Handwritten Digit Recognition

Machine Learning

Author 1, Author 2, Author 3, Author 4

Faculty of Science and Engineering

University of Groningen

The Netherlands

February 2022

Abstract

In this paper, we present the results of three different classification models on a handwritten digit recognition task. We employ Random Forests and a Convolutional neural network (CNN) and compare the performance these models achieve to a Linear Regression model serving as a baseline. For the baseline and the Random Forest, hand-crafted features were utilized. It is found that the CNN performed better than the other models with an overall error rate of 1.4% on the test data, while the Random Forest achieved an error rate of 3.4%.

CONTENTS

I. Introduction

Since 1998, the MNIST database of handwritten digits has become one of the most popular benchmarking datasets in machine learning [1]. A similar but smaller dataset of handwritten digits is the CEDAR-CDROM database, which was first used in [2]. This dataset is used in the current project to compare the performance of three different machine learning methods (linear regression, random forests, and convolutional neural networks) in terms of classification error. Additionally, there is a consideration of which specific classification mistakes the models tend to make, and whether there is a notable difference between the models on this account.

II. Data

The data used for the chosen models consists of handwritten digits along with ground truth labels for each digit. There were 2000 examples of 16×15 gray-scale images of digits arranged in $n = 240$ dimensional vectors, examples of which can be seen in Figure 1. These images were encoded in integer steps of 0 to 6, where 0 was white and 6 was black. Of the 2000 samples in the dataset, there are 200 examples of each digit from 0 to 9. In order to be able to properly test the performance of the models, 1000 of the samples from the dataset were set aside for testing once a best-performing model was found, while the other 1000 samples were used for training and validation.

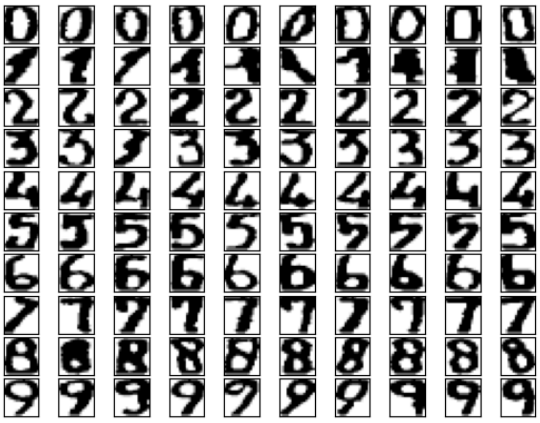

Fig. 1: Examples of the handwritten digits found in the dataset.

III. Methods

In the present paper, two different models were built and compared to a baseline in order to properly evaluate their performance. For the baseline, Linear Regression was used and compared to a Random Forest model as well as a Convolutional Neural Network. For both the Linear Regression model and the Random Forest model, handcrafted features were designed to reduce dimensionality and improve performance. Additionally, the data was augmented due to the dataset being relatively small. The implementation details will now be explained.

A. Data Augmentation

Due to the relatively small amount of data found in the training set, some data augmentations were considered. With the training set containing just 1000 samples the risk of overfitting is large, especially for powerful models such as Convolutional Neural Networks. One way to reduce the amount of overfitting with machine learning models is to augment the data to produce more datapoints [3]. Two methods were considered for this data, namely rotation and adding noise. However, the training data and the testing data used for the models is very clean and have little mismatch between them, which discouraged the idea of adding noise. In the case of rotations, using small random rotations (between -20 and 20 degrees) is a sensible idea for handwritten digit recognition [3]. Though with the small resolution of the images used, rotations of that magnitude sometimes caused some unwanted alterations to the certain samples like seen in Figure 2. Using smaller rotations on a range of -10 to 10 produced more consistent images with less artifacting as seen in Figure 3.

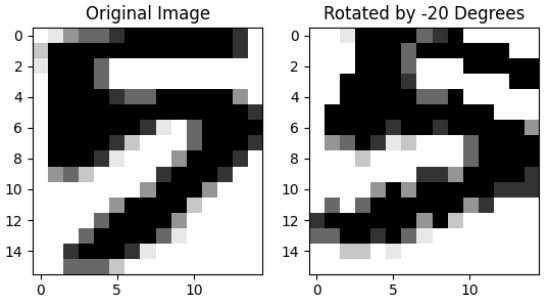

Fig. 2: Artifacting seen with large rotations.

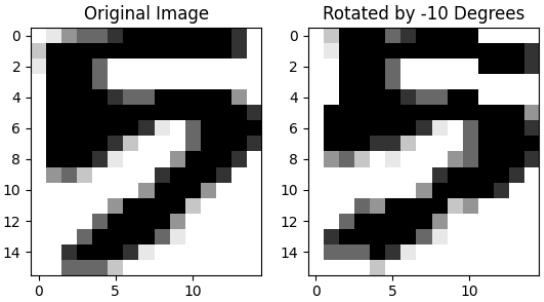

Fig. 3: Smaller rotations produce better augmentations of the data.

In order to confirm that the augmentations sufficiently help the model perform better, a small analysis was performed. A CNN model with two convolutional layers and two dense layers was used in 3 fold cross validation to make sure the performance would not decrease. Results of this analysis can be seen in Table I. It is clear that using the augmented dataset led to a slight increase in performance. Therefore the additional rotated images are used in training the final models.

B. Handcrafted features

The aim while developing handcrafted features was to find features with distinctive distributions between classes. The

Dataset	Fold 1	Fold 2	Fold 3	Avg. accuracy
Original	0.949	0.994	1.0	0.981
Rotated	0.993	0.999	1.0	0.997

TABLE I: Accuracies from K-fold cross validation runs to determine the effectiveness of image rotations.

implemented features are *Vertical ratio*, *Islands*, *Laplacian*, *Fourier*, *Regression on row averages*, *Mixture of Gaussians*, *Mean brightness*, and *Prototype matching*. Except for the last feature, the output distributions of the features per class are shown in Figure 4.

The training dataset consists of 1000 labelled data points $(\mathbf{x}_i, y_i)_{i=1,...,1000}$, with $\mathbf{x}_i \in \mathbb{R}^{240}$ and $y_i \in \mathbb{R}$. 18 features were designed to reduce the dimensionality of the input data. Combined, the features form a feature map $(f_1, ..., f_{18})' =: \mathbf{f} : \mathbb{R}^{240} \to \mathbb{R}^{18}$ of input vectors to feature vectors. Some of the features do not operate on the vector representations $\mathbf{x}_i$ of the images, but rather on the corresponding matrices $X_i \in \mathbb{R}^{16\times15}$ to which this representation can be reshaped.

