DS807: Applied Machine Learning

Exam

Helle Juul Hansen, hehan20

Kristian Peter Lorenzen, krlor17

Rune Heidtmann, ruhei08

January 31st 2022

Introduction

Authors: hehan20, ruhei08, krlor17

DIDA [1][2] is a dataset of images of handwritten digits collected from Swedish historical documents all of which have been written between 1800 and 1840. In this project we are going to look at the part of the dataset that consists of 12.000 labelled digit string images. From Figure 1 we see that most of the digit strings consist of 4 digits. In Figure 2 we take a closer look at the digit strings of length four. We see that most, but not all, of these correspond to years in the range from 1800 to 1840.

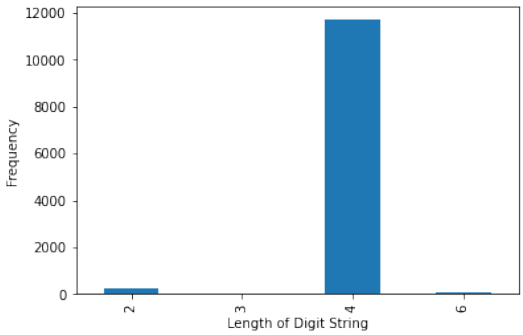


Figure 1: The distribution of the lengths of the digit strings present in the dataset.

The aim of the project is to perform image classification of these images. Instead of building a single classifier to predict the entire year-string, we use a CC-D-Y modelling strategy. That is, we build three separate classifiers: one for predicting the century, one for predicting the decade and one for predicting the year (see Figure 3).
For the classifier *modelCC* to predict the century we make the labels binary: $(18, \neg 18) := (0, 1)$ so that we simply predict whether it is the 19^{th} century or not. This makes these labels extremely unbalanced as seen in Figure 5a.

Helle Juul Hansen, hehan20
Kristian Peter Lorenzen, krlor17
Rune Heidtmann, ruhei08

DS807

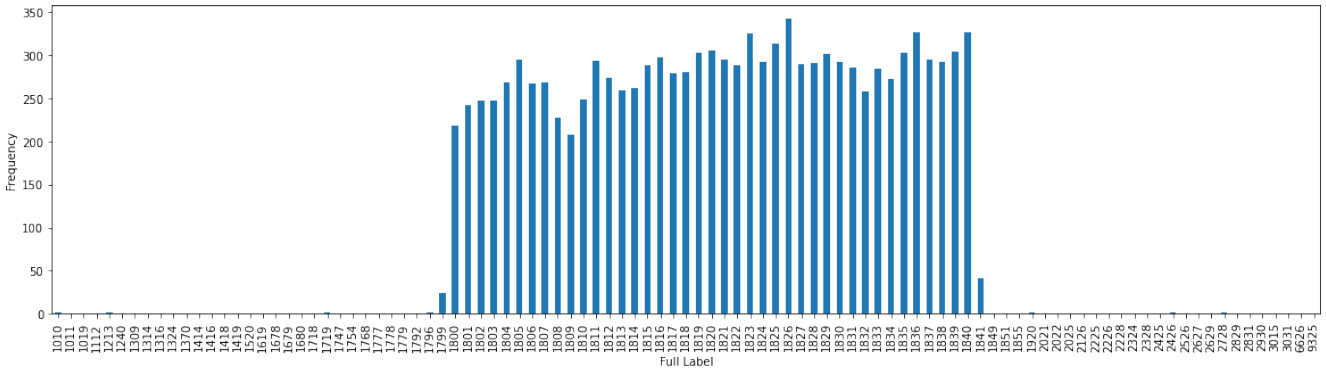


Figure 2: The frequencies of the four digit numbers present in the dataset.

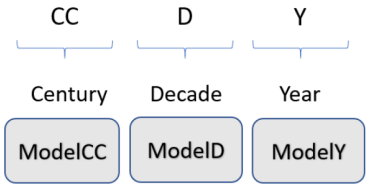


Figure 3: The CC-D-Y modelling strategy. Each four digit year string is divided into the century, *CC*, the decade, *D* and the year, *Y*. We build three distinct models for predicting each part of the year-string.

For the classifier *modelD* to predict the decade we create 5 classes: 0, 1, 2, 3, 4 and 10. The first five classes correspond to the digit shown in the image. The last class is a "wildcard" class. Since we mainly have years in the range 1800 to 1840 in the dataset the decade digit rarely takes the values 5, 6, 7, 8 and 9 (see Figure 4). In fact the digit 8 is only present one time in the entire dataset! Therefore, the aim of the decade classifier is only to recognize the digits from 0 to 4. The rest of the images are put in the wildcard class. Further, all images showing a digit string that is not of length four is also labelled as "wildcard". If it differs too much where in the image the decade digit occurs, the classifier will have difficulty learning the right patterns. In Figure 5b the distribution of the D labels are shown.

For the Y labels for the *modelY* classifier to predict the last digit we create 11 classes: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10. Again class 10 is a "wildcard" class. Every image showing a digit string that is not of length four are labelled as class 10. Again, this is to be certain that the classifier only needs to look in roughly the same area of the images for the Y digits. The Y labels are almost evenly distributed as can be seen in Figure 5c.

The code for creating the labels for the three classifiers can be seen in `Introduction_Create_Labels.ipynb`.

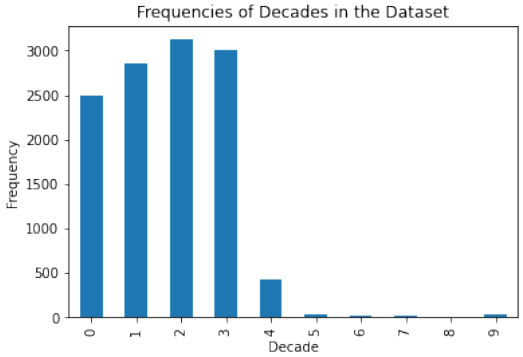
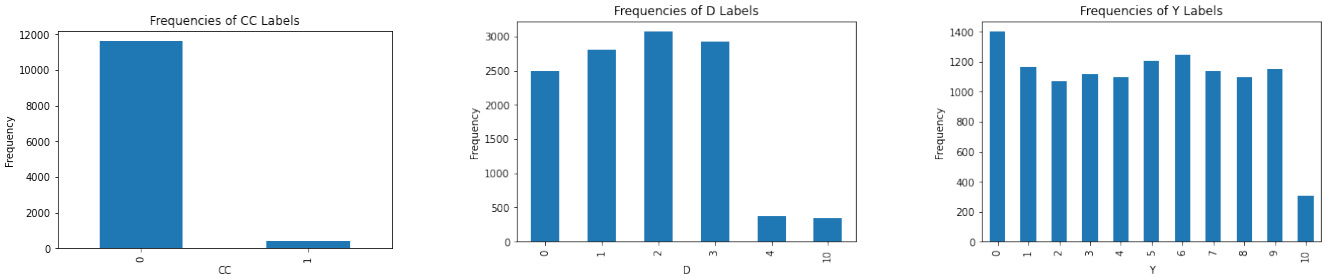


Figure 4: The frequencies of the digit in the decade position in the string.



(a) CC labels **(b)** D labels **(c)** Y labels

Figure 5: The distribution of the labels for the three classifiers.

1 Question 1: A Non-Deep Learning Approach

Authors: hehan20

As a first attempt at building three classifiers for image classification according to the CC-D-Y modelling strategy we will look at possible non-deep learning solutions.

1.1 Possible Methods

The methods we will consider are support vector machines, random forests and tree boosting. Knowing a priori which of these models will perform best is not possible. As will be explained shortly, we work in a 2,500-dimensional feature space with each pixel being its own dimension. The shape of the decision boundary most suited to separate the classes can only be determined by experiment.

Support Vector Machines

The idea of support vector machines is to separate two classes by a hyperplane with as large a margin as possible [3][4]. The width of the margin is given by the distance of the support vectors to the hyperplane, where support vectors are the name given to the data points closest to the hyperplane. Classification of a new data point is then

based on what side of the hyperplane the point lies.

For multi class classification with k classes SVC always works by training $k \cdot (k-1)/2$ classifiers following a one-vs-one strategy. The final classification is then based on a majority vote. Using a one-vs-rest strategy is not implemented in Sci-Kit Learn following the reasoning in [5] that it cannot detect correlations between classes, since it breaks the problem into *independent* binary problems.

Transforming the data from the original representation to a new feature space can in some cases help to make the data linearly separable, e.g., going from Cartesian to polar coordinates. Instead of doing this transformation as a separate step, this is done implicitly by the use of kernel functions ("the kernel trick"). The most used kernel functions are:

- Linear: $< x, x' >$ (the normal dot-product).
- Radial Basis Function (rbf): $exp(-\gamma||x - x'||^2)$ where γ is a hyperparameter.
- Polynomial (poly): $(\gamma < x, x' > +r)^d$ where d is the degree and γ and r are hyperparameters.

When using different kernels the decision boundary created is no longer linear. For polynomials using a high degree d the decision boundary is very flexible and "wobbly", indicating that this would easily overfit. The rbf kernel finds areas in the original feature space of the same class creating a decision boundary tending towards the circular.

Only rarely is the data fully separable and therefore support vector machines uses soft margins. As a hyperparameter you choose a value C, that works as a "slack budget". Each datapoint is then allowed to be a distince ϵ_i within the margin, but the overall sum of these distances is at most allowed to be C.

$$\sum_{i=1}^{n} \epsilon_i \leq C, \qquad where\ \epsilon_i \geq 0$$

It is important to note that the implementation of SVC in SciKit-Learn uses different notation than our textbook. Confusingly, they also have a parameter C that controls how soft the margins are, but with opposite meaning: a low value of C means wide margins, and a high value of C means narrow margins.

Random Forests

A single decision tree has high variance and easily overfits the dataset [3][4]. In a Random Forest we train multiple trees instead of just one and classify new instances based on the majority vote of the trees. This method for combining multiple learners is called bagging. It is important to ensure the trees are not identical, therefore the bootstrap method is used to create a new dataset for each tree. That is, you draw from the training data with replacement, meaning an instance can be drawn more than once - or not at all. Random Forests reduces the variance compared to a single decision tree and gives a more robust classifier that does not overfit. Like a single decision tree, a random forest creates a decision boundary consisting of axis-parallel piece-wise lines.

Tree Boosting

The idea of boosting is similar to that of a Random Forrest, but instead of training each new tree on the input data, they are trained on the residuals, that is, the difference between the prediction of the previous tree and the correct labels. This means the trees are grown sequentially [3]. The model $\hat{f}(x)$ is updated by adding a shrunken version of the new learned tree for each iteration $\hat{f}^b(x_i)$. The learning rate λ determines the shrinkage:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \cdot \hat{f}^b(x_i)$$

By setting a low learning rate we reduce the risk of overfitting.
CatBoostClassifier, XGBClassifier and GradientBoostingClassifier from Scikit-Learn are all examples of boosting classifiers based on decision trees. They are implemented in slightly different ways resulting in slightly different performance. Out of the three, CatBoost is currently considered to be state-of-the-art.

A variant of boosting is AdaBoost [4]. Multiple trees are trained and the final class is decided by majority vote. Like in Random Forrest the bootstrap method is used to create a new dataset for each tree. The difference in AdaBoost is that before using the bootstrap method the instances in the dataset are weighed based on whether the previous tree misclassified them or not. As a consequence AdaBoost can be over-sensitive to noisy data and outliers.

1.2 Solving the Problem

Authors: hehan20

Preparing the Data

The images in the DIDA dataset are of varying sizes (see Figure 6). Before they can be used to train the classifiers they must all be resized to the same size. To make the computations faster a small size of 50 by 50 pixels is chosen.

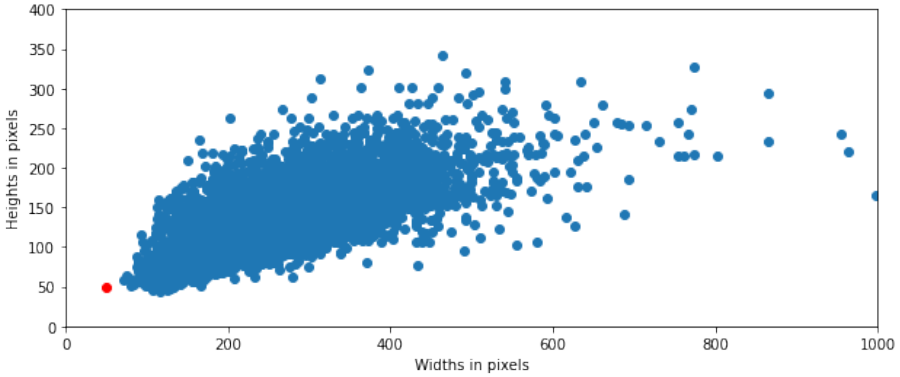
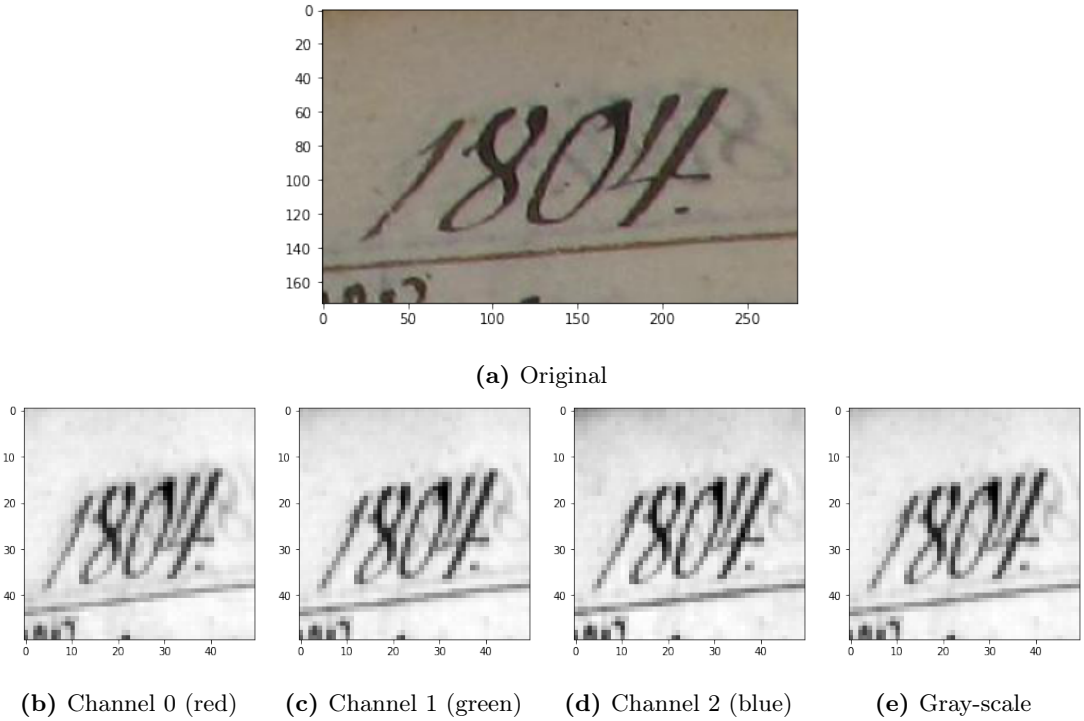


Figure 6: The heights and widths of the original unprocessed images. The chosen image size for resizing is marked with red.

The original images are in color, that is, each pixel is associated with three values: one for each color channel.



(a) Original

(b) Channel 0 (red) **(c)** Channel 1 (green) **(d)** Channel 2 (blue) **(e)** Gray-scale

Figure 7: Example of the effect of resizing an image to 50 x 50 pixels.

However, upon resizing we found that the images looked very similar in all three channels (see Figure 7). Thus, to further reduce the size of the feature space and keep the training times down, we merged the three channels to one, that is, we gray-scaled the images. As the final step the pixel values were normalized to be in the range 0 to 1 and the image vectors were reshaped to be 25,000-dimensional vectors instead of images of shape 50 x 50. A random sample of 25 of the resized, gray-scaled images can be seen in Appendix A.

Setting a Baseline

Before selecting what type of classifier to use for the three models it is a good idea to establish a simple baseline for the accuracy. For the modelCC it is easy to reach a high accuracy since the dataset is so unbalanced. By always predicting the majority class we get an accuracy of 97.5%. For the two other models, modelD and modelY, we can set a baseline by random guessing. We set the probability of picking a class to be the ratio the class occurs with in the dataset. We repeat the experiment 10.000 times and take the average. The result is seen in Table 1.

Choice of Model and Hyperparameters

The dataset is split into three parts: training, validation and test. We use 8,640 images for training, 2,160 for validation and 1,200 for test. We stratify the split on the CC labels making sure enough of the "odd length" digit-strings are present in all datasets. The classifiers are trained on the training dataset and then repeatedly tested up

Model	**Accuracy**
ModelCC	0.975
ModelD	0.227
ModelY	0.096

Table 1: Baseline accuracy for the three models.

against the validation set in order to choose the right hyperparameters. When model tuning is finished the final model performance is evaluated using the test set.

For training modelCC we need to consider how to deal with the unbalanced dataset to make sure the minority class does not get overlooked during training. We create a version of the training data set that is evenly balanced by using random oversampling on the training dataset. That is, we increase the number of images of the minority class by including them more than once in the training set until the number of minority class images equals the number of majority class images.

