difficult to discover what complexity is best suited for a given data set.

References

[1] Aannerud, S., Børvind, M. B., Fremstad, D. $\hat{H}$., Wetzel, A. (2022). Deep learning methods: *Regression Analysis and Resampling Methods*: Oslo, Norway.

[2] Aannerud, S., Børvind, M. B., Fremstad, D. $\hat{H}$., Wetzel, A. (2022). Feed Forward Neural Network: *From Linear and Logistic Regression to Neural Networks*: Oslo, Norway.

[3] Hjorth-Jensen, M. (2022). Project 3: *Machine Learning*: https://compphysics.github.io/MachineLearning/doc/Projects/2022/Project3/pdf/Project3.pdf

[4] Hjorth-Jensen, M. (2021). Decision trees, ensamble methods and boosting: *Applied Data Analysis and Machine Learning Resampling Methods*: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html

[5] GERRY (2022). Kaggle dataset: *Cards Image Dataset-Classification*: https://www.kaggle.com/datasets/gpiosenka/cards-image-datasetclassification

[6] Li, F., Wu, J., Gao, R. (2022). CS231n: Convolutional Neural Networks (CNNs / ConvNets): *Deep Learning for Computer Vision*: https://cs231n.github.io/convolutional-networks/

[7] Solai, P. (2018). Chain Rule in a Convolutional Layer: *Convolutions and Backpropagations*: https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c

[8] Inada, H. (2019). Objectives: *Calculate CNN backprop with padding and stride set to 2*: https://hideyukiinada.github.io/cnn_backprop_strides2.html

[9] Jefkine. (2016). Convolution Neural Networks - CNNs: *Backpropagation In Convolutional Neural Networks*: https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/.

[10] Zhang, Z. (2016). Feed Forward: *Derivation of Backpropagation in Convolutional Neural Network (CNN)*: https://zzutk.github.io/docs/reports/2016.10%20-%20Derivation%20of%20Backpropagation%20in%20Convolutional%20Neural%20Network%20(CNN).pdf

[11] sklearn.ensemble: *A Bagging classifier*: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html