1) Vertical ratio: The original idea for this feature was to measure the symmetry between the top half and the bottom half of the image. However, it was found that this feature was more informative when using different splits instead of using the middle row. This feature first thresholds all elements in the input vector to $\{0, 1\}$. Here this means that all pixels with a value of 6 become 0, while other pixels become 1. Then ratio between the number of non-zero pixels in rows 1 to k and the total number of non-zero pixels in the image is calculated, which is

$$f(X, k) = \frac{\sum_{i=1}^{k}\sum_{j=1}^{15} X_{i,j}}{\sum_{i=1}^{16}\sum_{j=1}^{15} X_{i,j}}. \quad (1)$$

Calculating the value distributions of this feature for $k \in \{1, ..., 15\}$ showed that the best results are obtained for $k = 3$, and $k = 8$. Therefore two features $f_1(X)$ and $f_2(X)$ were created, which respectively use $k = 3$ and $k = 8$. Figure 4-A shows that the digit 7 can be distinguished with hyperparameter $k = 3$, and Figure 4 B shows that the digits 4 and 6 can be distinguished with hyperparameter $k = 8$.

2) Islands: The islands feature counts the number of isolated white areas in an image. In general, the digits 1-5, and 7 have no islands, digits 0, 6, and 9 have one island, and the digit 8 has two islands. To prevent that islands are identified at the borders of the image, the image is padded with zero-valued pixels. Each pixel with value lower than a certain threshold is mapped to one, and the other elements are mapped to zero. The algorithm visits each pixel in the image. If a pixel has value one, a white area has been found. The eight adjacent pixels are then recursively visited. If an adjacent pixel within the image boundaries has value one, its value is changed such that it is not one. Else, the recursive function returns. Because all border pixels have value one, at least one isolated area will be found for each image. The final feature value for image $\mathbf{x}$ then is calculated as

$$f_3(\mathbf{x}) = \text{areas}(\mathbf{x}) - 1. \quad (2)$$

It was found that a threshold value of three led to the best performance. Figure 4-C shows the result of the feature on the training data.

3) Laplacian: The Laplacian feature responds to large differences in pixel values in an input image, which approximates the amount of edges a digit has. It is expected that digit 1 has less edges than the digit 8, for example, because it has less transitions from dark to light pixels, and vice versa. The function first pads white pixels around the image to properly detect all edges. Then the image is blurred with a (5×5) Gaussian kernel to reduce the response on irregularities of the digits. Finally, the image is transformed with a Laplacian kernel of (3×3) which highlights the sharpness of edges. The result of the feature is

$$f_4(X) = \frac{\sum_{i=1}^{16}\sum_{j=1}^{15} Laplace\left(Gaussian(X_{i,j})\right)}{\sum_{i=1}^{16}\sum_{j=1}^{15} X_{i,j}}. \quad (3)$$

Here, the numerator calculates the sum of all elements in the transformed matrix X, and the denominator calculates the sum of all elements in the matrix X. Kernel sizes of (3×3) up to and including (11×11) were considered in the search for the Gaussian and Laplacian kernels that resulted in the best performance. The results are shown in Figure 4-D. As expected, the digit 1 can be distinguished from the other digits.

4) Fourier: The Fourier feature is developed for similar reasons as for the Laplacian feature. It transforms the images to the frequency domain with a discrete Fourier transformation (DFT) to measure the number and sharpness of edges in the image.

As a simplified one-dimensional example to illustrate how DFT works, row 13 of the digit 6 shown in Figure 5-A is analysed. The pixel values for the row are shown in plot B, on which the DFT is as shown by the blue curve in plot C. The curve shows three peaks: at the frequencies 0 (DC component), 2 and 13 "oscillations per row". The sum of only these three sine waves (see plot D) already approximates the original curve in plot B quite well (see plots E and F).

The DFT on a 2D matrix is calculated with

$$F(k, l) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} f(i, j)e^{-i2\pi\left(\frac{ki}{N}+\frac{lj}{N}\right)}, \quad (4)$$

which is performed by the dft-function of OpenCV [4]. Although we lack the understanding to explain this equation, it transforms the image to its frequency domain, which consists of a real image F_1 and a complex image F_2. The feature calculates the average log of the magnitude of all pixels in the frequency domain with

$$f_5(X) = \frac{1}{240}\mathbf{1}'_{16}\log\left(1 + \sqrt{F_1^2 + F_2^2}\right)\mathbf{1}_{15}. \quad (5)$$