For each of the three models, modelCC, modelD and modelY, we first make a single training run of the different classifiers considered above using the default parameter values. This is not very accurate, but gives a good idea of what type of models to proceed forward with. The result is seen in Table 2. For modelCC the highest accuracy 0.9815 is reached with SVC with a radial basis function kernel using the resampled dataset. For modelD the highest accuracy is 0.6741 and for modelY 0.5838, both reached using the CatBoostClassifier. These accuracies are well above the baseline accuracies, showing the models have learned from the data.

Classifier	**ModelCC (unbalanced)**	**ModelCC (resampled)**	**ModelD**	**ModelY**
CatBoostClassifier	0.9722	0.9764	**0.6741**	**0.5838**
XGBClassifier	0.9713	0.9787	0.6426	0.5389
GradientBoostingClassifier	0.9690	0.9662	0.5542	0.4560
RandomForestClassifier	0.9694	0.9699	0.5398	0.4578
AdaBoostClassifier	0.9713	0.9139	0.3245	0.1967
SVC(kernel='linear')	0.9718	0.9616	0.4393	0.4028
SVC(kernel='rbf')	0.9713	**0.9815**	0.5444	0.4634
SVC(kernel='poly', degree=2)	0.9708	0.9634	0.4676	0.4079

Table 2: Accuracy for ModelCC, ModelD and ModelY using non-deep learning approaches. for ModelCC accuracy is shown both on the unmodified dataset (unbalanced) and the resampled dataset (balanced).

To find the optimal hyperparameters for the SVC with rbf kernel for modelCC we do a 5-fold cross validated grid search using the build-in SciKit-Learn method. We also want to examine the effect of oversampling and therefore create a pipeline with the RandomOversampler and the classifier. By using a pipeline we ensure that there is no data leakage during the grid search: for each fold the random oversampler only upsamples data from the training set. The following hyperparameters are tested:

- *Oversampling strategy.* We try the values "minority" and 0.5. The first creates a dataset with an equal amount of class 1 (minority) and class 0 (majority) images. The second creates a dataset with half as many class 1 images as class 0.
- *C.* For the strength of the regularization parameter we try the values 0.0001, 0.001, 0.01, 0.1, 1, 10. The lower the value, the larger the margins.
- *Class weight.* For class weight we try the values 'balanced' and None. If set to balanced, the parameter C of class i is set to $class\_weight[i] \cdot C$, where the weights are inversely proportional to the class frequencies.

The optimal hyperparameters are found to be *sampling_strategy=0.5*, *C=1* and *class_weight=None*. The accuracy is then 0.9825.

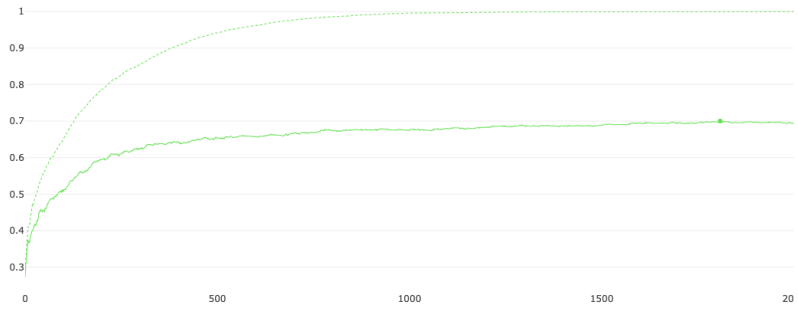
For modelD and modelY we search for the optimal hyperparameter settings for the CatBoostClassifier using the grid search option from the CatBoost library. This uses 3-fold cross validation. The following hyperparameters are tested:

- *learning_rate.* This is the parameter called λ in the above description of boosting. It determines the shrinkage to apply to each new learned tree before adding it to the model. For modelY the values 0.02, 0.1 and 0.3 are tested. For modelD the values 0.03 and 0.1 are tested.
- *depth.* This is the depth of the decision trees used in the boosting ensemble. Note, that CatBoost grows symmetrical trees so all leaves will have this depth. For modelY the values 1, 2 and 4 are tested. For modelD the values 4, 6 and 10 are tested.
- *l2_leaf_reg.* This adjusts the strength of the L_2-regularization applied to the cost function used for determining where to make the most optimal split. For modelY the values 1, 3 and 5 are tested. For modelD the values 1, 3, 5, 7 and 9 are tested.

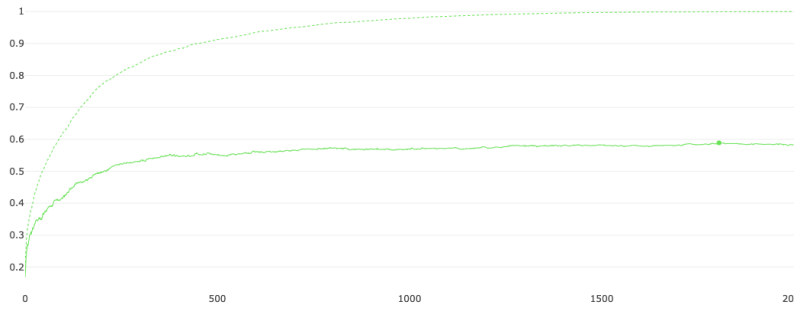
The idea behind not using the exact same parameter grid for the two grid searches is, that the two models are going to be very similar since they look at the same training data. By having two different grids we effectively widen the grid, while still keeping the training time down.
Using grid search the highest accuracy seen for modelY is 0.5481 using a learning rate of 0.1, a depth of 4 and a *l2_leaf_reg*-value of 3. This is however lower than the accuracy reached with the out-of-the-box default parameter

values. Similarly, the out-of-the-box accuracy is not improved for modelD. The highest accuracy seen during the grid search is 0.6615 with a depth of 10, a learning rate of 0.1 and a *l2_ leaf_ reg*-value of 5. This matches the claim that using CatBoost you can "*Reduce time spent on parameter tuning, because CatBoost provides great results with default parameters*"[6]. This is achieved by automatically adjusting the default parameters to match the input data. For modelD the default values are *l2_ leaf_ reg = 3*, *depth = 6* and *learning_ rate = 0.112735* and for modelY the values are *l2_ leaf_ reg = 1*, *depth = 4* and *learning_ rate=0.10000*.

Another important hyperparameter for the CatBoost Classifier is the number of iterations, that is, the number of trees in the ensemble. As default 1,000 trees are used. The effect of using a higher number of iterations is explored by using CatBoosts build-in method for plotting the train and test accuracy while training. In addition early stopping is used with a patience of 100 iterations to ensure we do not continue training unnecessarily long. From Figure 8 we see that the best result is achieved if the number of iterations is increased to 1,800. The accuracies then reach 0.5889 and 0.6597, respectively.



(a) ModelD



(b) ModelY

Figure 8: Accuracy on the training and validation data set during training of the CatBoostClassifier for modelD and modelY as a function of the number of iterations (trees). The solid line is the score on the validation set. The dashed line is the score on the training set. The best score on the validation set is 0.5889 reached at iteration 1808 for modelD. For modelY it is 0.6597 at iteration 1805.

Evaluation

The performance of the three final models on the training, validation, and test data can be seen in Table 3. For all three models we see the pattern that the accuracy is highest on the training data, lower on the validation data and then even lower on the test data. This is a very typical pattern. The model overfits on the training data, giving a high accuracy. The hyperparameters are chosen to give a good performance on the validation set, therefore we also slightly overfit to the validation data. Finally, the performance on the unseen test set is usually a bit lower than the accuracy on the validation set.

Model	**Training**	**Validation**	**Test**
ModelCC	0.9992	0.9796	0.9817
ModelD	0.9976	0.6907	0.6733
ModelY	0.9877	0.5856	0.5666

Table 3: Final accuracies of the three models on the training, validation and test datasets.

To capture the overall performance of the three models we create two new metrics: Character accuracy and Sequence accuracy. Sequence accuracy only counts an image as correctly classified if all three classifiers predict the correct label for that image. Character accuracy rewards 1/3 point towards the final accuracy score every time one of the classifiers predicts a correct label. For the non-deep learning approach we end up with:

- Character accuracy: 0.741
- Sequence accuracy: 0.406

Less than half of the digit-strings would be correctly recognized using this approach.

The code for finding the best possible non-deep learning models can be seen in `Question1.ipynb`.

2 Question 2

Authors: hehan20, krlor17, ruhei08

In this section we will explore different Deep Learning models to solve the CC-D-Y problem on the DIDA data. Specifically, the exploration will be restricted to Convolutional Neural Network (CNN) architectures with justification given in the following section 2.1.

2.1 Convolutional Networks for the CC-D-Y problem

Authors: krlor17

Assigning *a priori* preference to a specific model class characterized by parameters θ is well-framed as choosing a prior $p(\theta)$ on the model space of a machine learning task $\max_\theta p(\theta|\mathcal{D}) = \max_\theta p(\mathcal{D}|\theta)p(\theta)$ for a given data sample $\mathcal{D}$. Following this analogy to bayesian inference we argue that CNN models are reasonable for the CC-D-Y problem in terms of pragmatic/empirical arguments, corresponding in a sense to bayesian updates, and in terms of prior beliefs or assumptions on the given task. To this end we note that the task involves classification of images, specifically images of handwritten numbers, and that the task on an intuitive level boils down to checking whether *"the right shape is in the right spot"* for all three models.

Prior beliefs

The central assumption imposed on a task by CNN models is that of hierarchical decomposability – namely that the patterns we would like the model to detect can be decomposed into a combination of simpler patterns. For images this assumption is often intuitively justified by the example hierarchy *edges* $\rightarrow$ *simple shapes* $\rightarrow$ *complex shapes* which we expect to apply to this task as well. We thus expect CNN models to be able to effectively detect the *"right shape"*, but we also expect that a CNN model can be trained to detect whether such patterns are in the *"right spot"*. The equivariance to translations inherent to the convolution operation ensures that positional information is roughly preserved throughout the network even when pooling is applied. Hence, the deep hidden layer activations of a CNN corresponds exactly to positional feature maps of complex shapes i.e. shapes and the spots they are in.

Pragmatic argument

The pragmatic argument can shortly be summarized as "it worked for them"[7][8][9]. Convolutional Neural Networks has been the state-of-the-art models for image classification for at least the past decade, with some of the previous top scorers on the ImageNet dataset e.g. AlexNet[9] and ResNet[7] being small enough that they may feasibly be trained on somewhat average current-day equipment. Note also that CNNs have been shown to perform well with character recognition on the simpler MNIST[10] dataset of handwritten digits[8].

2.2 Constructing a CNN model for the CC-D-Y problem

Authors: ruhei08

A systematic search e.g. random grid search over all types and variations of CNN architectures would be computationally prohibitive and unlikely to yield any useful result within a realistic time limit. For that reason we again restrict ourselves to consider variations of well-tested architectures namely AlexNet[9] and residual networks i.e. ResNet[7] for models we will train "from scratch". AlexNet is notable for its simple structure, while the skip connections of the residual blocks in ResNet allow for much deeper convolutional networks, often leading to better generalization. In sections 2.2.a to 2.2.c we will explore aspects of constructing and training such models.
A common method for obtaining high performance with little training is by employing pre-learned representations of some high-performance model eg. VGG19[11] and adapting them to the problem at hand. This is known as transfer learning and will be explored for the CC-D-Y task in section 2.2.d.

The CNN models

For each of the three subproblems of the CC-D-Y problem we will construct three candidate models: 1) a CNN corresponding to a simplified AlexNet trained from scratch, 2) a ResNet inspired CNN trained from scratch and 3) a model using the pretrained VGG19 network for transfer learning.
A systematic search for candidate architectures could be implemented using e.g. the KerasTuner library, but such grid searches will ultimately still require *a priori* chosen constraints, and the weaker these are chosen to be the higher computational time would be required for proper search. Instead, we will perform a qualitative search for each architecture-type adhering to the following approach

1. Construct a model showing clear signs of overfitting during training
2. Find a regularization scheme to (hopefully) reduce overfitting and achieve generalizability.

Preparing the Data

For the neural networks we choose the size 150 by 100 pixels for the image resizing, because this allows us to make deeper neural network architectures with more pooling layers, while still keeping the training times reasonable. As long as we reduce the dimensionality enough before the flatten layer, a bigger image size will not necessarily result in larger models. We still reduce the number of channels to one since no extra information is gained by including all three, and we still normalize the pixel values to lie in the range 0 to 1. We do not flatten the images, but keep them as 2-dimensional Numpy Arrays of shape (100, 150, 1). Another difference from the non-deep learning approach is that we rename the "wildcard" class for modelD. Instead of 10, it is now called 5. This is to make the labels compatible with Keras. We still perform a train/val/test split in the same way as described in Question 1.

2.2.a Optimization strategy

Authors: krlor17

The training of a Deep Learning model is reduced to an optimization problem by the introduction of a surrogate loss — a loss assumed correlated with desired objective i.e. accuracy for classification. Following convention we use the crossentropy of the empirical label distribution and the label distribution encoded by the model as the surrogate loss and greedily minimize it using a gradient-based method.

The canonical gradient-based optimizer is the minibatch stochastic gradient descent (SGD) that updates the parameters based on an averaged gradient estimate $\theta \leftarrow \theta - \eta\left\langle \nabla_\theta f(x^{(i)}, y^{(i)}; \theta)\right\rangle_{i\in B}$ over a randomly sampled minibatch B. The hyperparameter η is the learning rate which modifies the size of the change in parameters pr. minibatch update. This hyperparameter is famously difficult to select a value for, since

a. the best suited learning rate for a task varies dependent on data, (mini)batch size and the model being trained.

b. the best suited learning changes *during* training, with a general consensus that the best strategy is a "large" value initially to get to the right neighborhood, followed by a gradually smaller value as the algorithm iteratively gets to a (local) minimum.

For the reasons mentioned in point (b) we will train models using the `ReduceLROnPlateau` callback, that reduces the learning rate by some constant factor when a plateau-criterion is met e.g. a number of k epochs without improvement of validation-loss. An alternative approach is to choose a learning rate schedule to decrease η according to some predefined function of the elapsed number of training steps e.g. exponentially – this of course introduces several other hyperparameters that can be difficult to set meaningfully without extensive grid-searching.

SGD faces several other problems during optimization of a non-convex function that in part are due to it being a first-order method i.e. it uses only the gradient and no higher-order derivatives. Especially its behavior in the neighborhood of local minima and saddle points (which are abundant in high dimensions) has inspired variants such as RMSprop and Adam.

Both RMSprop and Adam are popular choices, with RMSprop being the default optimizer in the Keras framework[12]. Common to both variants is that the parameter updates are now based on an "effective" learning rate, which corresponds to the base learning rate η modified by a term dependent on the gradients estimated in the previous minibatch updates. In RMSprop this is done by dividing the learning rate $\eta/\sqrt{\delta + r}$ by a exponentially decaying running average of the squared magnitude of past gradients r for each parameter, introducing the decay rate ρ as an extra hyperparameter. Adam additionally introduces momentum as exponential running average leading to two decay rates ρ_1, ρ_2 as extra hyperparameters. Note that RMSprop with momentum still isn't exactly equivalent to Adam that also features e.g. correction for initialization bias. Although in theory we have to set one more hyperparameter, we will primarily use the Adam optimizer for two reasons: 1) the default parameters suggested by the authors of the original paper tend to work well, and 2) we have a personal preference for the learning trajectories of Adam and so have used it more often, leading to a better intuition for parameter choice.

Finally, we will also employ batch normalization layers in our models to speed up convergence. While this is tech-

nically a choice of architecture, batch normalization has been shown to have great effect on the learning trajectory, with the authors of the original paper arguing that this is due to reduced internal covariate shift[13] and others arguing that the benefit comes from batch normalization leading to a smoother estimate of the loss surface[14]. Regardless of the mechanism, the clear consensus is that batch normalization has a beneficial effect on training.

2.2.b Effect of Regularization & Data Augmentation on Overfitting

Authors: hehan20

To illustrate the effects of different forms of regularization on the performance of a CNN, we construct a simple CNN architecture loosely inspired by AlexNet. Having few trainable parameters in the network will speed up the training time. The simple CNN is however big enough to be capable of learning to recognize the handwritten digits. In Table 4 we see that without any forms of regularization the validation accuracy of ModelCC, ModelD and ModelY reach 0.9894, 0.9162 and 0.8620. That is, the simple CNN easily beats the non-deep learning approach. The architecture of the simple CNN can be seen in Figure 9 (The print of the model summary can be seen in Appendix C). The simple CNN with no regularization have only 83,339 trainable parameters.