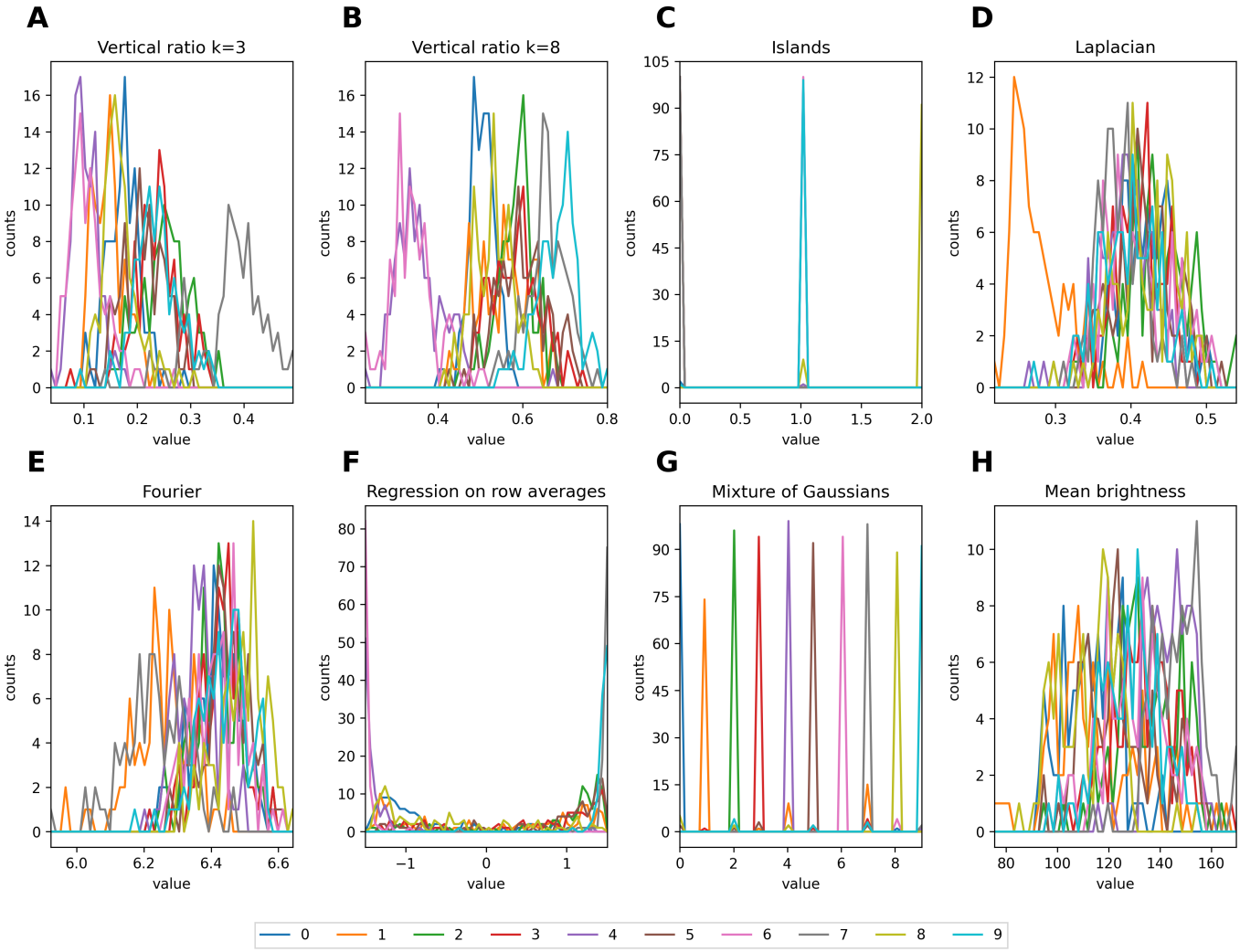

Fig. 4: Outputs of handcrafted features per class.

In this equation, the magnitude is simply calculated by the Euclidean norm of the real and complex image, and then scaled by the log functions. The sum of the result is calculated, which is divided by the total number of elements. The effect of this feature on the training data set is shown in Figure 4-E, which shows that the feature has some similarities with the Laplacian feature. This can be expected given that both features are in some form a way of detecting edges. Nevertheless, the Fourier feature distinguishes the digit 7 better than the Laplacian feature.

5) Regression on row averages: Each row in the image of a digit contains a different number of black pixels. For example, an image of the digit 9 contains more black pixels for the rows that make up the 'circle', than the rows that make up the 'tail'. This feature distinguishes digits by calculating the average black pixels value per row, and then calculates the weight of a linear regression curve on it. If the weight is zero, the digit is vertically symmetric, such as a perfect digit 0 or 8. If the weight is positive, there are more black pixels in the top rows compared to the bottom rows, which is the case for the digits 7 and 9. When the weight is negative, the bottom rows have more black pixels than the top rows, for the digits 4 and 6 for example. The arctangent of the weight is calculate to normalize the output. The calculation is done by

$$f_6(X) = \operatorname{atan}\left(\underset{a^*}{\operatorname{argmin}} \sum_{i=1}^{16} \left(a^* i - \frac{\sum_{i=1}^{16} X_{i,*}}{16}\right)^2\right), \quad (6)$$

in which $a \in \mathbb{R}$. $\sum_{i=1}^{16} X_{i,*}$ is the sum of all elements in row i, which is then divided by 16 to get the average value per row. The weight a for which the sum of squared errors is lowest is then selected. Finally, the arctangent of a is the result of the feature. As shown in Figure 4-F, the digits 6 and 4 can be distinguished from the digits 7 and 9 quite well.

6) Mixture of Gaussians: The probability distribution of the dataset was approximated using a mixture of Gaussians (MoG). The MoG was optimized using an expectation-

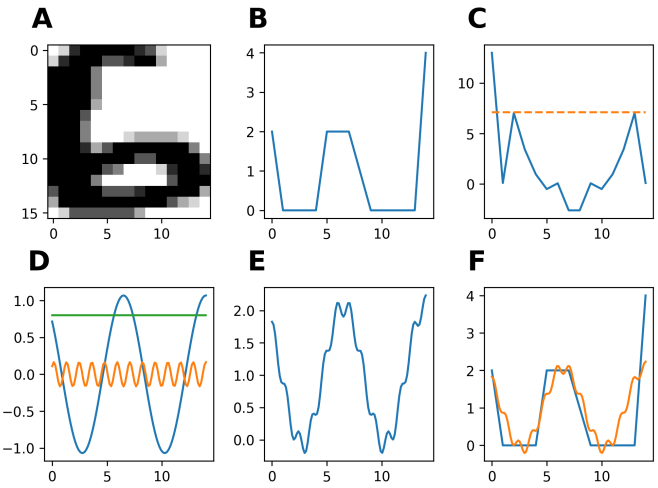

Fig. 5: A digit 6 (A) with its pixel values of row 13 (B), on which a DFT is calculated (C). The main components of the frequency domain (D) are summed (E), which approximated the original signal (F).

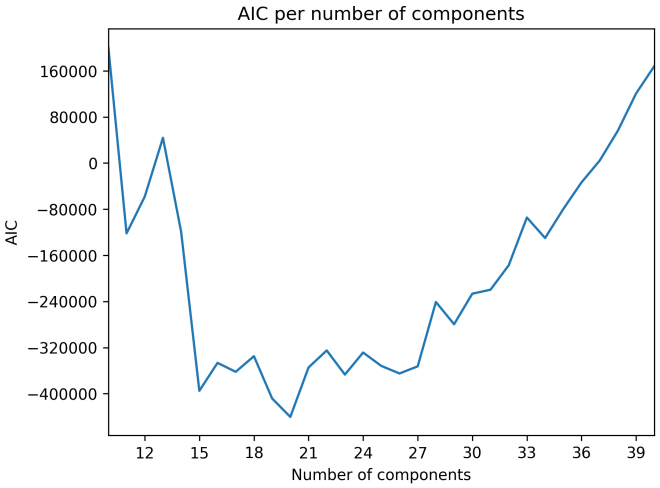

Fig. 6: AIC estimation of models with different mixture components.

maximization (EM) algorithm. The EM algorithm iterated until the change in likelihood was less than 0.001 per iteration. The optimal number of Gaussians was found by calculating the Akaike information criterion (AIC) for MoGs with between 10 and 40 Gaussians (see Figure 6). The MoG with 20 Gaussians was found to have the highest quality in relation to the other MoGs. For each image in the dataset, it was calculated from which Gaussian it was most likely sampled. Each Gaussian was then labelled by the most frequent class of the images that were most likely sampled from it. It was then calculated from which Gaussian i a new image $\mathbf{x}$ most likely originated, after which $\mathbf{x}$ received the same label as this Gaussian:

$$f_7(\mathbf{x}) = Label\left[\underset{1\leq i\leq 20\in\mathbb{N}}{\operatorname{argmax}}\left(P(Y=i|X=\mathbf{x})\right)\right]. \quad (7)$$

The predictions of the model on the training data are shown in 4-G. The graph shows that the digit 5 is estimated most accurately, and the digit 7 is estimated least accurately. The task of the decision forest is to take these different accuracy into account.

7) Mean brightness: The mean value of all pixels in an image is a rough indication about how much ink is representing a digit, and is calculated by

$$f_8(\mathbf{x}) = 1'_n\mathbf{x}/n, \quad (8)$$

where n is the number of dimensions of $\mathbf{x}$. As shown in 4-H, the distributions per class are not very distinctive from each other. However, they could occasionally support decision making.

8) Prototype matching: The prototype matching feature is a collection of ten features, $f_9(\mathbf{x}), ..., f_{18}(\mathbf{x})$, each one indicating the similarity between an image $\mathbf{x}$ and a prototype digit. Each prototype $\boldsymbol{\pi}_i$ is created by calculating the mean of all training images per class, which is

$$\boldsymbol{\pi}_i = \frac{1}{100}\sum_{j=1}^{100}\mathbf{x}_{(100i+j)}, 0\leq i\leq 9, i\in\mathbb{N}. \quad (9)$$

Each feature f_{9+i} then calculates the dot product between $\mathbf{x}$ and the prototype of digit class i as a measure of similarity:

$$f_{9+i}(\mathbf{x}) = \mathbf{x}\cdot\boldsymbol{\pi}_i, 0\leq i\leq 9, i\in\mathbb{N}. \quad (10)$$

Although the results of the features are not visualized, it is easy to see why these features would work well. Essentially, these features compare the images to the best possible estimate we have of what each number looks like. In other words, these feature are about as close as one can get to asking: 'How much does this number look like a zero?', 'How much does this number look like a one?', etc. If the number is not too atypical, it should always be the most similar to the prototype of the class it belongs to. Additionally, the dissimilarity to a number is also useful. If a number does not look like a zero at all, it is also unlikely to be a six or an eight.

C. Linear regression

To compare the errors of the models to a baseline, a linear regression model is calculated for on both the pixel values of the training set, and on the feature vector of the handcrafted features. Linear regression aims to find a linear map from the input to the output. This linear map is represented as a regression weight vector, and can be calculated as

$$(\mathbf{w},\mathbf{b}) = \underset{\mathbf{w}^*,\mathbf{b}^*}{\operatorname{argmin}}\sum_{i=1}^{N}(\mathbf{w}^*x_i+\mathbf{b}^*-y_i)^2, \quad (11)$$

where $\mathbf{w}$ is the linear map, $\mathbf{b}$ is a bias, $\mathbf{w}^*$ and $\mathbf{b}^*$ are proposed values for the regression weight vector and bias respectively, and x_i and y_i are the input vectors and output values, respectively. In this case, the input vectors are either

pixel values or feature vectors, and the output values are the classes the sample belongs to.

In both models, linear regression without regularization has been used because a quick and dirty 5-fold cross validation parameter search for Ridge regression pointed out that the accuracy improvement is negligible.

In order to calculate the weight matrix for linear regression of the handcrafted features, features with the categorical outputs where first transformed to dummy variables.

The linear regression is expected to work poorly on the pixel values. This is because it is highly unlikely that there is a linear relationship between the pixel values of an image and the class it belongs to. By extracting features from the image, which is nonlinear, the linear regression should improve.

D. Random Forest

The second classification algorithm that was tried on the dataset was a random forest [5]. A random forest is a collection of decision trees. A decision tree can classify a sample by querying a feature of that sample at every node of the tree. In this case, the decision tree could classify images using either the pixel values of the image, or by first extracting features from the image. An example of a feature that could be queried is the amount of islands in the image. If the image contains two islands, it can be classified as the number 8, since that is the only number from 0-9 containing two islands. Luckily, a decision tree does not have to be created manually. Instead, a simple algorithm exists [6]. The goal is to create a decision tree with "pure" leaf nodes. This means that, at the nodes at the bottom of the tree, only images of the same class end up. This would lead to perfect classification of the training data. To achieve this, the right features should be queried at the right time. To decide which features should be queried, and how they should be queried, the information gain is calculated. The information gain for a node is calculated as

$$\Delta i_{entropy}(v, Q) = i_{entropy}(v) - \sum_{l=1,...,k} \frac{n_l}{n} i_{entropy(v_l)}, \quad (12)$$

where v is a node, Q is the feature that has k attributes, n is the size of the sample set in the current node, and $n_l, ..., n_k$ are the sizes of the k child nodes $v_1, ..., v_k$. $i_{entropy}$ measures how pure a node v is as

$$i_{entropy}(v) = -\sum_{i=1,...,q} \frac{n_i}{n} log_2(\frac{n_i}{n}), \quad (13)$$

where q is the number of classes in the training data at node v, which are represented by subsets of sizes $n_1, ..., n_q$. An alternative way to measure how pure a node is, is using the Gini impurity. The Gini impurity for node v is calculated as

$$i_{Gini}(v) = 1 - \sum_{1 \leq i \leq q} (\frac{n_i}{n})^2. \quad (14)$$

To calculate the information gain using the Gini impurity, $i_{entropy}$ in Equation 12 can be replaced by i_{Gini}. By calculating the information gain for every possible feature that can be queried, the best feature to query can be determined. This is the query that has the highest information gain. The version of the decision tree algorithm that was explained here is deterministic and greedy. This means that using the same hyperparameters on the same feature vectors will always give the same decision tree, and the tree will likely be overfit.