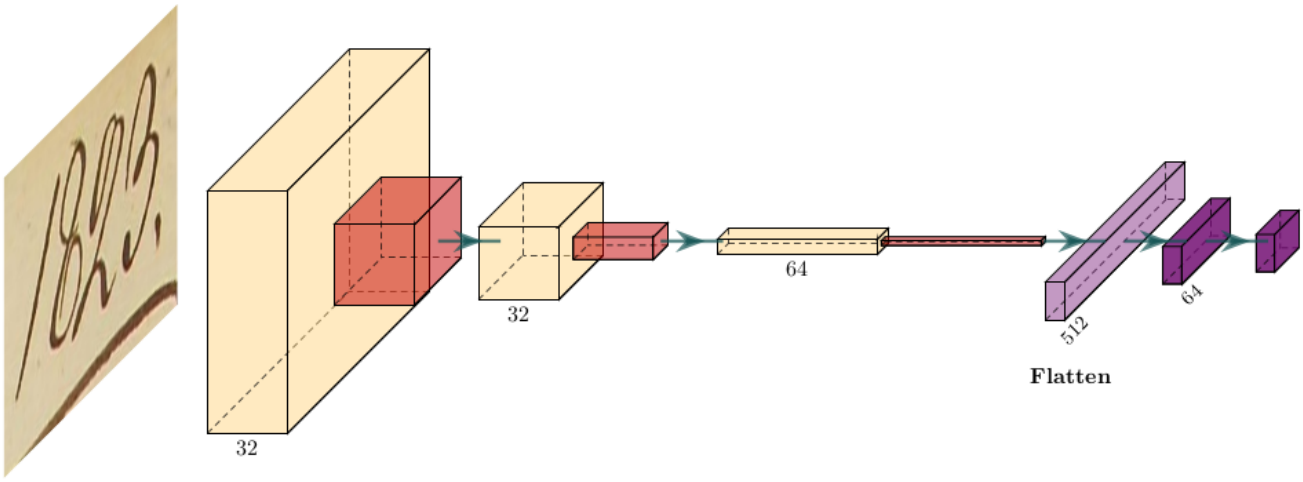


Figure 9: Architecture of the simple CNN. Three convolutional layers with kernel size 4 with max pooling of size 3 applied after each. The first two convolutional layers have 32 filters, the last have 64. Then a flatten layer, followed by a dense layer with 64 nodes and finally the output layer. Dependent on which model we are building the output layer has 1, 6 or 11 outputs. Made using PlotNeuralNet[15].

- *Early Stopping.* This is a simple idea, where the validation loss is monitored during training (or a different metric, if you prefer). If it stops improving, training is stopped. The patience parameter determines how many epochs the training continues after the best result is observed. Early Stopping prevents overfitting to the training data by stopping, when the learned model no longer generalize to the held out validation data. In Figure 23 in Appendix B we see exactly this behavior for our simple CNN models: the best performance is

reached before the 20^{th} epoch, and thanks to Early Stopping we do not move past this point. We have used Early Stopping with a patience of 10 and with "restore best weights" set to True.

- *Dropout.* When using Dropout as a layer in the neural network, a fraction of the weights from the previous layer are set to zero. The weights set to zero are chosen randomly, and it changes which are chosen for each training pass. This has the effect of making the trained network more robust, forcing it to use all nodes for predicting. In effect, you can think of this as an ensemble method where we train multiple sub-networks thereby making the final classifier stronger. Dropout of course is only applied at the training stage, since it would hinter correct classifications for the final model. We have added a Dropout layer, that drops 50% of the weights just before the final Dense layer. The effect can be seen in Figure 24 in Appendix B. We observe that learning is slower and that there is less overfitting.

- *Spatial Dropout.* In a CNN it is a better idea to use Spatial Dropout after a convolutional layer than to use normal Dropout. With Spatial Dropout entire feature maps are dropped instead of just individual weights. This deals with the problem that neighborhoods are often highly correlated, so that normal dropout would need a high dropout rate to have an effect, thereby introducing other problems. In our simple CNN we added Spatial Dropout of 20% after each of the three convolutional layers. We see that, like normal Dropout, this results in slower convergence and, more importantly, less overfitting (see Figure 24 in Appendix B).

- *Batch Normalization.* As described above, Batch Normalization ensures faster convergence. We have tested the effect of using Batch Normalization after each of the MaxPooling layers. Sure enough, in Figure 25 in Appendix B we see this results in faster convergence, and also an increase in validation accuracy.

- *L1 and L2 Regularization.* The idea of weight regularization is to add a penalty to the loss function ensuring the learned weights stay small. Keeping the learned model simple in this way, makes it less likely that the model overfits to the training data. For L1 regularization we adjust the loss function l to be:

$$l = L(y, \hat{y}) + \lambda \sum_{w \in W} |w|$$

and for L2 regularization:

$$l = L(y, \hat{y}) + \lambda \sum_{w \in W} w^2$$

where $L(y, \hat{y})$ is the normal loss function measuring the difference between the actual labels y and the predicted labels $\hat{y}$ and w are the weights of the network. The parameter λ adjusts the strength of the weight regularization. We have tested the effect of both L1 and L2 regularization on the kernels of our convoluational layers with a strength of $\lambda = 0.01$. As seen in Figure 26 in Appendix B this is however too strong a regularization for our network: no learning occurs.

- *Data Augmentation.* The more data you have, the easier it is to train a neural network to recognize patterns without risking overfitting. With too few samples you risk the model just memorizes the training data and is

unable to generalize to unseen data. In image recognizing tasks an easy way to generate more data is image augmentation. The idea is to perturb the images slightly before they are fed into the network during training, but still keep the image labels as-is. This is implemented as the first layer in the network and the result is the neural network never sees exactly the same image twice. Like with dropout, this layer is only used in the training phase. For our simple CNN we have tested the effect of:

- `RandomRotation(0.04)`: random rotation in the range $[-0.04 \cdot 2\pi, 0.04 \cdot 2\pi]$ or in other words $\pm 14.4°$.
- `RandomContrast(0.1)`: the pixel values of the input image are adjusted to be $(x - mean) \cdot contrast\_factor + mean$, where the $contrast\_factor$ is chosen randomly in the range $[1 - 0.1, 1 + 0.1]$.
- `RandomTranslation(width_factor=0.05, height_factor=0.05)`: the image is randomly shifted $\pm 5\%$ vertically and horizontally.

We have been careful to choose low amounts of augmentation to ensure the digits are still readable and, most importantly, still within the frame. Examples of the effect on the input images of the applied image augmentation can be seen in Figure 30 in Appendix G. In Figure 27 in Appendix B the effect on performance of adding the image augmentation layer to the simple CNN is shown. We see a clear boost in performance and no overfitting. Again, we see the regularization results in slower convergence.

	ModelCC	**ModelD**	**ModelY**
Plain	0.9894 (20)	0.9162 (20)	0.8620 (20)
Early Stopping	0.9917 (13)	0.9185 (6)	0.8769 (10)
Dropout	0.9926 (17)	0.9398 (15)	0.8829 (17)
Spatial Dropout	0.9898 (14)	0.9407 (20)	0.9153 (25)
Batch Normalization	0.9921 (4)	0.9412 (9)	0.9042 (11)
L2 Regularization	0.9815 (83)	0.2620 (14)	0.1097 (10)
L1 Regularization	0.9676 (4)	0.2620 (6)	0.1097 (8)
Data Augmentation	0.9935 (24)	0.9597 (38)	0.9301 (36)

Table 4: Accuracy of a simple CNN for the three classification tasks with different forms of regularization. In parenthesis is stated the epoch this accuracy was reached. All results better in terms of accuracy or convergence speed compared to the plain network are marked with blue.

A final version of the three simple CNN models are trained using a combination of the regularization techniques described above. For each model the beneficial types of regularization are marked with blue in Table 4. Either because they resulted in an increase in validation accuracy, an increase in convergence speed, or both. The final accuracies of the simple CNNs with regularization can be seen in Table 5. The learning curves showing the training

and validation accuracies and losses measured during the final training of the regularized models are shown in Figure 28 in Appendix B.

The code for experimenting with regularization on the simple CNN can be seen in `Question2_Simple_CNN.ipynb`.

2.2.c Creating a ResNet Inspired CNN

Authors: krlor17

To achieve a better performance than with the simple CNN we create a deeper model with more trainable parameters. The ResNet architecture is characterized by residual blocks. These consist of a number of sequentially composed convolutional layers that are then additively combined with their input i.e. have a so called "skip connection". In our implementation each residual block also has a max pooling layer at the end. Skip connections allow us to build much deeper convolutional networks without facing the problem of degradation, and deeper models tend to generalize better[7]. The methodology for architecture search was to 1) find a ResNet inspired architecture that shows clear signs of overfitting for the most complex subproblem, and 2) use that architecture for all problems, possibly with individual regularization schemes. With the label structure discussed in the introduction we deem the Y-problem the most challenging, since it has the most classes to predict. By iteratively adapting the number of residual blocks, the number of layers in each block and the number of channels / filters in each block we found an architecture showing clear signs of overfitting cf. figure 11. This architecture consists of 4 residual blocks, each of depth 4, having the following number of filters in order: 32, 64, 64, 128. The kernel sizes were all $(3, 3)$ with stride 1, and all maxpooling layers used disjoint pools of $(2, 2)$. A single dense layer with 128 units follows the flattening before the softmax output units. All layers use ReLU activation. The architecture is illustrated in fig. 10 and more details are given in appendix D.

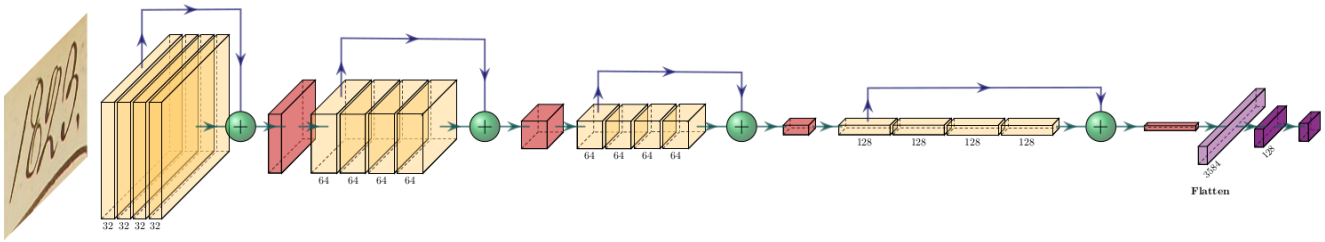


Figure 10: Architecture of the ResNet inspired network. 4 residual blocks each with 4 convolutional layers (yellow) with kernel size 3, followed by an "add" layer (green) and max pooling of size 2 (red). The residual blocks are followed by a flatten layer, followed by a dense layer with 128 nodes and finally the output layer. Dependent on which model we are building the output layer has 2, 6 or 11 outputs. The Y model w. 11 outputs has 1,283,691 trainable parameters. Made using PlotNeuralNet[15].

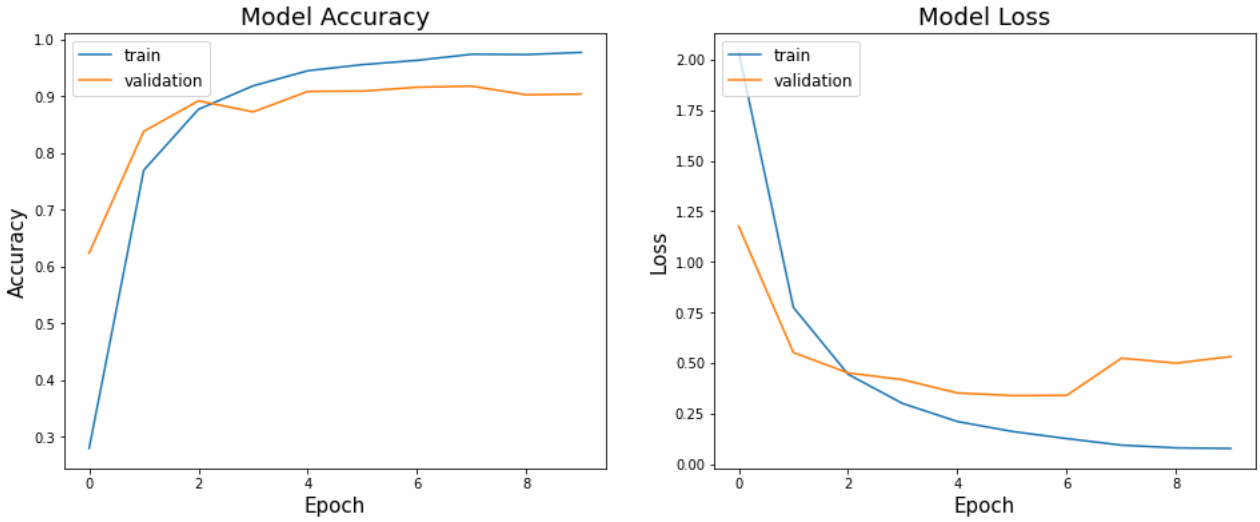
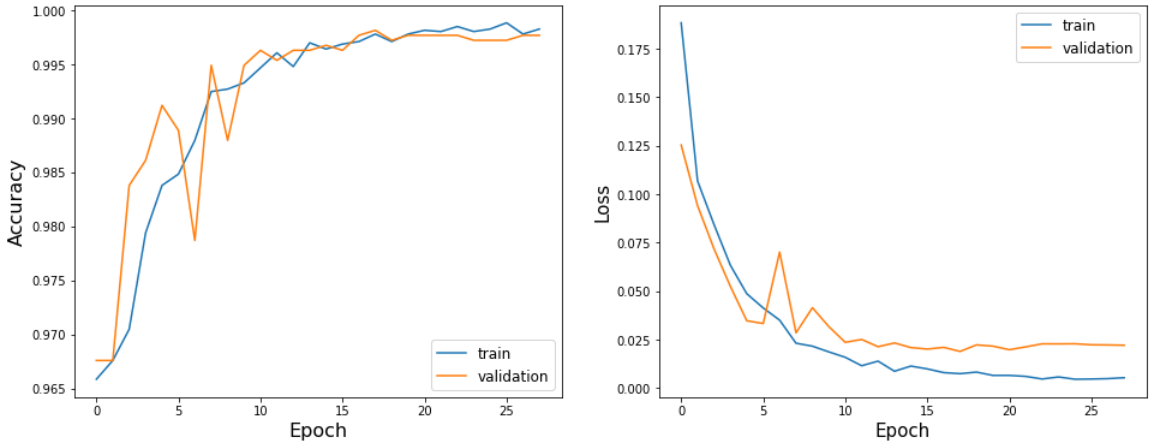


Figure 11: An unregularized ResNet inspired model showing clear signs of overfitting.

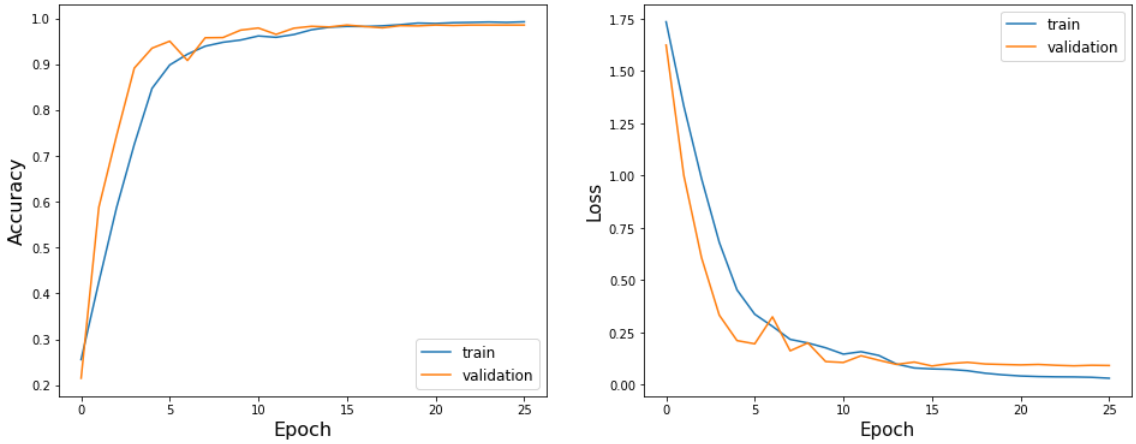
Using this architecture we could then search for individualized regularization schemes for each of the submodels i.e. CC, D and Y. It turned out, however, that the same combination of dropout worked fairly well for all three subproblems. We used spatial dropout after each convolutional layer with a rate of 8% and dropout with rate 10% for the dense layer. In addition, batch normalization was used following every convolutional layer to improve convergence. For the binary CC model we choose to use 2 units with softmax activation for output, as this enables us to apply Grad-CAM to all the models using the same method. For this reason the CC labels were one-hot encoded prior to training to be used with the Keras loss function `categorical_crossentropy`. Figure 12 shows the learning curves of the training of all three submodels, with all of them using the same architecture and regularization scheme save of course different numbers of output units. Following the labeling scheme discussed in the introduction and this section we used the following number of output units CC: 2, D: 6 and Y: 11.

The training was done using the Adam optimizer with an initial learning rate of 1×10^{-3} and default decay parameters. Additionally, the training used the `EarlyStopping` and a `ReduceLROnPlateau` callbacks with a patience of 10 epochs and 2 epochs respectively. The learning rate reduced by a factor 0.5 whenever the validation accuracy failed to improve for 2 consecutive epochs. The learning curves of the Y and D models indicate well suited regularization, with the validation accuracy being higher than the training accuracy for a long stretch of the training. This of course due in part to the dropout not being active during validation. The learning trajectory of the CC model is less stable than for the other two submodels, probably due to the class imbalance. A further note is that the inherent complexity of the CC model is likely smaller than the D and Y problems, so perhaps an even stronger regularization scheme could be in order. The code used for the training of the model is available in `Question2_ResNet.ipynb`.