The idea of a random forest is to combine the predictions of a number of different decision trees, and for example use a majority voting scheme to decide on the final classification of an input image. Using multiple version of the same tree would give exactly the same results as just using a single tree. This is why the decision trees in the random forest need to be different. To ensure that different decision trees are created, two things can be done. The first, is giving each decision tree a different set of bootstrapped data. This means that every tree uses only a part of the training data, and the differences in the data that each tree uses will then lead to different trees. This is also called bagging. The second method is to only use part of the features each time the information gain is calculated. This means that the feature with the maximum information gain is not always queried, since that feature might not be in the subset of considered features. Both of these methods are likely to make the single decision trees sub-optimal, but allow for more different decision trees, which leads to a better random forest. To create a random forest, scikit-learn's RandomForestClassifier was used. This is an implementation of Breimans random forests [5] in Python. It allows the user to change the number of trees in the forest, what impurity measure to use (entropy or Gini), stopping criteria of the trees, and the number of features that is used when determining a split. When creating the random forest, the performance using the default values for the hyperparameters was examined first, to verify that everything worked the way it should, and to get familiar with the process. For this, a 80-20 train-test split was used on the original training data, so without using the hand crafted features. After verifying that everything worked correctly, the handcrafted features were used. For the rest of this section, data will always refer to the feature vectors created using the hand crafted features, unless stated otherwise. First, the mean decrease in impurity of the features is used to measure their importance. The more a feature decreases the impurity of a node on average, the more important it is. The feature importance was measured using a 10-fold cross validation on the training data. This way, any features that were deemed unimportant could be removed. As can be seen in Table II, the most important feature is the Mixture of Gaussians, followed by the number of islands. The least important feature is the average pixel value. These results correspond to what was expected from the class distributions in Figure 4. Because there were no features with an exceptionally low importance, none of the features were removed.

After determining the features to be used, an exhaustive grid search using 10-fold cross validation on the training data was performed to find the best hyperparameters to use for the random forest. The grid search looked at the performance of using 10, 50, 100, 250, or 500 trees, using entropy or Gini impurity for calculating the information gain, different

TABLE II: The impurity-based importance values for the hand-crafted features using a random forest.

Feature	Mean importance value
Vertical Ratio x=3	0.05230371
Vertical Ratio x=8	0.06983256
Islands	0.1375078
Lapalacian	0.05548731
Fourier	0.0254011
Regression on row averages	0.06843745
Mixture of Gaussians	**0.22206059**
Average Pixel Value	0.01105021
Similarity to 0	0.03700622
Similarity to 1	0.02854172
Similarity to 2	0.03701364
Similarity to 3	0.04428705
Similarity to 4	0.03829522
Similarity to 5	0.03106872
Similarity to 6	0.05395421
Similarity to 7	0.03973339
Similarity to 8	0.02463343
Similarity to 9	0.02338566

ways of splitting the trees, and different stopping criteria for creating the trees. Through the grid search it was found that it was best to use a random forest consisting of 250 trees that uses the Gini impurity as the impurity measure. Additionally, every tree should have a maximum depth of 50, while not having a maximum allowed amount of leaf nodes. A node should only be considered a leaf node when it has at least 1 sample. Finally, at least 2 samples are necessary to split a node, and the amount of features that should be considered for a split should be equal to $\log_2(18) \approx 4$. As a final step in finding the optimal random forest, the randomness of the algorithm was manipulated. This was done to ensure that the model was reproducible. A 10-fold cross validation was done using different seeds for creating the random forest. The mean accuracies were recorded to determine which seed led to the best results. In Figure 7, a small part of the 125th tree of the random forest can be seen. Note that, because of the bagging, there are no longer 100 samples per class. The colors of the nodes indicate to what class the samples in that node belong to. The brightness of the color depends on the purity of the node, with purer nodes having brighter colors. In this figure, the purple color indicates that most of the samples in that node are sevens, while pink indicates the samples are nines. Despite being such a small part of a single decision tree, it does show how and why misclassifications happen in a random forest. Even when querying things like the similarity to a number, or the amount of islands in the figure, some unwanted samples 'slip through'. This can be seen at the root node of the figure, where the samples are split up by looking at the number of islands. The samples with 0 islands go to the left, while the samples with 1 or more islands go to the right. Contrary to what one would expect, three zeros and a nine end up on the left side, while two ones, a two, and a four end up on the right side. Another interesting thing to note is how the similarity to 4 is used to split up most of the nines from the rest. This happens at the right-most node in the second row.

To evaluate the performance of the random forest, it was used to classify the 1000 numbers in the testing set. The random forest is expected to perform quite well, since a 10-fold cross validation on the testing set gives a mean classification error of 3.2%

E. Convolutional Neural Network

A Convolutional Neural Network (CNN) is a type of neural network that is mainly applied to data that has a grid-like structure, such as images [7]. A CNN learns to extract features from images without human supervision, for example to be used in classification tasks. With more layers, the CNN can extract more complex features based on the earlier extracted simpler features. There are three main types of layers in a CNN: convolutional layers, pooling layers, and fully connected or dense layers. Features are extracted in the convolutional layers. In feature extraction, a filter (commonly with dimensions 2x2 or 3x3) slides over the input image, and at each point the convolution between this filter and the pixels that the filter covers is calculated. This convolution operation is essentially a sum of element wise products between the filter elements and the pixels covered by the filter. The output is a matrix containing these summed products, and is called a *feature map*. The operation is described by

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n), \tag{15}$$

where S is the feature map, I is a two-dimensional image, K is a two-dimensional filter or *kernel*, and m and n are the width and height of K.

Pooling layers downsize the feature maps. This reduces the number of weights to be trained, which allows for deeper networks resulting in more complex feature extraction. For example, a *max pooling* layer slides a pooling window over a feature map, and only retains the maximum value inside the window at each point. The neurons in a fully connected layer are connected to all of the neurons in the previous layer. The main purpose of the fully connected layers is to perform classification based on the calculated feature values.

The architecture of the implemented network is outlined in Figure 8. This architecture was decided on based on a 5-fold cross-validation search across various architectures. It was selected because it was able to overfit, but still showed adequate performance. In similar 5-fold cross-validation searches, the number of training epochs was set to 60, and the number of filters in the convolutional layers was set to 62. These settings were selected because the average validation error did not improve much for larger amounts of epochs and filters. The number of filters in the first convolutional layer was set to half of the number of filters in the other layers in all searches. The reasoning for this decision is that are not as many possible (simple) features to extract in the first layer in relation to the number of possible (more complex) features to extract in deeper layers. Adding more filters in the first layer would then mostly require more weights to be trained without much effect on performance. The ReLU activation function was applied to