(a) CC model



(b) D model



(c) Y model

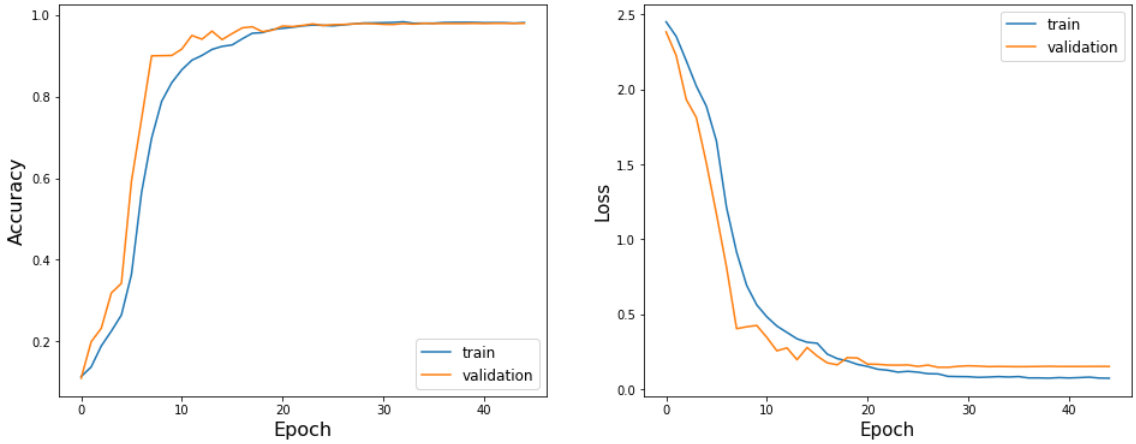


Figure 12: Learning curves from the training of the 3 submodels based on the same ResNet inspired architecture and using the same regularization scheme.

2.2.d Using a Pretrained Model: Transfer Learning

Authors: ruhei08

A common way to use deep learning on small datasets is to use a convolutional neural net that is already trained. We have 12.000 pictures in our dataset, which is small, compared to e.g. the mnist-dataset of 60K images. Usually these pretrained networks have been trained on millions of images to do image classification. We have chosen to use the pretrained network VGG19, which consists of 19 layers and is trained on over a million images to classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. The model-structure is shown in Appendix F.

Tensorflow comes with a few of these pretrained networks. We chose VGG19 because it's structure is fairly similar to what the ResNet-like model we trained from scratch.

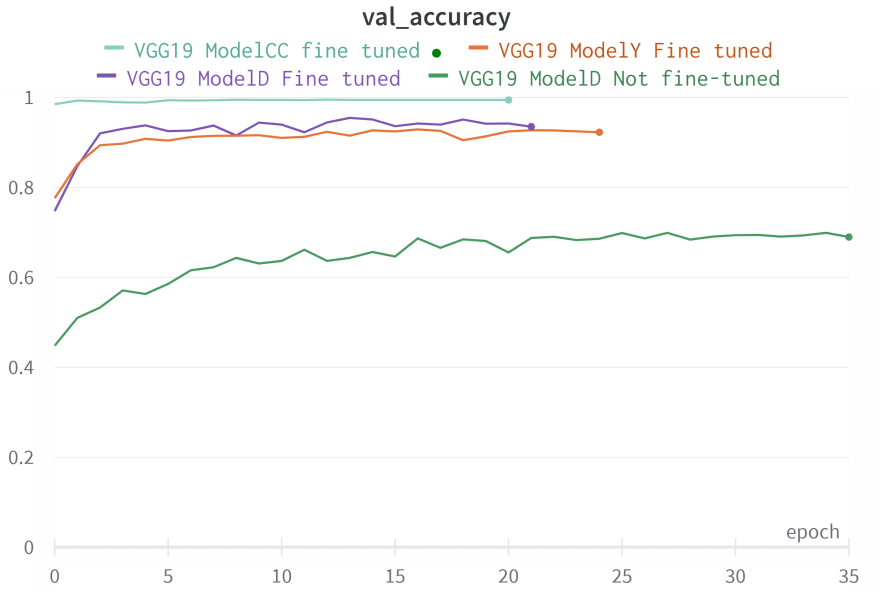
The effectiveness of transfer learning comes from the fact that CNN's for image classification consists of two parts. A convolutional base with a series of convolutions and poolings, and a densely connected classifier. By taking out the top classifier layer and replacing it with your own trained on new data, you can utilize all of the features learned in VGG19's convolutional base.

Although we already by replacing the top classifier have a model that "works", we can improve the performance by "fine tuning" the model for our use. Fine tuning is a process, where you unlock some of the layers in the model. You can use three different strategies for fine tuning[16].

- Strategy 1: Unlock all layers in the pretrained network and train the entire model from scratch. This requires a large dataset and a lot of compute.
- Strategy 2: Unlock some of the top layers and train them. This utilizes the fact that the lower layers of the model have learned problem unspecific/generic features like edges and colors, and the later layers have learned problem specific features or representations. If we retrain the problem specific layers, this should gives us the benefit of lower training time and high performance even though the dataset is small.
- Strategy 3: Freeze all layers and only train the classifier layers. This strategy can be use if you either have a very small dataset and very little computing power or if you have a problem that is very similar to the problem that the pretrained network is trained to solve.

We tried to apply both strategy two and three. It turned out that strategy 2 by far was the most accurate strategy as is shown in Figure 13. This is because strategy 2 learns to solve the specific problem at hand by overwriting the late layers.

The pretrained network VGG19 takes a three channel image as input. It is therefore not compatible with the greyscaled images we have used throughout this paper. We made sure to input 3 channel images instead of 1 channel images otherwise the data was prepared similarly. Like for the simple CNN and the ResNet inspired models we used image augmentation as the initial layer in the network thereby effectively increasing the size of the dataset.



(a) Validation accuracy during training

Figure 13: There is a clear increase in performance if you fine-tune using strategy two(Lower layers frozen) instead of strategy 3(All layers frozen).

In the process of fine-tuning we found out, that overwriting all of the layers ind VGG19 block5 using a very low learning rate gave us the best performance with the pretrained network. The training process of the fine tuned models based on VGG19 are plotted in Figure 29 in Appendix E.
Before using the pretrained network we expected the pretrained model to be the most accurate. But it turned out to be outperformed by a small margin by the ResNet-model we trained from scratch. The reason for this could be, that the pretrained model using fine-tuning strategy 2 does have some features learned, that is not usable to our specific problem. Further investigations of this problem could be to apply strategy 1 and use VGG19-model architecture but retrain it on our data set, though it requires more computing power and likely a larger dataset.

The code for using transfer learning to build a model can be seen in `Question2_Transfer_Learning.ipynb`.

2.3 Model selection

Authors: ruhei08

The final accuracies of the best simple CNN, ResNet inspired model and transfer learned VGG19 models are shown in Table 5. As explained in Question 1, we again see the training accuracy is higher than the validation accuracy. The Character and Sequence accuracies were also explained in Question 1. The overall best performing models are the ResNet inspired models. Here we achieved a sequence accuracy of 97.00%. We will look further into their performance in Question 3.

We chose ResNet-model as our preferred model based on the better performance. Although the ResNet-model did perform the best, it is worth mentioning, that the training-expense of this model was the highest. One advantage of the pretrained VGG19 model was that it started to converge already after 10-15 epochs as shown in figure 18 - where ResNet need to be trained for about 40-50 epochs before converging. This indicates that the pretrained model already has learned the representations of the lower layers, and therefor performs better out-of-the-box.

Model		Training Acc.	Validation Acc.	Test Acc.	Character Acc.	Sequence Acc.
Simple CNN	CC	0.9987	0.9968	0.9950	0.9717	0.9283
	D	0.9764	0.9648	0.9650		
	Y	0.9719	0.9528	0.9550		
ResNet	CC	0.9995	0.9981	0.9967	0.9875	0.9700
	D	0.9946	0.9852	0.9875		
	Y	0.9913	0.9792	0.9783		
VGG19 TL	CC	0.9999	0.9974	0.9971	0.9805	0.9425
	D	0.9987	0.9781	0.9733		
	Y	0.9990	0.9599	0.9496		

Table 5: Accuracy-based metrics of the best simple CNN, ResNet inspired and transfer learned VGG19 models.

NB. Character- and Sequence Accuracy are computed on the test set

3 Question 3

Authors: krlor17, ruhei08

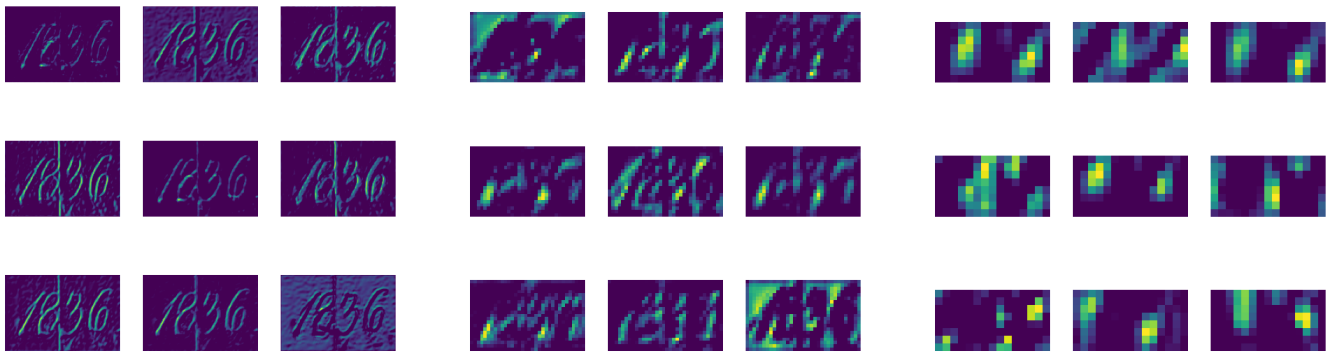
In this section we will look closer at the ResNet inspired models using standard visualization techniques for CNN models. These techniques help confirm that the model has learned generalizable concepts e.g. by checking what patterns maximize filter response in the deep layers or seeing what part of an input image contributes to correct / incorrect classification. Further we will look closer into model performance by investigating the images the models misclassify.

3.1 Visualization techniques for CNN models

3.1.a Visualizing activations

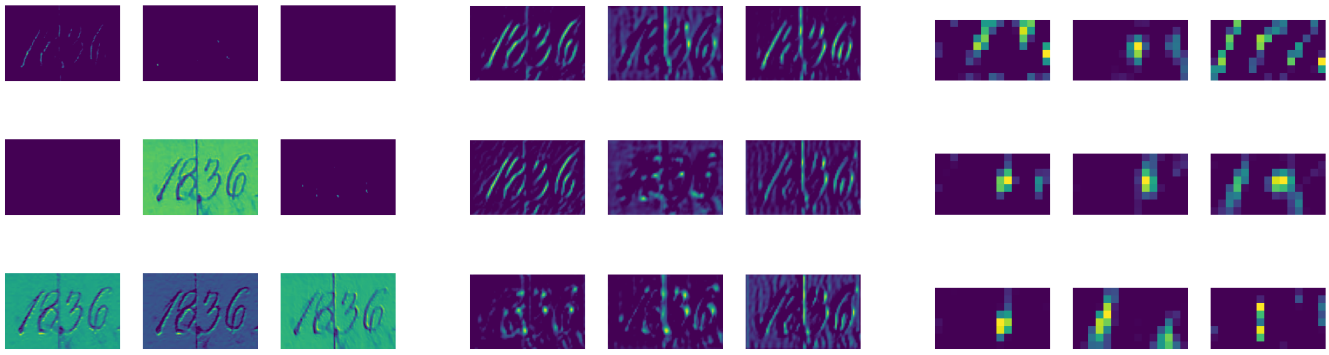
Authors: ruhei08

We can display feature maps that are outputs from convolution and pooling layers in our network. This visualized image is often called an activation, because it is the output of the activation function of the layer. These activations shows us how an input image is being divided into different filters learned by the network.
In Figure 14 we have plottet nine activations for three different layers from our ResNet-like modelCC classifier. This shows a general characteristic of representations learned by neural networks. The first layers carries more information of the original input, and the later layers are much more abstract.



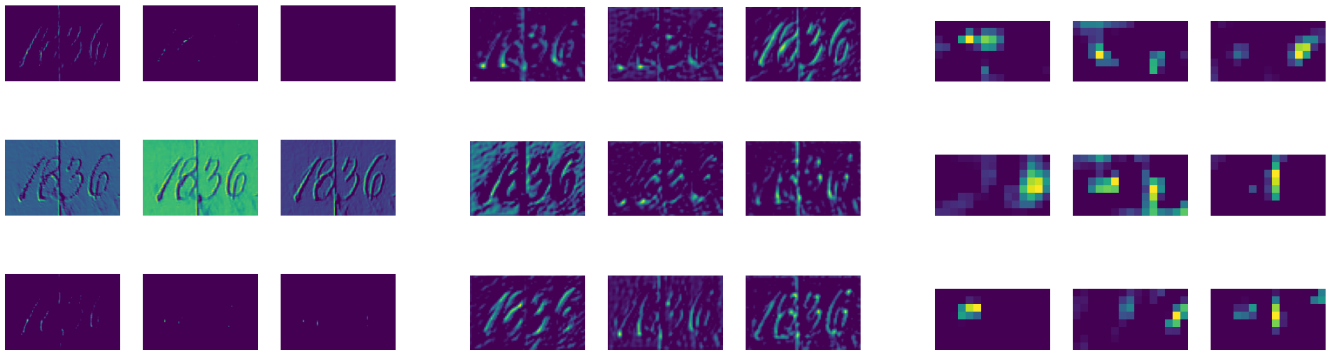
(a) ModelCC: First conv layer **(b)** ModelCC: Middle conv layer **(c)** ModelCC: Late conv layer

Figure 14: Activations of different convolutional layers in ModelCC. (9 out of 32 channels are shown)



(a) ModelD: First conv layer **(b)** ModelD: Middle conv layer **(c)** ModelD: Late conv layer

Figure 15: Activations of different convolutional layers in ModelD. (9 out of 32 channels are shown)



(a) ModelY: First conv layer **(b)** ModelY: Middle conv layer **(c)** ModelY: Late conv layer

Figure 16: Activations of different convolutional layers in ModelY. (9 out of 32 channels are shown)

The first activation output in Figure 14a is from the first convolutional layer in our modelCC. We can interpret this first layer as a collection of edge detectors.
The second layer (Figure 14b) is a bit more abstract and encode more complex structures. The third layer (Figure 14c) is much more specific and carries only little information.
As we glance over Figure 15 and Figure 16 we see that the activations are different, but that it is hard to make out what the specific difference between them is.

The code for visualizing the activations can be seen in `Question3_Activation_of_Feature_Maps.ipynb`.

3.1.b Maximizing filter response

Authors: krlor17

To explain the how and why a trained CNN works a fundamental question is *"what patterns does the kernels encode?"*. Inspecting the values of a kernel matrix gives only vague insight in the deep layers where the most

problem-specific patterns are supposed to appear. A better approach for image data is to construct an image that results in a large *response* in the activations the corresponding feature map of the kernel. A feature map F is a 2-dimensional array and does not have any intrinsic size, so in order to use gradient based optimization we define the response $r : \mathbb{R}^2 \to \mathbb{R}$ as some aggregation of the entire feature map e.g. the average

$$r(F) = \frac{1}{h_F w_F} \sum_{i=0}^{h_F-1} \sum_{j=0}^{w_F-1} F_{ij} \, . \tag{1}$$

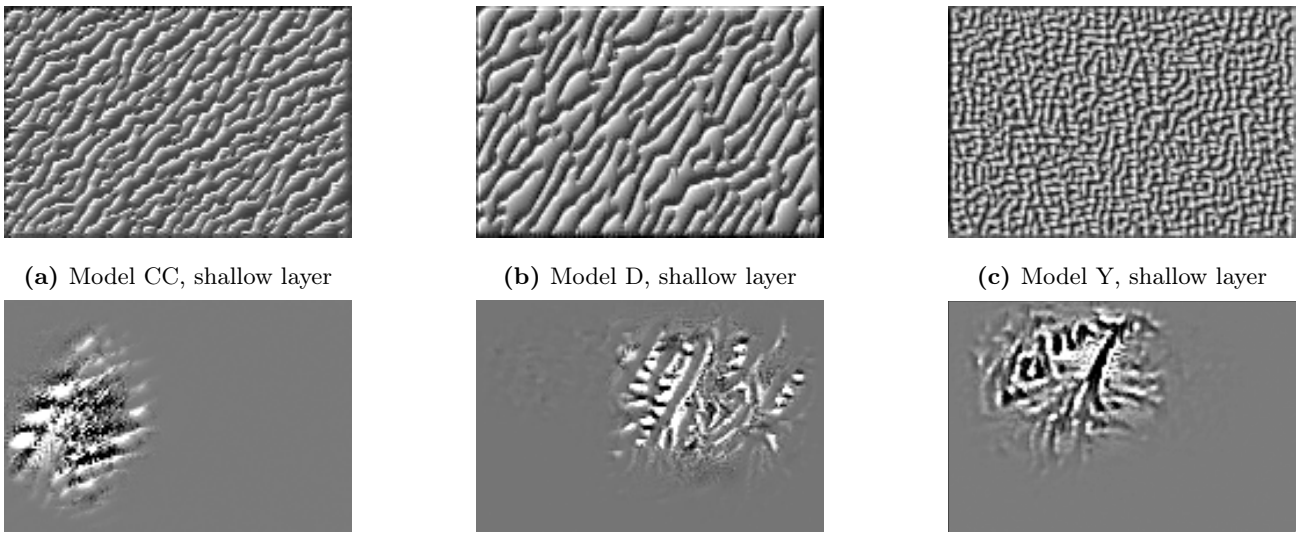
where h_F, w_F are the height and width of the feature map. We note that the feature map F is a differentiable function of the input image $X = (x_{ij})_{h \times w}$. This means that as long as $r(F)$ is chosen as a differentiable function such as the average then $\frac{\partial}{\partial x_{ij}} r(F(X))$ is defined and computable. This also works for $r(F) = \max_{ij}\{F_{ij}\}$ that has a well-defined gradient when there is a unique maximum, and tensorflow chooses a specific subgradient when there are several maxima[17].