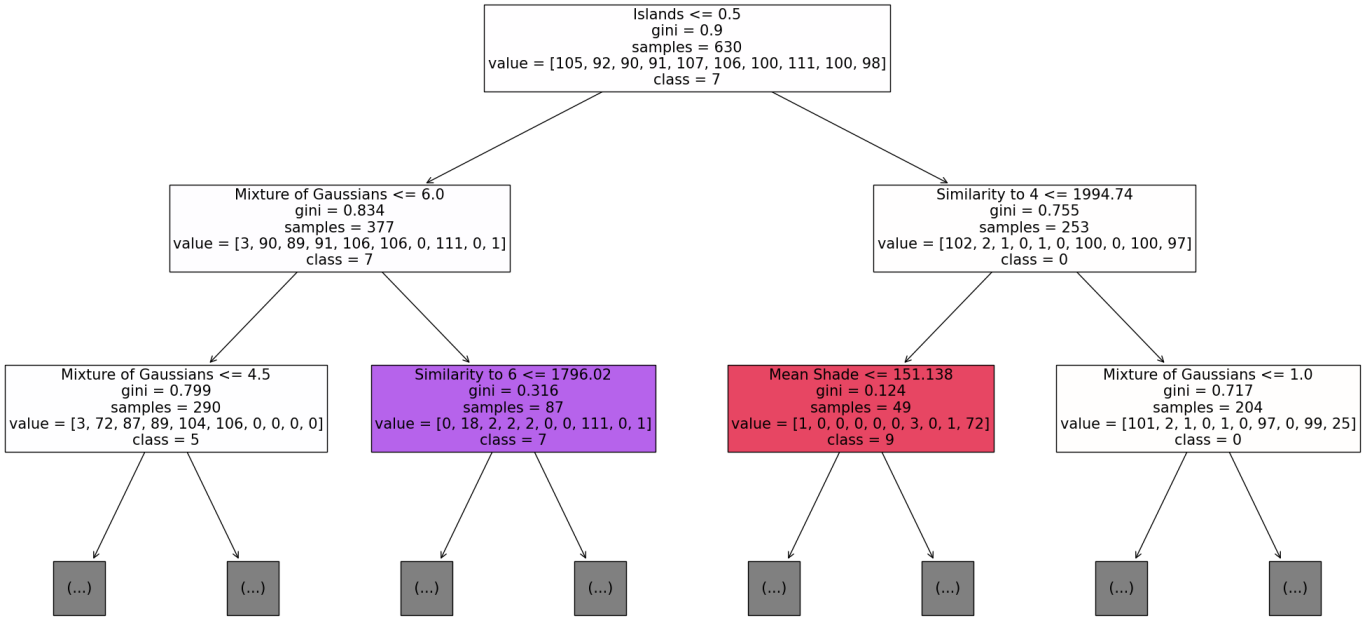

Fig. 7: A small part of tree 125 of the random forest.

each layer, except for the output layer to which no activation function was applied. The network was implemented using the Tensorflow and Keras libraries.

For optimization purposes, the Adam algorithm was selected [8]. In general in neural networks, the aim of optimization algorithms is to aid the (stochastic) gradient descent algorithm in converging to a minimum [9]. One class of such optimization algorithms is concerned with adapting the learning rate to the curvature of the error landscape. Adam is one of those algorithms. It combines ideas from optimizers such as gradient descent with momentum and Adadelta.

The standard gradient descent update rule is

$$\Delta \mathbf{w}^{(n+1)} = -\alpha \nabla F(\mathbf{w}^{(n)}), \tag{16}$$

where $\mathbf{w}$ is a vector of trainable weights, F is some loss function, α is a fixed learning rate, and n is the current step [10]. In (parts of) the error landscape where there is much more curvature in one dimension than in others, this process of updating the weights can lead to slow and oscillatory convergence, if that is even achieved at all. Adding a momentum term to gradient descent is a simple method of reducing the oscillations. By adding past weight updates to the current one, movements in the directions that the weight updates have in common are amplified, while movements in perpendicular directions are averaged out. The update rule then becomes

$$\Delta \mathbf{w}^{(n+1)} = -\alpha \nabla F(\mathbf{w}^{(n)}) + \gamma \Delta \mathbf{w}^{(n)}, \tag{17}$$

where the momentum term γ is generally set to some positive value below 1, such that weight updates that lie further in the past have less influence.

Adadelta is an optimization method that uses a different learning rate for each parameter [11]. The learning rate for each weight is scaled by the exponentially decaying sum of squared gradients with regard to the weights. In this way, the learning rates of weights for which the gradient is large are small, and the learning rates of weights for which the gradient is small are large. For each step, the weights are then updated as follows:

$$\mathbf{g}_{n+1} = \nabla F(\mathbf{w}^{(n)}) \tag{18}$$

$$E[\mathbf{g}^2]_{n+1} = \rho E[\mathbf{g}^2]_n + (1-\rho)\mathbf{g}_n^2 \tag{19}$$

$$\Delta \mathbf{w}^{(n+1)} = -\frac{\alpha}{\sqrt{E[\mathbf{g}^2]_{n+1} + \epsilon}} \mathbf{g}_{n+1}, \tag{20}$$

where $E[\mathbf{g}^2]_{n+1}$ is a running average of past and current sqaured gradients, ρ is a fixed decay factor, α is a fixed global learning rate, and ϵ is a small positive value to avoid division by zero.

Adam utilises both a decaying average of past gradients and a decaying average of past squared gradients, such as momentum and Adadelta respectively use. After calculating these averages, a correction is applied because they tend to be biased towards their initial values. Then the update is as follows:

$$\mathbf{m}_{n+1} = \beta_1 \mathbf{m}_n + (1-\beta_1)\mathbf{g}_{n+1} \tag{21}$$

$$\mathbf{v}_{n+1} = \beta_2 \mathbf{v}_n + (1-\beta_2)\mathbf{g}_{n+1}^2 \tag{22}$$

$$\hat{\mathbf{m}}_{n+1} = \frac{\mathbf{m}_{n+1}}{1-\beta_1^{n+1}} \tag{23}$$

$$\hat{\mathbf{v}}_{n+1} = \frac{\mathbf{v}_{n+1}}{1-\beta_2^{n+1}} \tag{24}$$

$$\Delta \mathbf{w}^{(n+1)} = -\frac{\alpha}{\sqrt{\hat{\mathbf{v}}_{n+1}} + \epsilon} \hat{\mathbf{m}}_{n+1}, \tag{25}$$

where β_1 and β_2 are fixed decay factors.