We can then construct a greedy estimate $\hat{X}$ of the input image maximizing $r(F(X))$ by

1. Initialize an input image $\hat{X}$ as random noise
2. Update $\hat{X}$ for k iterations with gradient ascent $\hat{X} \leftarrow \hat{X} + \eta \nabla_x r(F(\hat{X}))$

In addition, we normalize the gradient $g = \nabla_x r(F(\hat{X}))$ by the Root Mean Square of component-sizes i.e. $g' = g/\sqrt{\langle g^2 \rangle + \delta}$ where δ is some small value to avoid computational issues. This serves to lessen the contribution of random variations in the gradient and provides a less noisy and more pronounced result.

Examples of how this looks for our models can be seen in Figure 17. This shows clearly that the shallow layers in the models look for simple patterns, while the deeper layers look for more textured and complex patterns.



(a) Model CC, shallow layer **(b)** Model D, shallow layer **(c)** Model Y, shallow layer

(d) Model CC, deep layer **(e)** Model D, deep layer **(f)** Model Y, deep layer

Figure 17: Visualization of maximal filter response for sample filters in deep and shallow layers of each of the three models. For the shallow layers we use mean response and for the deep we use max. response.

The code for constructing these filter visualizations can be found in `Question3_Filter_Maximization.ipynb`.

3.1.c Gradient-weighted Class Activation Maps (Grad-CAM)

Authors: krlor17

For the classification task, the CNN assigns an input X to a class c based on the learned features in the deep layers. To help explain why the model assigns the correct class to some images and incorrect classes to other images it would be helpful to know *what the network 'looks' at* to make a decision. The technique of Gradient-weighted Class Activation Maps (Grad-CAM)[18] attempts to answer this question by enabling us to construct a coarse intensity-map of what parts of the input image X contributes to driving up the predicted class-score y^c.
Let F^k be the feature map / activations of the kth kernel of the last convolutional layer and let $\frac{\partial y^c}{\partial F^k}$ denote the gradient of the class score y^c wrt. to the elements F^k_{ij} of the feature map. The class score y^c is the weighted sum of input from the second-last layer of the network – softmax is not applied. The idea of Grad-CAM is to then assign a weight to each kernel / feature map that is its average gradient component

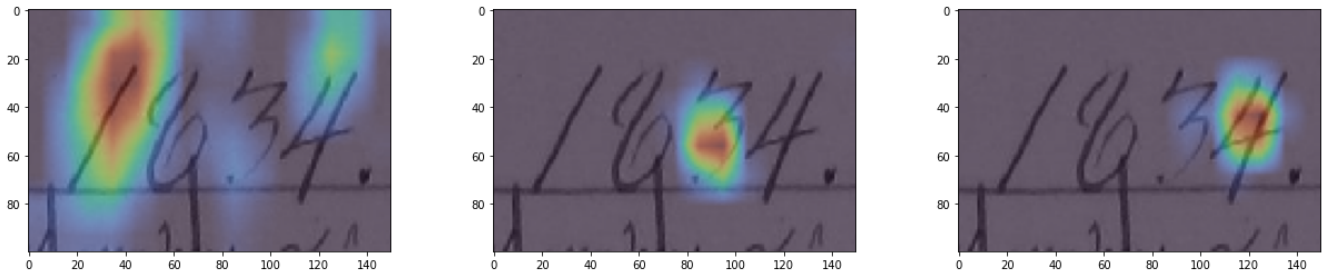
$$\alpha^c_k = \frac{1}{h_F w_F} \sum_{i=0}^{h_F-1} \sum_{j=0}^{w_F-1} \frac{\partial y^c}{\partial F^k_{ij}} \tag{2}$$

and then compute a weighted combined feature map $\tilde{F}_c$, discarding negative contributions using a ReLU mapping as a component-wise $\max\{0, z\}$

$$\tilde{F}_c = \text{ReLU}\left(\sum_k \alpha_k F^k\right). \tag{3}$$

Finally, this combined feature map $\tilde{F}_c$ can be scaled up to match the image input size using interpolation resulting in a heatmap that can be overlayed onto an input image X. Note that if we had used the negative gradient $-\frac{\partial y^c}{\partial F^k_{ij}}$ in eq. (2), we would have obtained a contrafactual heatmap i.e. a visualization of what parts of the image that would contribute to the network making a different decision.

Applying this technique to our models show very clearly that the models classify the images based on the correct areas of the image. In Figure 18 we see that Model CC mainly looks at the first digit, the Model D looks correctly at the decade digit and the model Y looks correctly at the final digit. The code used to make these visualizations can be found in `Question3_GradCAM.ipynb`.



(a) Model CC **(b)** Model D **(c)** Model Y

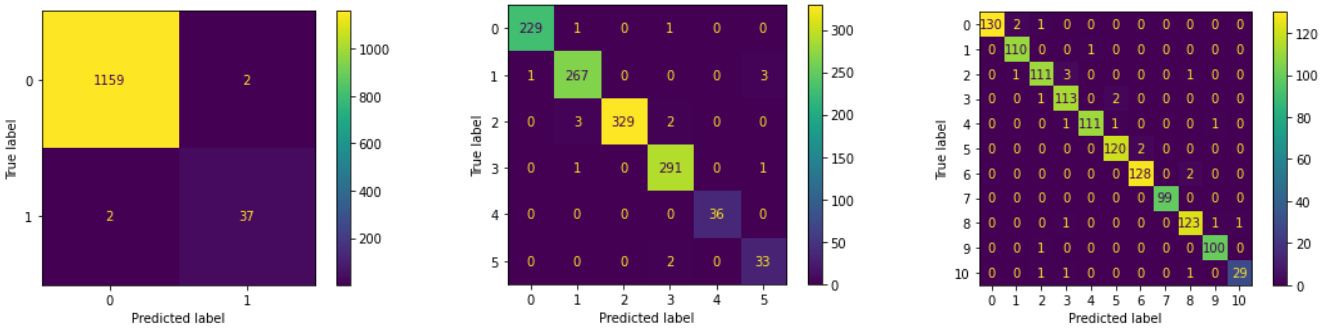
Figure 18: Grad-CAM visualizations clearly show the models are making classifications based upon the correct areas of the images.

3.2 Model Performance

Authors: ruhei08

The ResNet inspired models we have trained to recognize handwritten digits according to the CC-D-Y modelling strategy have a very high accuracy (see Table 5). On out test set consisting of 1,200 images this means that the CC model misclassifies only 4 images, the D model 15 images and the Y model 26 images. To better understand which classes are being mixed up with each other we look at the confusion matrices for the three models in Figure 19. We see that there is in fact no real mix-up occurring between classes: the errors are evenly distributed. In Appendix H all the misclassified images are shown. On closer inspection it turns out that 2 out of the 15 mistakes the D classifier made are due to wrongly labelled images. The classifier gets it right! The same picture repeats itself for the Y classifier. Here it is 7 out of the 26 images that have wrong labels, but that are classified correctly by our model.

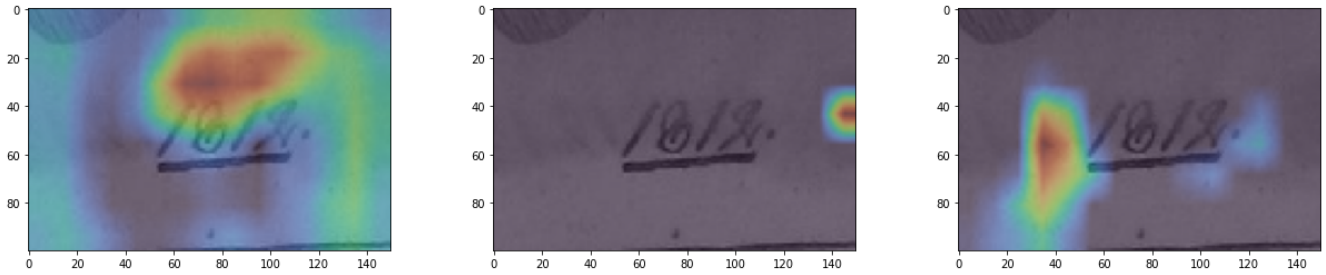
Some of the misclassified images depict digits written across the gutter (the center fold of the book). This creates a shadow in the image that both the D and Y classifier mistakes for the digit 1. The other misclassifications seems to be more miscellaneous stemming from for example blurry images, smudges, lines drawn on the paper, etc.



(a) Model CC **(b)** Model D **(c)** Model Y

Figure 19: Confusion Matrix for the predictions of the ResNet inspired models on the test set.

One of the images is misclassified by all three classifiers: the number 1812 with a line under. We use the Grad-CAM approach to visualize what went wrong, see Figure 20. The intensity maps show that the classifiers does not look in the right places for the digits in stark contrast to what we saw for a correct classification (Figure 18). This could be because the digits are smaller than in other images. Most of the images in the dataset are cropped to fit the digits.



(a) Model CC **(b)** Model D **(c)** Model Y

Figure 20: Grad-CAM visualizations of a misclassified image. The heatmaps show the classifiers are not looking in the right areas.

Interestingly, the CC model looks at a large seemingly smudged area above the digits in figure 20a. Relating to figure 17d we suspect that the CC model doesn't look so much for highly-specific patterns rather than evaluating the position of blobs. The code for investigating the performance of the final models can be seen in `Question3_Model_Performance.ipynb`.

Conclusion

We have successfully trained three classifiers to recognize handwritten digits in the DIDA dataset according to the CC-D-Y modelling strategy. Using an architecture similar to ResNet with convolutions and skipped connections we reached a sequence accuracy of 0.9700 and a character accuracy of 0.9875. The ResNet inspired models outperformed simple CNN models and, surprisingly, transfer learned models build upon the pretrained VGG19 net. We suspect that a broader hyperparameter search and careful training could lead to even better transfer learned models. All models benefited from the use of image augmentation during training. Unsurprisingly, all of the neural network models outperformed the non-deep learning models by a wide margin. For the non-deep learning models the best performance was seen using the ensemble method CatBoost for the D and Y models. Because of the unbalanced dataset for model CC the tree-based models did not perform as well. Here the best result was achieved with a support vector machines model with an rbf-kernel and an oversampled version of the dataset where the number of minority class images was increased to be half that of the majority class. With the non-deep learning approach we only reached a sequence accuracy of 0.406 and a character accuracy of 0.741. Using different visualizations of the trained networks we have shown that the ResNet models appear to have learned generalizable patterns. From the Grad-CAM visualizations of correct examples it is clear that the model considers the correct positions, and for the D and Y models appear to match these against learned complex patterns, whereas the CC model appears to mostly consider positional information. The ResNet models do not have any distinct bias against certain classes of digits, and even predicts correctly some examples where the given labels are incorrect.

References

[1] Huseyin Kusetogullari, Amir Yavariabdi, Johan Hall, and Niklas Lavesson. Dida: The largest historical handwritten digit dataset with 250k digits. `https://github.com/didadataset/DIDA/`. Accessed: 2022-01-24.

[2] Huseyin Kusetogullari, Amir Yavariabdi, Johan Hall, and Niklas Lavesson. Digitnet: A deep handwritten digit detection and recognition method using a new historical handwritten digit dataset. Big Data Research, 2020.

[3] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. An Introduction to Statistical Learning: with Applications in R. Springer, 2013.

[4] Scikit learn user guide. `https://scikit-learn.org/stable/user_guide.html`. Accessed: 2022-01-27.

[5] Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. Journal of machine learning research, 2(Dec):265–292, 2001.

[6] Catboost official webpage. `https://catboost.ai`. Accessed: 2022-01-27.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[8] Shivam S Kadam, Amol C Adamuthe, and Ashwini B Patil. Cnn model for image classification on mnist and fashion-mnist dataset. Journal of Scientific Research, 64(2):11, 2020.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25:1097–1105, 2012.

[10] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. IEEE Signal Processing Magazine, 29(6):141–142, 2012.

[11] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[12] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International conference on machine learning, pages 448–456. PMLR, 2015.

[14] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Mądry. How does batch normalization help optimization? In Proceedings of the 32nd international conference on neural information processing systems, pages 2488–2498, 2018.

[15] Haris Iqbal. PlotNeuralNet. `https://github.com/HarisIqbal88/PlotNeuralNet`. Accessed: 2022-01-29.

[16] Transfer learning from pre-trained models. `https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751`. Accessed: 2022-01-30.

[17] Tensorflow github repository. `https://github.com/tensorflow/tensorflow`. Accessed: 2022-01-29.

[18] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In Proceedings of the IEEE international conference on computer vision, pages 618–626, 2017.

A Effect of Resizing

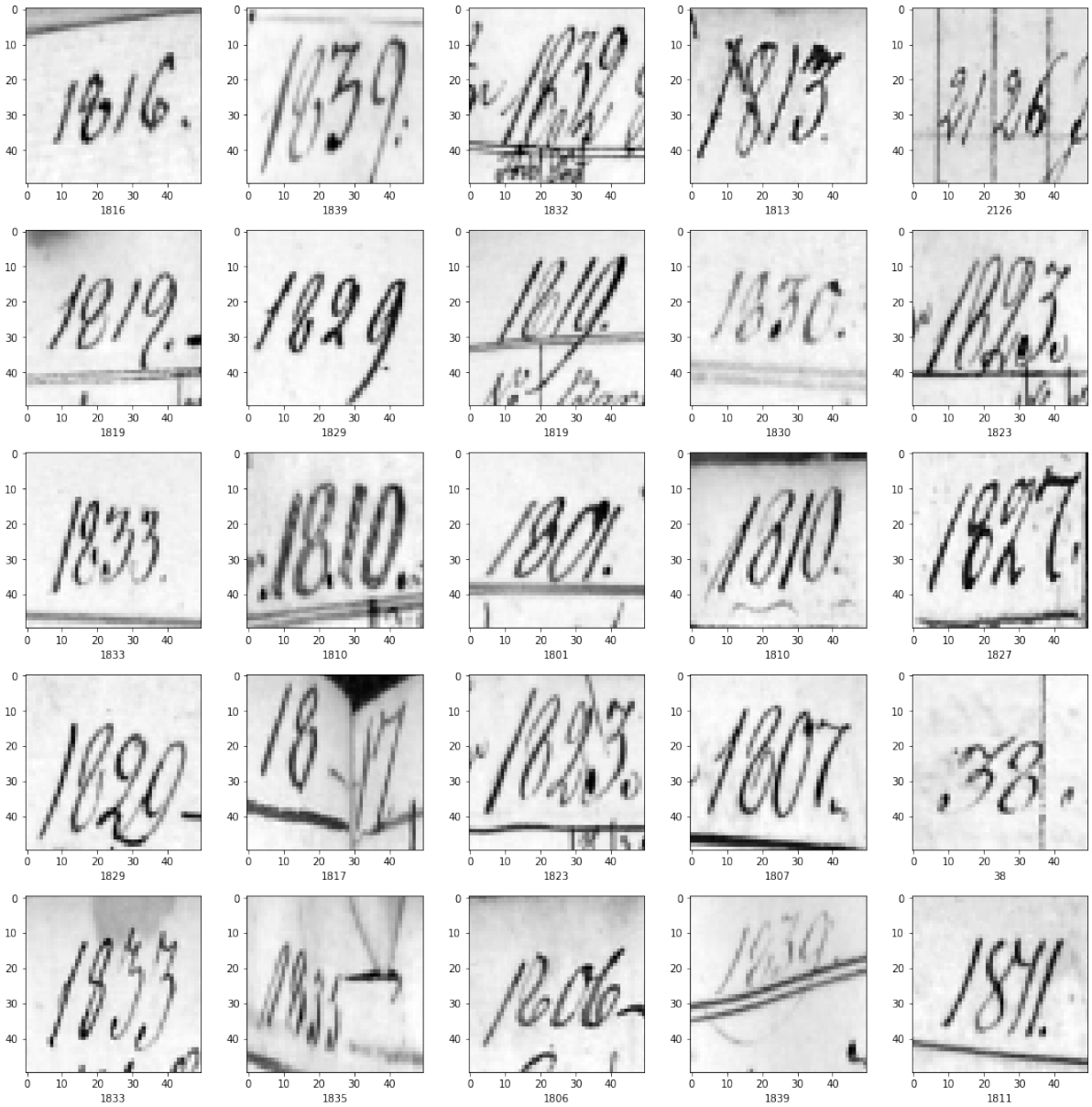


Figure 21: A random sample of 25 of the resized images used for the non-deep learning approach.

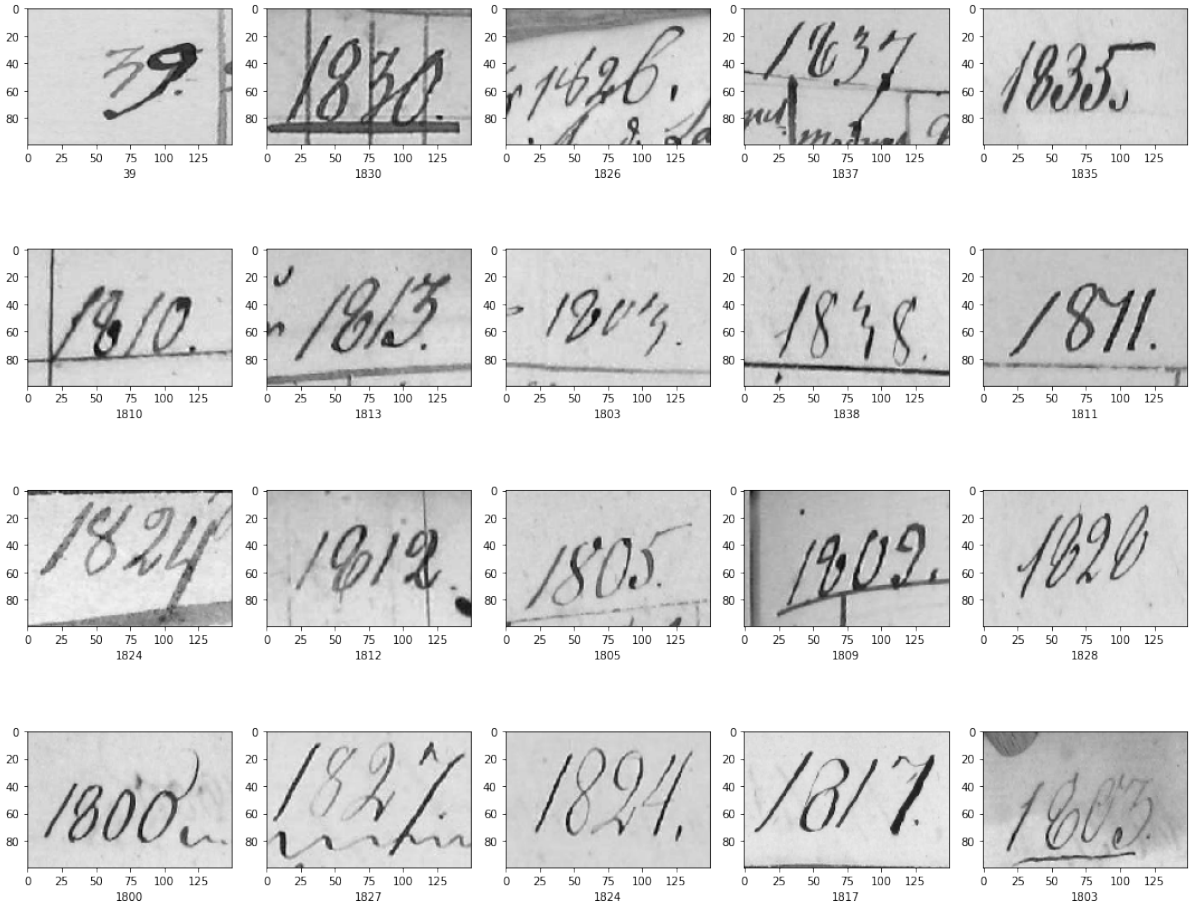


Figure 22: A random sample of 20 of the resized images used for the CNN models.

B Effect of Regularization on CNN

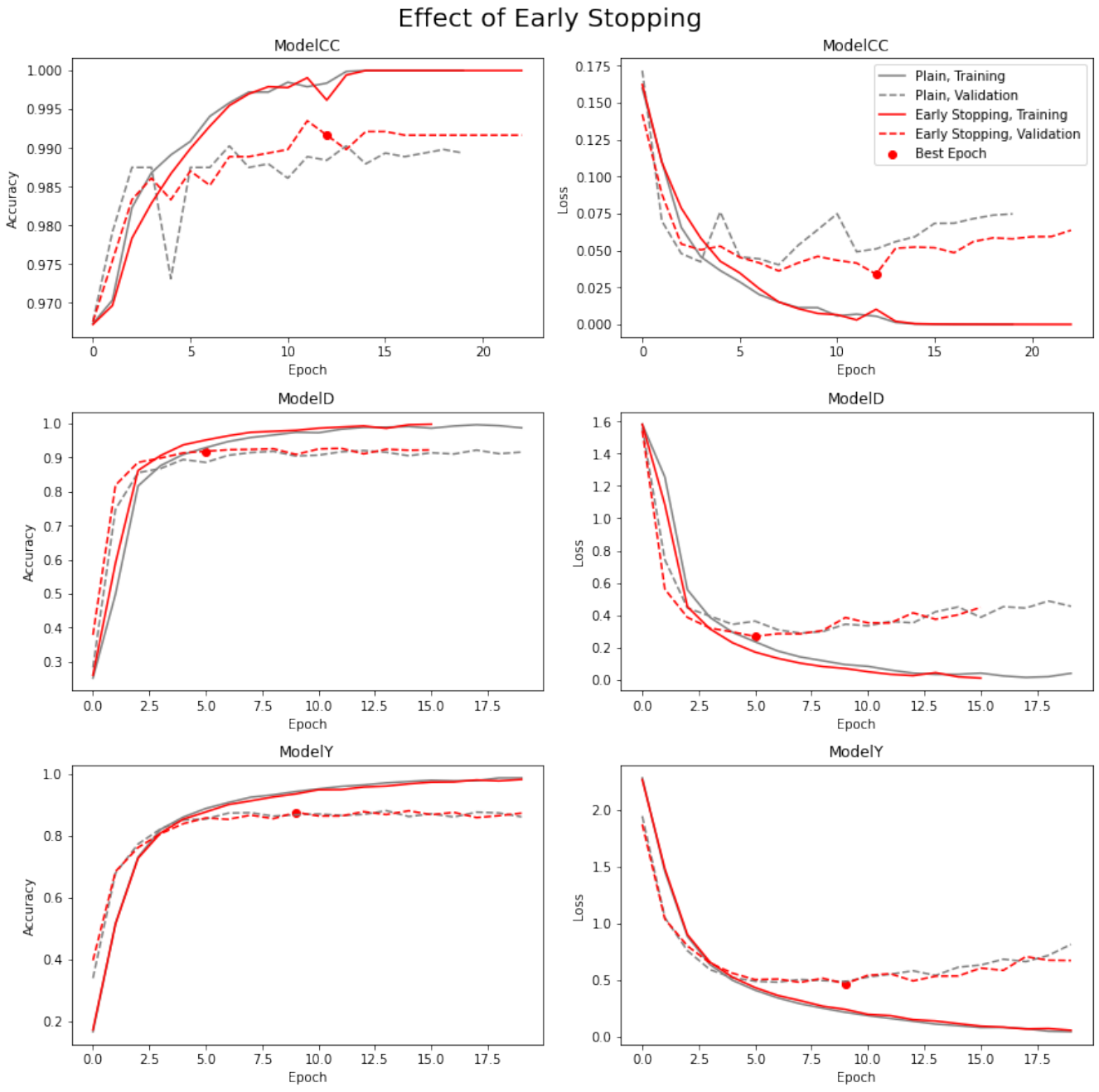


Figure 23: Effect of Early Stopping monitoring the validation loss with a patience of 10 on a simple CNN. We avoid overfitting by stopping the training, when the validation loss starts to increase. The best weights seen during training are restored.

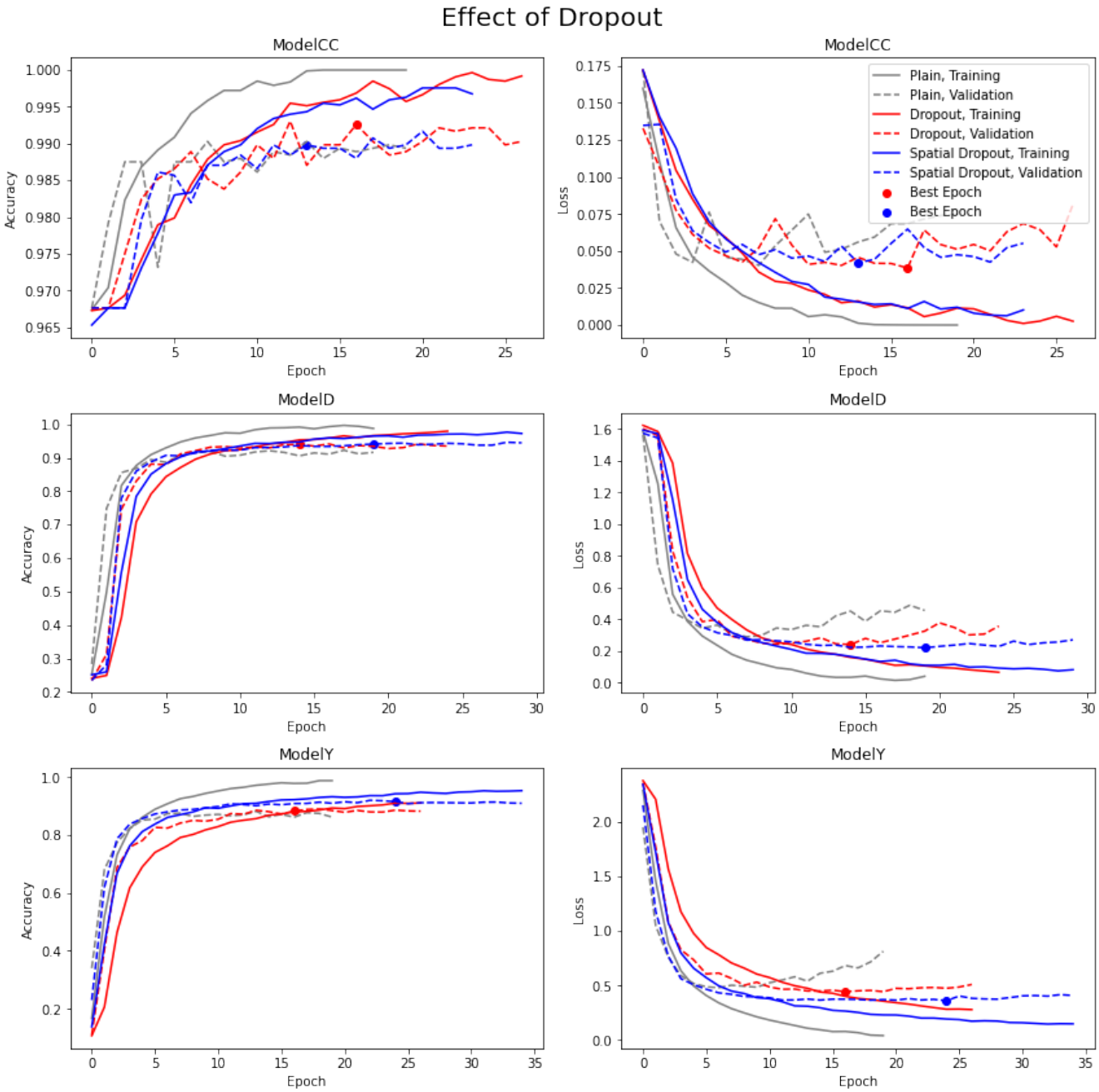


Figure 24: Effect of Dropout and Spatial Dropout on a simple CNN. Both types reduce overfitting, but increase the training time.

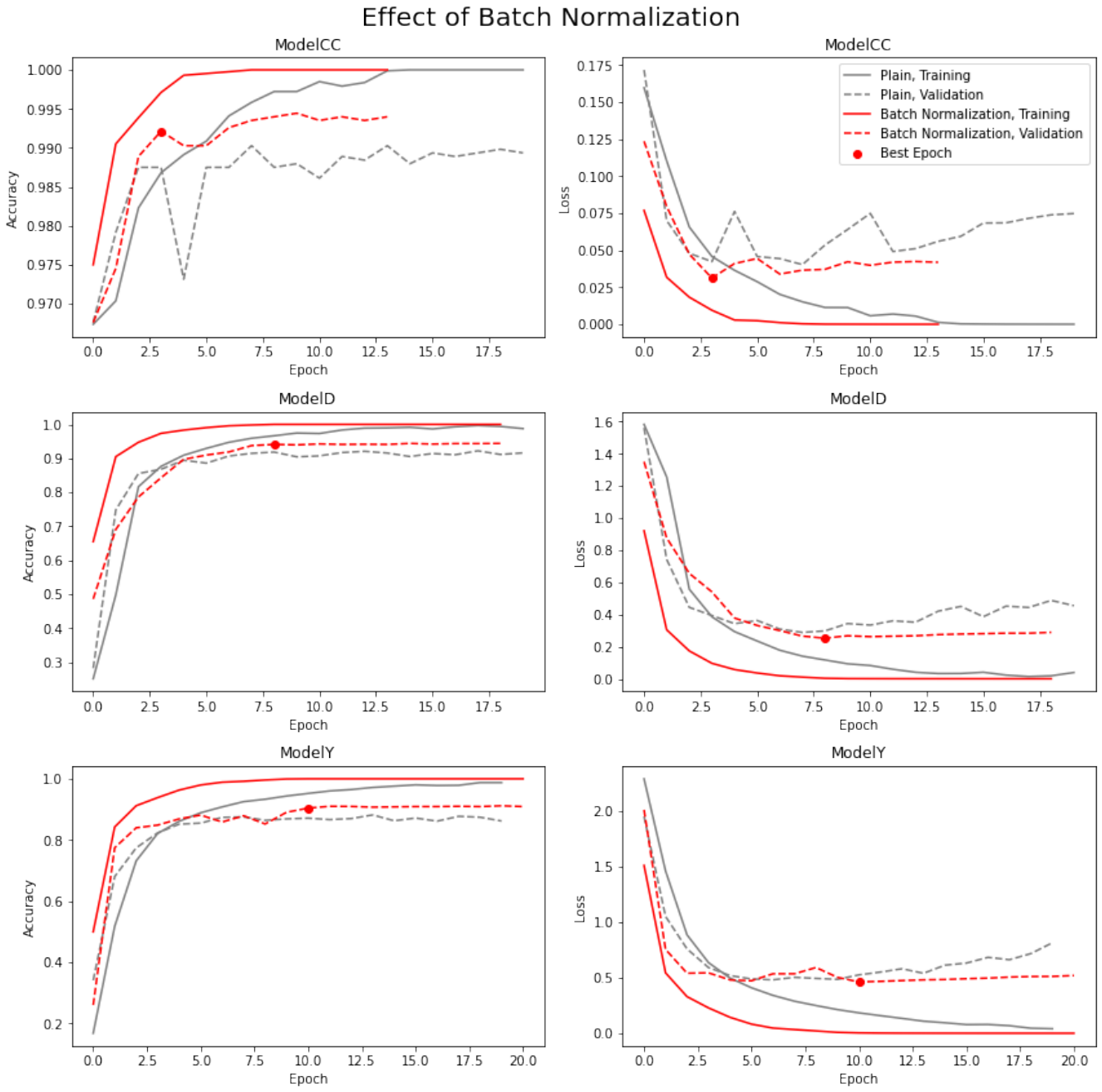


Figure 25: Effect of Batch Normalization on a simple CNN. We see faster and more robust convergence (fewer sudden spikes in the validation loss).