Sparse categorical cross-entropy was implemented as the loss function [12]. In this loss function, it is assumed that the

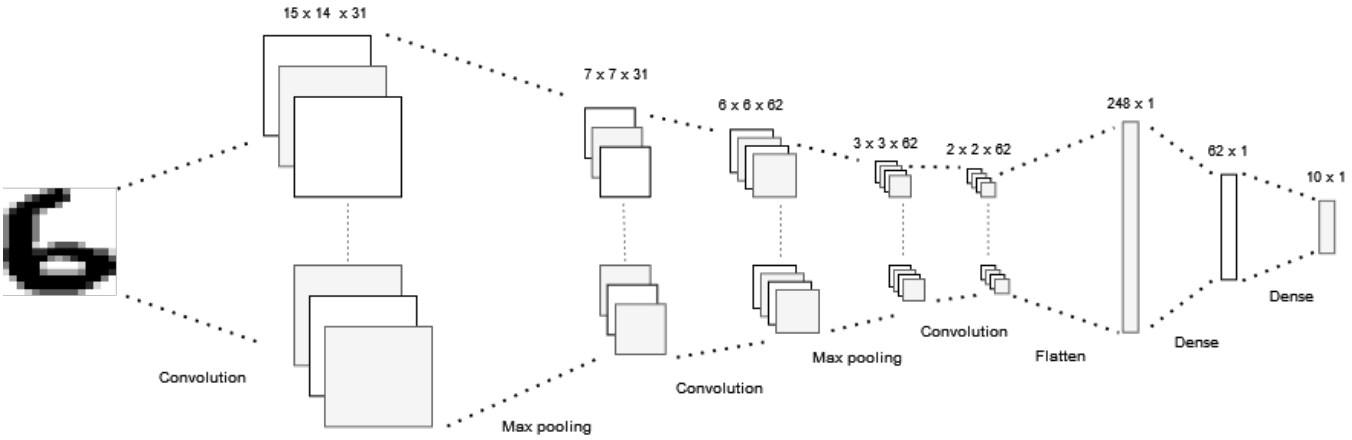

Fig. 8: The architecture of the implemented CNN.

softmax function has been applied to the model output. This transforms the output to a probability vector $\mathbf{p}$. Given these predicted output probabilities and the target output values $\mathbf{t}$, the categorical cross-entropy is then calculated as

$$CCE(\mathbf{p}, \mathbf{t}) = -\sum_{c=1}^{C} log(p_c)t_c, \tag{26}$$

where C is the number of classes. Because only one t_c is equal to 1 instead of 0 (namely, for the actual class of the input data), this simplifies to

$$CCE(\mathbf{p}, \mathbf{t}) = -log(p_{c^*})t_{c^*}, \tag{27}$$

where c^* is the true class of the input data. Note that "sparse" refers to the encoding of the target outputs as integer values in Keras, while a one-hot encoding would be the standard.

L_2-norm weight regularization was applied to each layer, except for the output layer. This consists of adding a penalty to the loss function that is proportional to the L_2-norm on the weights of a model , with the underlying idea that this leads to "simpler" models that are less prone to overfitting [7], [13]. Initial searches revealed that a combination with L_1-norm activation regularization on only the output layer tended to lead to an additional error decrease. This method not only tends to lead to smaller activations, but also to more activations being exactly zero (sparse). Similar to large weights indicating an overfitted model, large activations are also a sign of a model that will not generalize well [14]. The desire to have sparse activations is partly inspired by the small amount of neurons in the brain being activated at any moment, which is more energy efficient [15]. A 5-fold cross-validation search was performed to find the regularization factors that lead to the lowest validation error. To save on computational costs, the same factor was used on each layer's weight regularization. This resulted in a weight regularization factor of 10^{-5}, and an activation regularization factor of 0.005.

TABLE III

Method	Classification error
Linear regression on pixel values	78.8%
Linear regression on handcrafted features	30.2%
Random forest	3.4%
Convolutional neural network	1.4%

IV. Results

The performance of each implemented method on the test data can be observed in Table III. The CNN performs best, with a classification error of 1.4%, and is followed by the random forest at 3.4%. Figure 9 and Figure 10 contain the confusion matrices for the random forest model and the CNN, respectively.

V. Discussion

In this section, the results for the linear regression, random forest, and CNN will first be discussed individually, and then compared to each other. The comparison mainly involves the random forest and the CNN, since these methods were deemed the most interesting, and since the linear regression model was meant more as a baseline.

A. Linear Regression

As expected, linear regression performed poorly when using the pixel values. When using the feature vectors, performance was a lot better, but not nearly on par with the other methods. Since linear regression was meant more as a baseline, the poor performance was expected and not unwelcome. Spending more time on the linear regression and adding ridge regression might have improved its performance, but it would likely not be comparable with the random forest, and especially not with the CNN.

B. Random Forest

As can be seen in Figure 9, the random forest was best at classifying sevens and eights, while being worst at classifying threes and sixes. It is interesting to note that the mean

importance value for the similarity to 3 and similarity to 6 were higher than the other similarity features. This might indicate that those features are important to separate the threes and sixes from other numbers, because the other features might not do this well enough. This dependency on a single feature would then lead to the lower accuracy for those numbers. When examining the misclassifcations, some are easy to explain, such as when a number does not have the expected amount of islands, such as an eight having only one island, and nines and sixes with zero islands. Other seemingly typical examples of numbers are misclassified for no apparent reason. It also seems like the random forest has issues with rotated numbers. Examples of numbers that were classified and seem to be slightly rotated can be seen in Figure 11. This rotation can lead to atypical results for the vertical ratio features. Most of the features that were used are sensitive to rotation. Adding more rotation-invariant features, such as the islands feature, might help the random forest classify these numbers correctly.

C. Convolutional neural network

The CNN achieved at least a nearly perfect accuracy for each class, but it did not perform equally well on each class. It achieved a perfect accuracy for the twos and fours, while it made the most misclassifications for the sixes.

D. Comparison

While for both the random forest and the CNN the sixes were among the classes for which the accuracy was lowest, there is no overlap in the classes for which they achieved the highest accuracies. It is not clear why this difference arises.

Both the random forest and the CNN only misclassify the sixes as fours or eights, with a larger proportion of eights for the random forest. Additionally, the only time the random forest misclassifies an eight it is misclassified as a six, while for the CNN it is misclassified as a four. Notably, the fours are never misclassified as sixes or eights. Thus there seems to be some relation between fours, sixes, and eights. Even though the random forest was worse at classifying the numbers than the CNN, the random forest is a lot less complex. This makes it easier to understand how the random forest works, and can help with improving the model further.

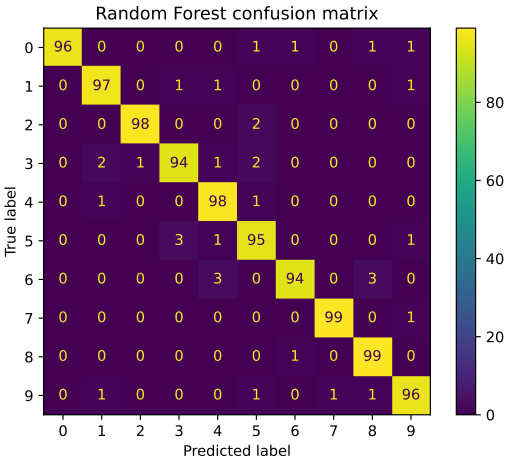

Fig. 9: Confusion matrix for the random forest.