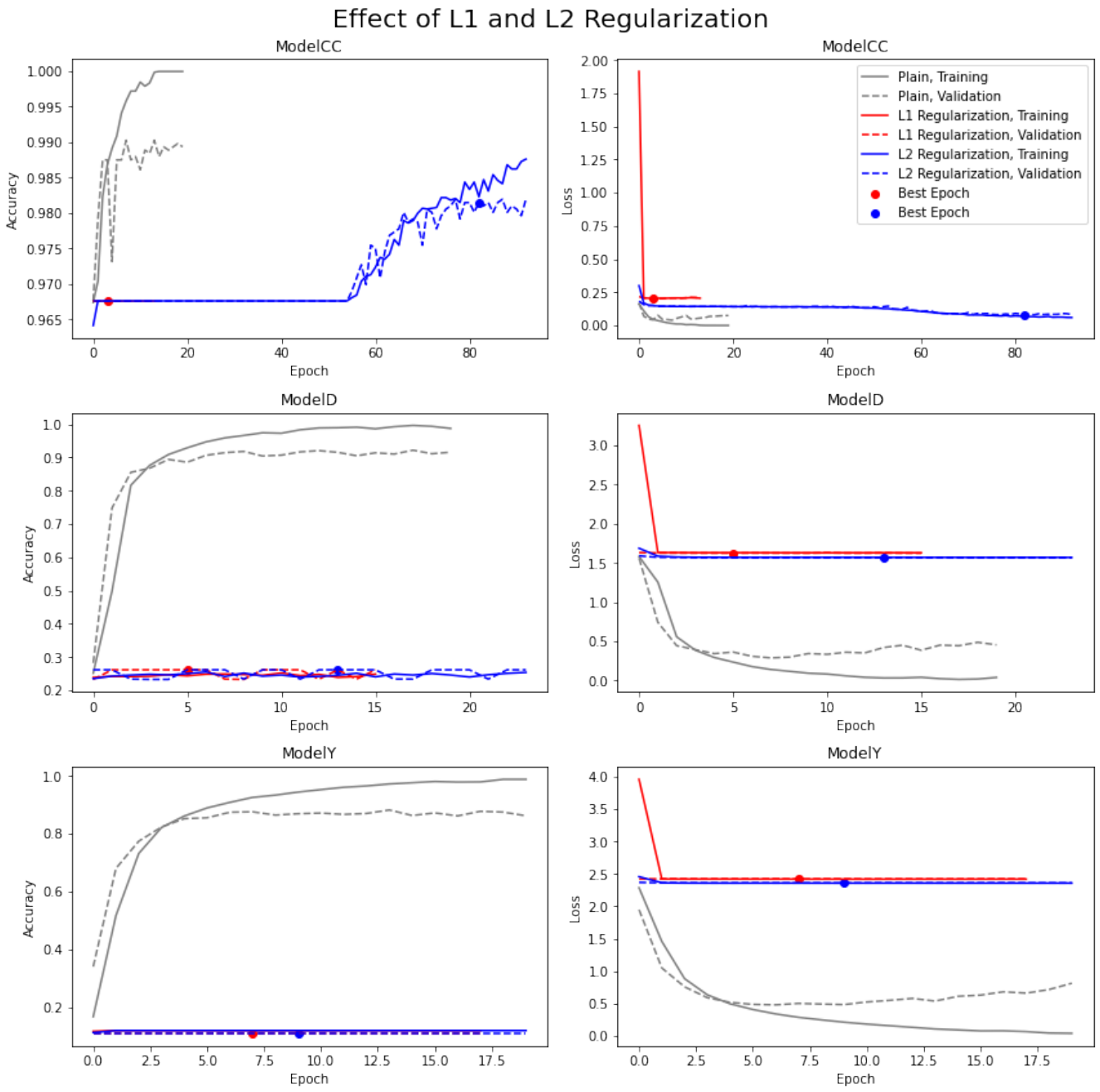


Figure 26: Effect of L1 and L2 Regularization on a simple CNN. Clearly this is too much regularization and the models loose their ability to learn. For ModelCC after approximately 55 epochs some learning occur, but with a worse result than without regularization.

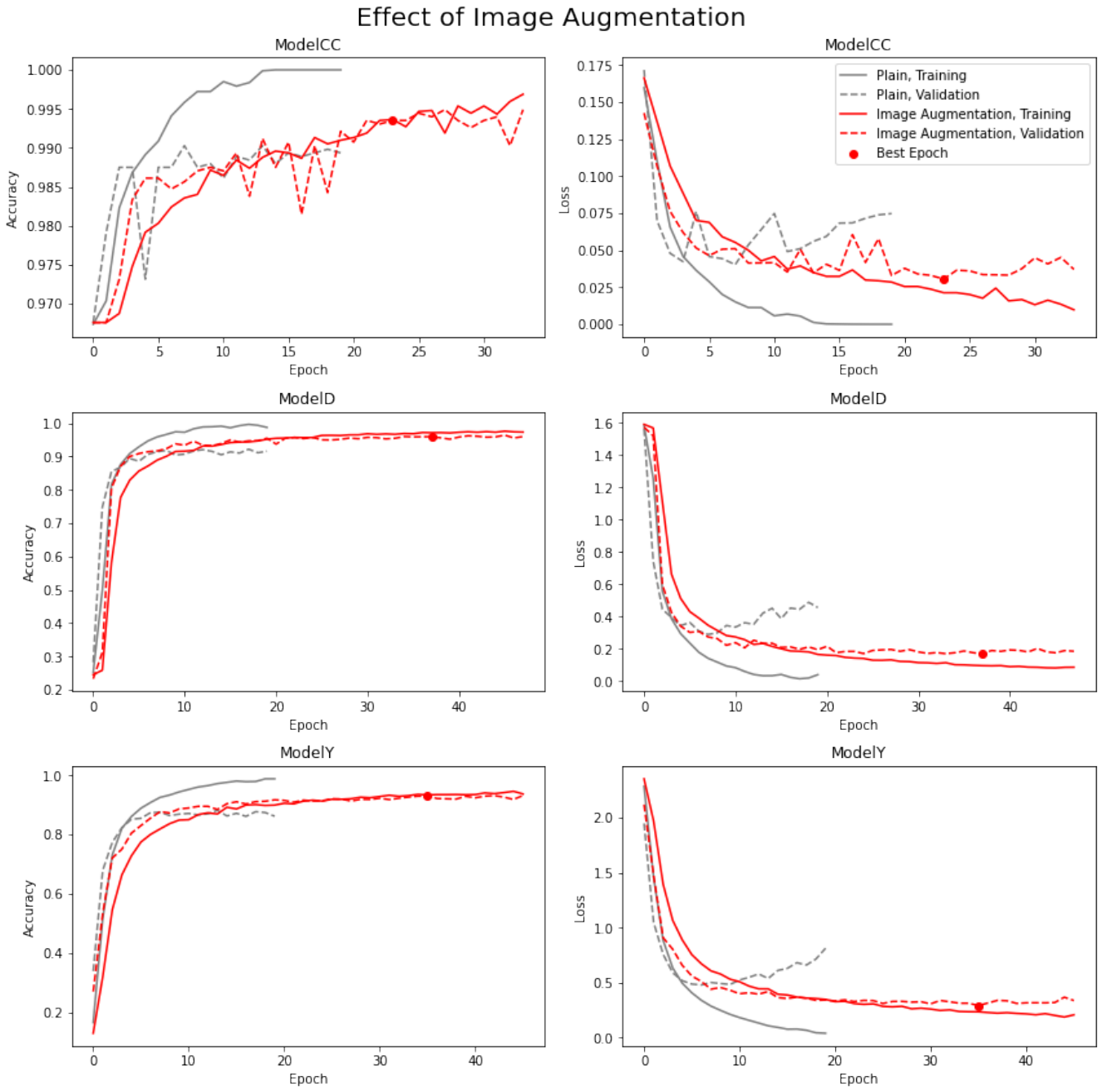


Figure 27: Effect of Image Augmentation on a simple CNN. This greatly reduces overfitting and boosts accuracy. The required training time to reach convergence is increased.

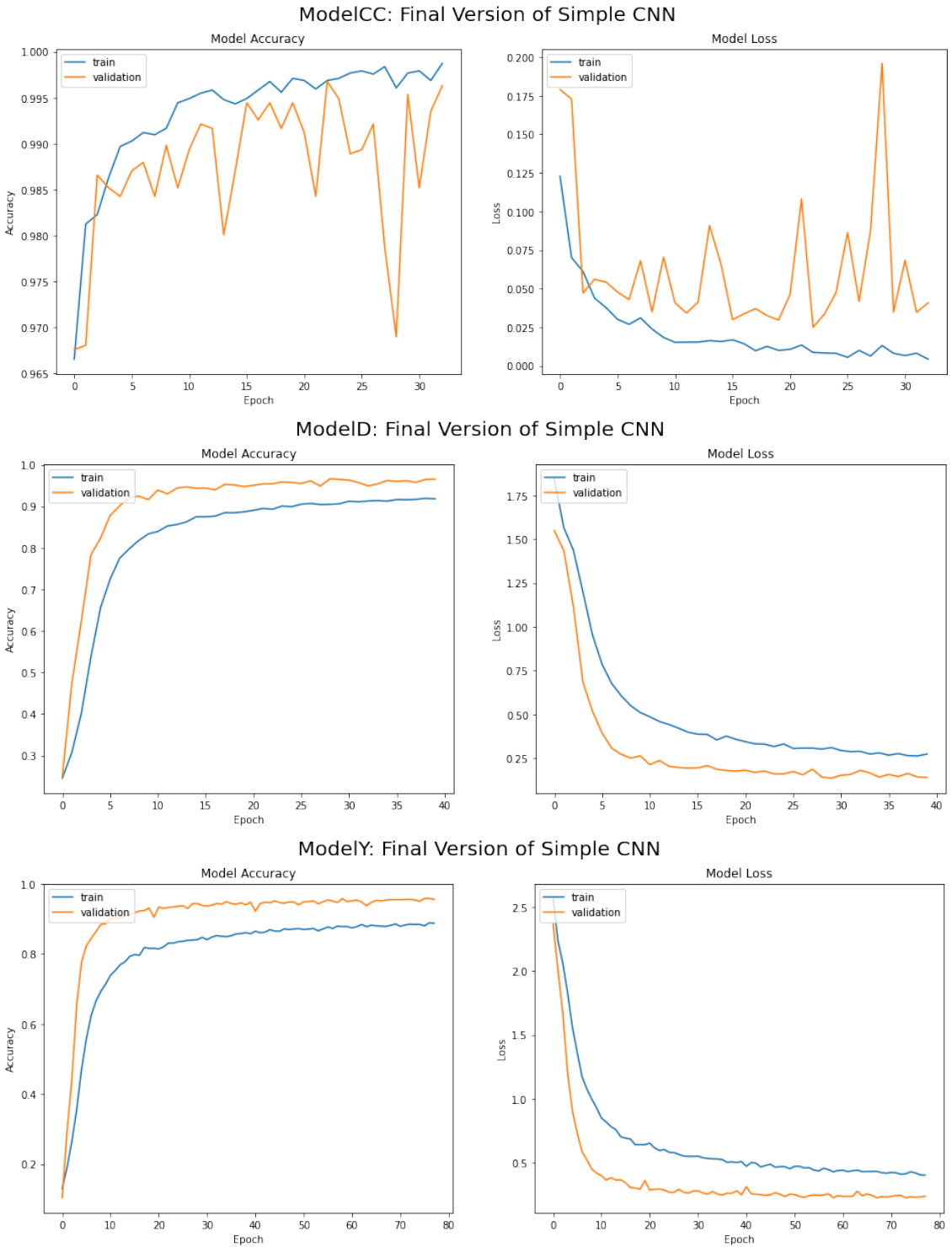


Figure 28: Final training run of the three simpel CNN models. Note that the validation accuracy is higher than the training accuracy indicating that perhaps too much image augmentation and dropout is applied.

C Architecture of Simple CNN

```
Model: "modelY_no_regularization"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_16 (Conv2D)           (None, 97, 147, 32)       544
_________________________________________________________________
max_pooling2d_16 (MaxPooling (None, 32, 49, 32)        0
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 29, 46, 32)        16416
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 9, 15, 32)         0
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 6, 12, 64)         32832
_________________________________________________________________
max_pooling2d_18 (MaxPooling (None, 2, 4, 64)          0
_________________________________________________________________
flatten_5 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_10 (Dense)             (None, 64)                32832
_________________________________________________________________
dense_11 (Dense)             (None, 11)                715
=================================================================
Total params: 83,339
Trainable params: 83,339
Non-trainable params: 0
_________________________________________________________________
```

```
Model: "modelY_with_regularization"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_5 (Sequential)    (None, 100, 150, 1)       0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 97, 147, 32)       544
_________________________________________________________________
spatial_dropout2d_3 (Spatial (None, 97, 147, 32)       0
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 32, 49, 32)        0
_________________________________________________________________
batch_normalization_6 (Batch (None, 32, 49, 32)        128
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 29, 46, 32)        16416
_________________________________________________________________
spatial_dropout2d_4 (Spatial (None, 29, 46, 32)        0
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 9, 15, 32)         0
_________________________________________________________________
batch_normalization_7 (Batch (None, 9, 15, 32)         128
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 6, 12, 64)         32832
_________________________________________________________________
spatial_dropout2d_5 (Spatial (None, 6, 12, 64)         0
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 2, 4, 64)          0
_________________________________________________________________
batch_normalization_8 (Batch (None, 2, 4, 64)          256
_________________________________________________________________
flatten_4 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_8 (Dense)              (None, 64)                32832
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
```

```
_________________________________________________________________
dense_9 (Dense)              (None, 11)                715
=================================================================
Total params: 83,851
Trainable params: 83,595
Non-trainable params: 256
_________________________________________________________________
```