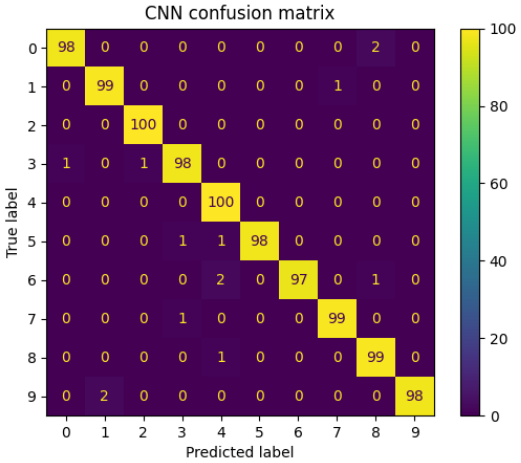

Fig. 10: Confusion matrix for the Convolutional Neural Network

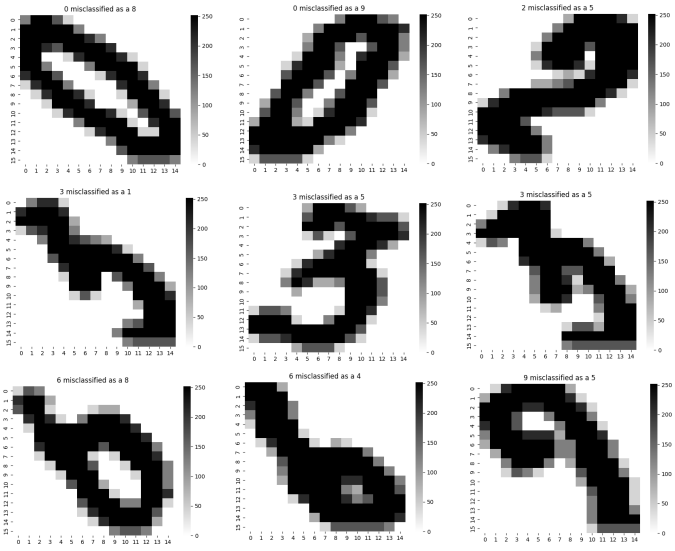

Fig. 11: Nine samples that were misclassified by the random forest.

E. Further work

As some digits in the dataset already appeared upside down before augmentation, it would be interesting to explore image rotations beyond 10 degrees. Figure 12 for example shows an obviously upside-down three being misclassified as a two by the CNN, which signifies that the CNN is not robust against large rotations. While using the current method of rotating images, increasing the range of possible rotations would cause unwanted artifacts in the augmented data. And with the low resolution of the images, some rotations could potentially become cropped. However, this could possibly be solved by

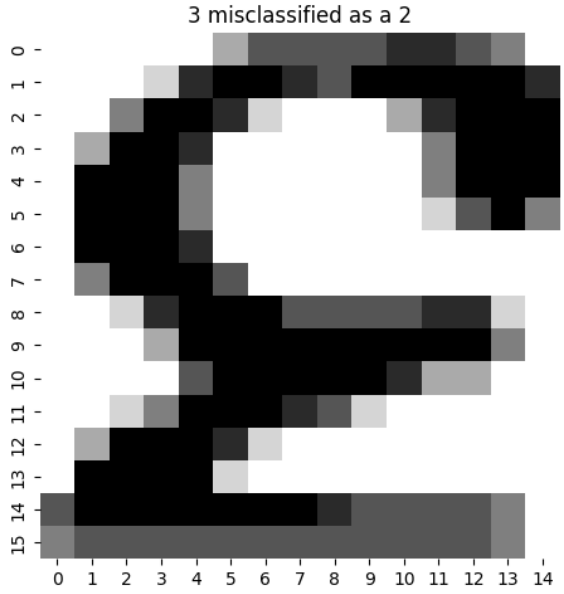

Fig. 12: A three rotated by about 180 degrees misclassified as a two by the CNN.

rotating images only after padding them. Doing this could help the model recognize flipped digits like highlighted above and reduce overall error.

For the random forest, it would be interesting to try more handcrafted features. Especially adding more rotation-invariant features could help eliminating most of the errors the random forest currently makes. Additionally, adding more features could prevent the random forest from being over-reliant on some features. This mainly happened with the islands feature, where atypical samples having the 'wrong' amount of islands would be misclassified. By adding more features, such samples could be classified correctly. To achieve even better classification, the models could also be combined. Since the current models give poor classifications for the same numbers, this might not help much, but the models could be tweaked to 'fill in the gaps' of the other models. Adding more models to compare would also be interesting. Especially trying different neural networks to compare to the CNN.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Bearning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] J. Kittler, M. Hatef, R. P. Duin, and J. Matas, "On combining classifiers," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 3, pp. 226–239, 1998.

[3] C. Shorten and T. M. Khoshgoftaar, "A survey on Image Data Augmentation for Deep Learning," *Journal of Big Data*, vol. 6, no. 1, p. 60, Jul. 2019, ISSN: 2196-1115. DOI: 10.1186/s40537-019-0197-0. [Online]. Available: https://doi.org/10.1186/s40537-019-0197-0 (visited on 02/03/2022).

[4] OpenCV. "Discrete Fourier Transform." (n.d.), [Online]. Available: https://docs.opencv.org/4.x/d8/d01/tutorial_discrete_fourier_transform.html (visited on 02/06/2022).

[5] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001, ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. [Online]. Available: http://dx.doi.org/10.1023/A%3A1010933404324.

[6] D. G. Stork, R. O. Duda, P. E. Hart, and D Stork, "Pattern Classification," *A Wiley-Interscience Publication*, 2001.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[8] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[9] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[11] M. D. Zeiler, "Adadelta: An adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[12] J. Brownlee. "A gentle introduction to cross-entropy for machine learning." (2019), [Online]. Available: https://machinelearningmastery.com/cross-entropy-for-machine-learning/ (visited on 02/05/2022).

[13] J. Brownlee. "Use weight regularization to reduce overfitting of deep learning models." (2018), [Online]. Available: https://machinelearningmastery.com/weight-regularization-to-reduce-overfitting-of-deep-learning-models/ (visited on 02/05/2022).

[14] J. Brownlee. "A Gentle Introduction to Activation Regularization in Deep Learning." (2018), [Online]. Available: https://machinelearningmastery.com/activation-regularization-for-reducing-generalization-error-in-deep-learning-neural-networks/ (visited on 02/05/2022).

[15] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2011, pp. 315–323.