D Architecture of ResNet inspired CNN

Below is the summary of the ResNet inspired model used for the Y problem. Aside from the size of the output both the CC and D models have identical architecture. This architecture is also illustrated in figure 10 on page 17.

```
    Model: "model_6"
_________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
=================================================================================================
input_7 (InputLayer)            [(None, 100, 150, 1) 0
_________________________________________________________________________________________________
sequential_6 (Sequential)       (None, 100, 150, 1)  0           input_7[0][0]
_________________________________________________________________________________________________
conv2d_96 (Conv2D)              (None, 98, 148, 32)  288         sequential_6[0][0]
_________________________________________________________________________________________________
batch_normalization_120 (BatchN (None, 98, 148, 32)  128         conv2d_96[0][0]
_________________________________________________________________________________________________
spatial_dropout2d_120 (SpatialD (None, 98, 148, 32)  0           batch_normalization_120[0][0]
_________________________________________________________________________________________________
conv2d_97 (Conv2D)              (None, 98, 148, 32)  9216        spatial_dropout2d_120[0][0]
_________________________________________________________________________________________________
batch_normalization_121 (BatchN (None, 98, 148, 32)  128         conv2d_97[0][0]
_________________________________________________________________________________________________
spatial_dropout2d_121 (SpatialD (None, 98, 148, 32)  0           batch_normalization_121[0][0]
_________________________________________________________________________________________________
conv2d_98 (Conv2D)              (None, 98, 148, 32)  9216        spatial_dropout2d_121[0][0]
_________________________________________________________________________________________________
batch_normalization_122 (BatchN (None, 98, 148, 32)  128         conv2d_98[0][0]
_________________________________________________________________________________________________
spatial_dropout2d_122 (SpatialD (None, 98, 148, 32)  0           batch_normalization_122[0][0]
_________________________________________________________________________________________________
conv2d_99 (Conv2D)              (None, 98, 148, 32)  9216        spatial_dropout2d_122[0][0]
_________________________________________________________________________________________________
batch_normalization_123 (BatchN (None, 98, 148, 32)  128         conv2d_99[0][0]
_________________________________________________________________________________________________
spatial_dropout2d_123 (SpatialD (None, 98, 148, 32)  0           batch_normalization_123[0][0]
```

```
____________________________________________________________________________________________________
add_24 (Add)                    (None, 98, 148, 32)  0           spatial_dropout2d_123[0][0]
                                                                 conv2d_96[0][0]
____________________________________________________________________________________________________
batch_normalization_124 (BatchN (None, 98, 148, 32)  128         add_24[0][0]
____________________________________________________________________________________________________
spatial_dropout2d_124 (SpatialD (None, 98, 148, 32)  0           batch_normalization_124[0][0]
____________________________________________________________________________________________________
max_pooling2d_24 (MaxPooling2D) (None, 49, 74, 32)   0           spatial_dropout2d_124[0][0]
____________________________________________________________________________________________________
conv2d_100 (Conv2D)             (None, 47, 72, 64)   18432       max_pooling2d_24[0][0]
____________________________________________________________________________________________________
batch_normalization_125 (BatchN (None, 47, 72, 64)   256         conv2d_100[0][0]
____________________________________________________________________________________________________
spatial_dropout2d_125 (SpatialD (None, 47, 72, 64)   0           batch_normalization_125[0][0]
____________________________________________________________________________________________________
conv2d_101 (Conv2D)             (None, 47, 72, 64)   36864       spatial_dropout2d_125[0][0]
____________________________________________________________________________________________________
batch_normalization_126 (BatchN (None, 47, 72, 64)   256         conv2d_101[0][0]
____________________________________________________________________________________________________
spatial_dropout2d_126 (SpatialD (None, 47, 72, 64)   0           batch_normalization_126[0][0]
____________________________________________________________________________________________________
conv2d_102 (Conv2D)             (None, 47, 72, 64)   36864       spatial_dropout2d_126[0][0]
____________________________________________________________________________________________________
batch_normalization_127 (BatchN (None, 47, 72, 64)   256         conv2d_102[0][0]
____________________________________________________________________________________________________
spatial_dropout2d_127 (SpatialD (None, 47, 72, 64)   0           batch_normalization_127[0][0]
____________________________________________________________________________________________________
conv2d_103 (Conv2D)             (None, 47, 72, 64)   36864       spatial_dropout2d_127[0][0]
____________________________________________________________________________________________________
batch_normalization_128 (BatchN (None, 47, 72, 64)   256         conv2d_103[0][0]
____________________________________________________________________________________________________
spatial_dropout2d_128 (SpatialD (None, 47, 72, 64)   0           batch_normalization_128[0][0]
____________________________________________________________________________________________________
add_25 (Add)                    (None, 47, 72, 64)   0           spatial_dropout2d_128[0][0]
```

```
                                                                    conv2d_100[0][0]
__________________________________________________________________________________________________
batch_normalization_129 (BatchN (None, 47, 72, 64)   256         add_25[0][0]
__________________________________________________________________________________________________
spatial_dropout2d_129 (SpatialD (None, 47, 72, 64)   0           batch_normalization_129[0][0]
__________________________________________________________________________________________________
max_pooling2d_25 (MaxPooling2D) (None, 23, 36, 64)   0           spatial_dropout2d_129[0][0]
__________________________________________________________________________________________________
conv2d_104 (Conv2D)             (None, 21, 34, 64)   36864       max_pooling2d_25[0][0]
__________________________________________________________________________________________________
batch_normalization_130 (BatchN (None, 21, 34, 64)   256         conv2d_104[0][0]
__________________________________________________________________________________________________
spatial_dropout2d_130 (SpatialD (None, 21, 34, 64)   0           batch_normalization_130[0][0]
__________________________________________________________________________________________________
conv2d_105 (Conv2D)             (None, 21, 34, 64)   36864       spatial_dropout2d_130[0][0]
__________________________________________________________________________________________________
batch_normalization_131 (BatchN (None, 21, 34, 64)   256         conv2d_105[0][0]
__________________________________________________________________________________________________
spatial_dropout2d_131 (SpatialD (None, 21, 34, 64)   0           batch_normalization_131[0][0]
__________________________________________________________________________________________________
conv2d_106 (Conv2D)             (None, 21, 34, 64)   36864       spatial_dropout2d_131[0][0]
__________________________________________________________________________________________________
batch_normalization_132 (BatchN (None, 21, 34, 64)   256         conv2d_106[0][0]
__________________________________________________________________________________________________
spatial_dropout2d_132 (SpatialD (None, 21, 34, 64)   0           batch_normalization_132[0][0]
__________________________________________________________________________________________________
conv2d_107 (Conv2D)             (None, 21, 34, 64)   36864       spatial_dropout2d_132[0][0]
__________________________________________________________________________________________________
batch_normalization_133 (BatchN (None, 21, 34, 64)   256         conv2d_107[0][0]
__________________________________________________________________________________________________
spatial_dropout2d_133 (SpatialD (None, 21, 34, 64)   0           batch_normalization_133[0][0]
__________________________________________________________________________________________________
add_26 (Add)                    (None, 21, 34, 64)   0           spatial_dropout2d_133[0][0]
                                                                    conv2d_104[0][0]
__________________________________________________________________________________________________
```

```
batch_normalization_134 (BatchN (None, 21, 34, 64)   256         add_26[0][0]
____________________________________________________________________________________________________
spatial_dropout2d_134 (SpatialD (None, 21, 34, 64)   0           batch_normalization_134[0][0]
____________________________________________________________________________________________________
max_pooling2d_26 (MaxPooling2D) (None, 10, 17, 64)   0           spatial_dropout2d_134[0][0]
____________________________________________________________________________________________________
conv2d_108 (Conv2D)             (None, 8, 15, 128)   73728       max_pooling2d_26[0][0]
____________________________________________________________________________________________________
batch_normalization_135 (BatchN (None, 8, 15, 128)   512         conv2d_108[0][0]
____________________________________________________________________________________________________
spatial_dropout2d_135 (SpatialD (None, 8, 15, 128)   0           batch_normalization_135[0][0]
____________________________________________________________________________________________________
conv2d_109 (Conv2D)             (None, 8, 15, 128)   147456      spatial_dropout2d_135[0][0]
____________________________________________________________________________________________________
batch_normalization_136 (BatchN (None, 8, 15, 128)   512         conv2d_109[0][0]
____________________________________________________________________________________________________
spatial_dropout2d_136 (SpatialD (None, 8, 15, 128)   0           batch_normalization_136[0][0]
____________________________________________________________________________________________________
conv2d_110 (Conv2D)             (None, 8, 15, 128)   147456      spatial_dropout2d_136[0][0]
____________________________________________________________________________________________________
batch_normalization_137 (BatchN (None, 8, 15, 128)   512         conv2d_110[0][0]
____________________________________________________________________________________________________
spatial_dropout2d_137 (SpatialD (None, 8, 15, 128)   0           batch_normalization_137[0][0]
____________________________________________________________________________________________________
conv2d_111 (Conv2D)             (None, 8, 15, 128)   147456      spatial_dropout2d_137[0][0]
____________________________________________________________________________________________________
batch_normalization_138 (BatchN (None, 8, 15, 128)   512         conv2d_111[0][0]
____________________________________________________________________________________________________
spatial_dropout2d_138 (SpatialD (None, 8, 15, 128)   0           batch_normalization_138[0][0]
____________________________________________________________________________________________________
add_27 (Add)                    (None, 8, 15, 128)   0           spatial_dropout2d_138[0][0]
                                                                 conv2d_108[0][0]
____________________________________________________________________________________________________
batch_normalization_139 (BatchN (None, 8, 15, 128)   512         add_27[0][0]
____________________________________________________________________________________________________
```

```
spatial_dropout2d_139 (SpatialD (None, 8, 15, 128)   0          batch_normalization_139[0][0]
_____________________________________________________________________________________________
max_pooling2d_27 (MaxPooling2D) (None, 4, 7, 128)    0          spatial_dropout2d_139[0][0]
_____________________________________________________________________________________________
flatten_6 (Flatten)             (None, 3584)         0          max_pooling2d_27[0][0]
_____________________________________________________________________________________________
dense_12 (Dense)                (None, 128)          458880     flatten_6[0][0]
_____________________________________________________________________________________________
dropout_6 (Dropout)             (None, 128)          0          dense_12[0][0]
_____________________________________________________________________________________________
dense_13 (Dense)                (None, 11)           1419       dropout_6[0][0]
=============================================================================================
Total params: 1,286,571
Trainable params: 1,283,691
Non-trainable params: 2,880
_____________________________________________________________________________________________
```

E Transfer Learning: Learning Curves

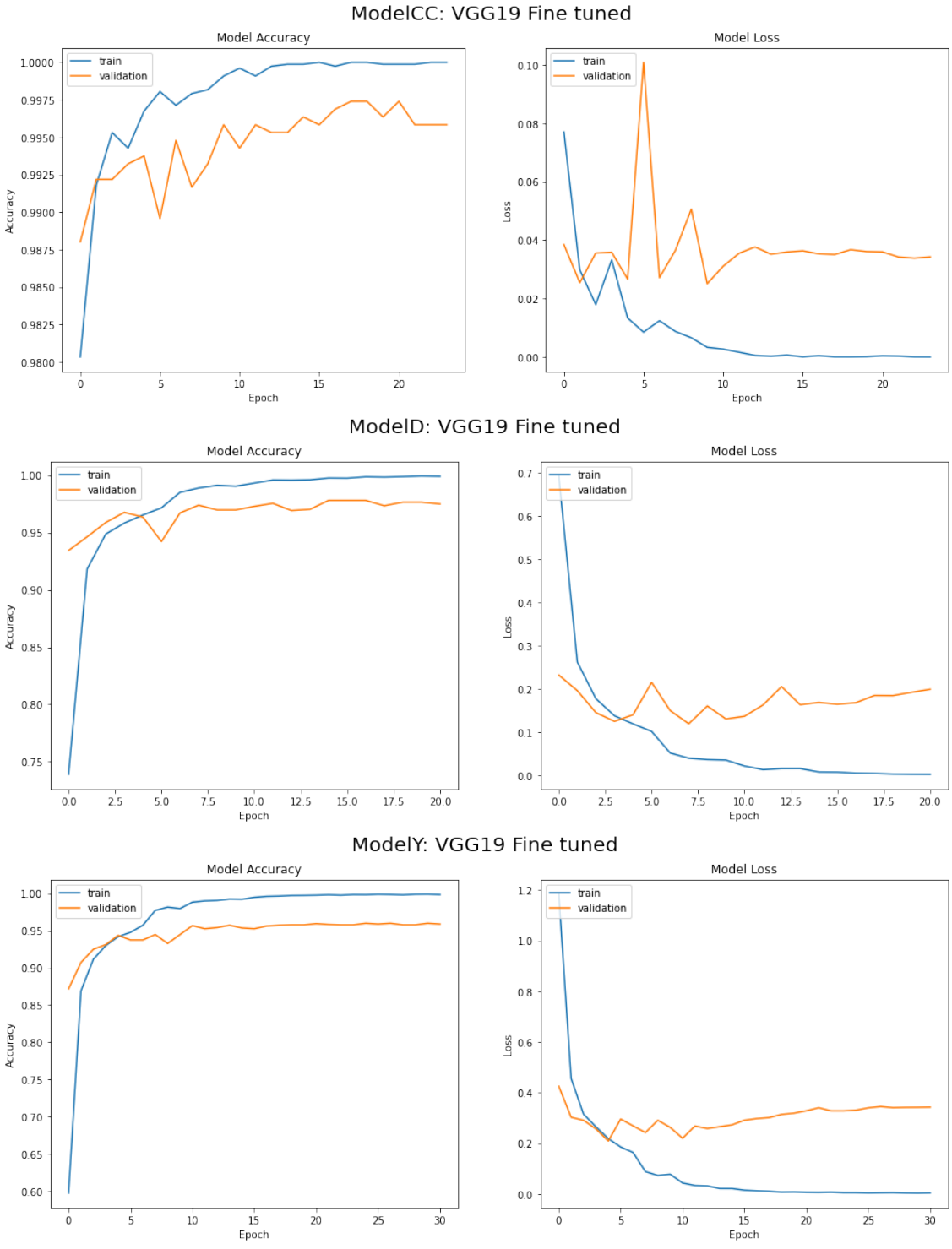


Figure 29: Final training run of the three models based on pretrained VGG19.

F Model structure of pretrained model VGG19

```
Model: "vgg19"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 100, 150, 3)]     0
 block1_conv1 (Conv2D)       (None, 100, 150, 64)      1792
 block1_conv2 (Conv2D)       (None, 100, 150, 64)      36928
 block1_pool (MaxPooling2D)  (None, 50, 75, 64)        0

 block2_conv1 (Conv2D)       (None, 50, 75, 128)       73856
 block2_conv2 (Conv2D)       (None, 50, 75, 128)       147584
 block2_pool (MaxPooling2D)  (None, 25, 37, 128)       0

 block3_conv1 (Conv2D)       (None, 25, 37, 256)       295168
 block3_conv2 (Conv2D)       (None, 25, 37, 256)       590080
 block3_conv3 (Conv2D)       (None, 25, 37, 256)       590080
 block3_conv4 (Conv2D)       (None, 25, 37, 256)       590080

 block3_pool (MaxPooling2D)  (None, 12, 18, 256)       0
 block4_conv1 (Conv2D)       (None, 12, 18, 512)       1180160
 block4_conv2 (Conv2D)       (None, 12, 18, 512)       2359808
 block4_conv3 (Conv2D)       (None, 12, 18, 512)       2359808
 block4_conv4 (Conv2D)       (None, 12, 18, 512)       2359808
 block4_pool (MaxPooling2D)  (None, 6, 9, 512)         0

 block5_conv1 (Conv2D)       (None, 6, 9, 512)         2359808
 block5_conv2 (Conv2D)       (None, 6, 9, 512)         2359808
 block5_conv3 (Conv2D)       (None, 6, 9, 512)         2359808
 block5_conv4 (Conv2D)       (None, 6, 9, 512)         2359808
 block5_pool (MaxPooling2D)  (None, 3, 4, 512)         0
=================================================================
Total params: 20,024,384
Trainable params: 9,439,232
```

```
Non-trainable params: 10,585,152

_________________________________________________________________
-


Model: "Fine tuned VGG19 model with new dense classifier top"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential (Sequential)     (None, None, None, 3)     0

 vgg19 (Functional)          (None, 3, 4, 512)         20024384

 flatten_3 (Flatten)         (None, 512)               0
 dense_6 (Dense)             (None, 1024)              525312
 dropout_3 (Dropout)         (None, 1024)              0
 dense_7 (Dense)             (None, 6)                 6150

=================================================================
Total params: 20,555,846
Trainable params: 9,970,694
Non-trainable params: 10,585,152
_________________________________________________________________
```

G Effect of Image Augmentation

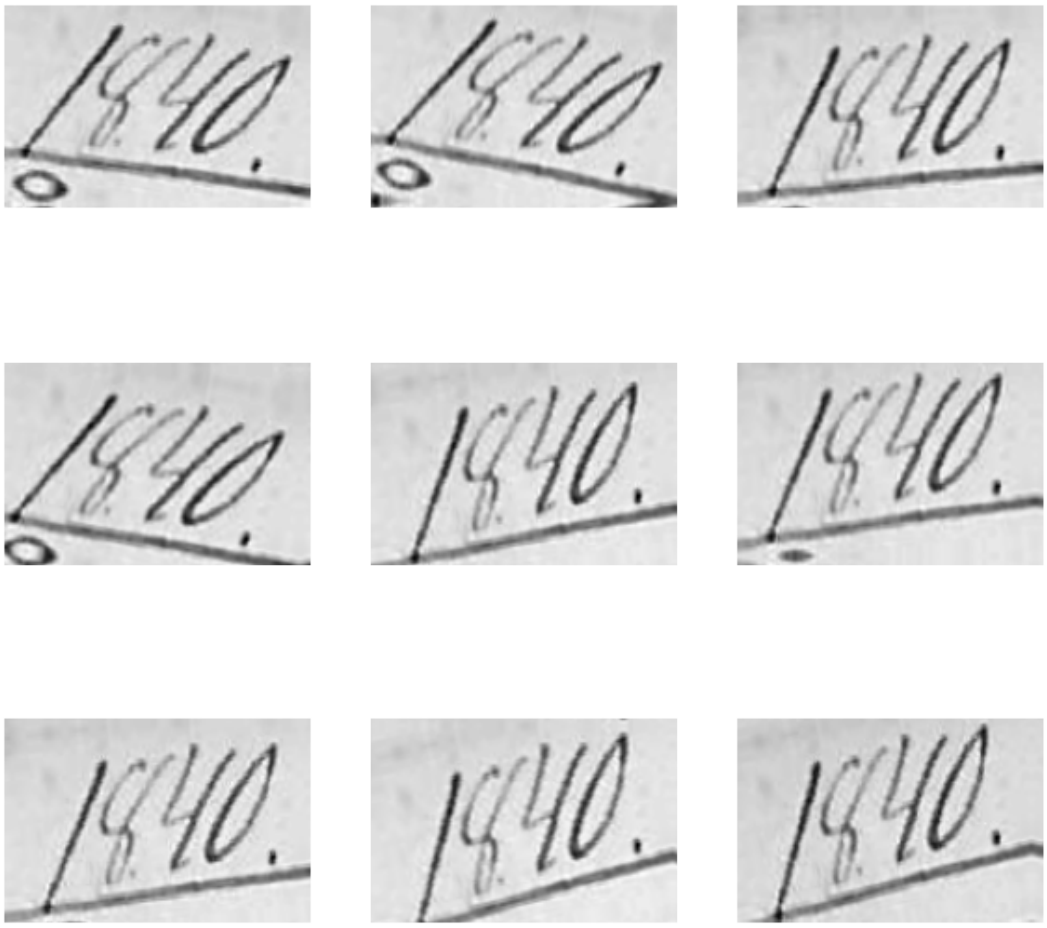


Figure 30: Example of how the chosen Image Augmentation effects a randomly chosen image. The images still resemble handwritten digits. We have used the settings: `RandomRotation(0.04)`, `RandomTranslation(width_factor=0.05, height_factor=0.05)` and `RandomContrast(0.1)`.

H Misclassified Images by the Final Model

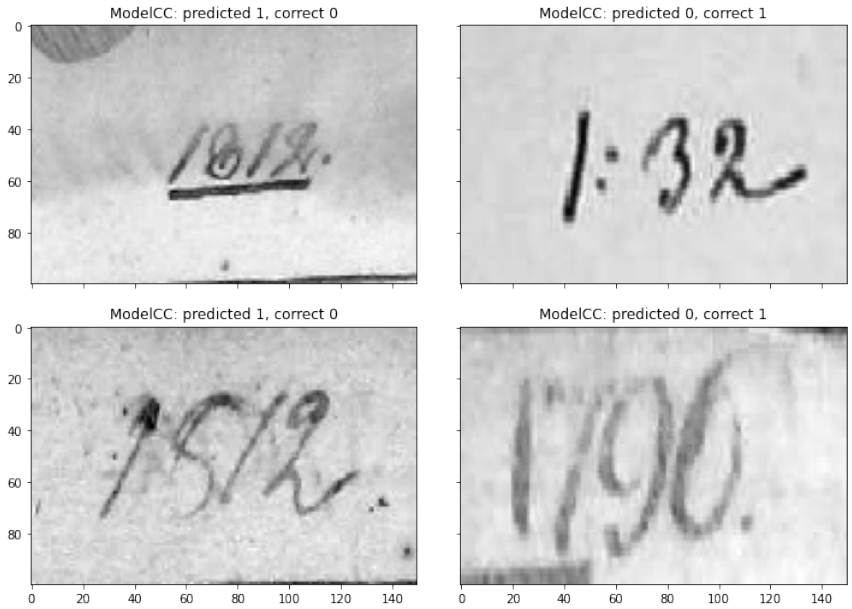


Figure 31: ModelCC misclassifies 4 out of the 1200 test images.

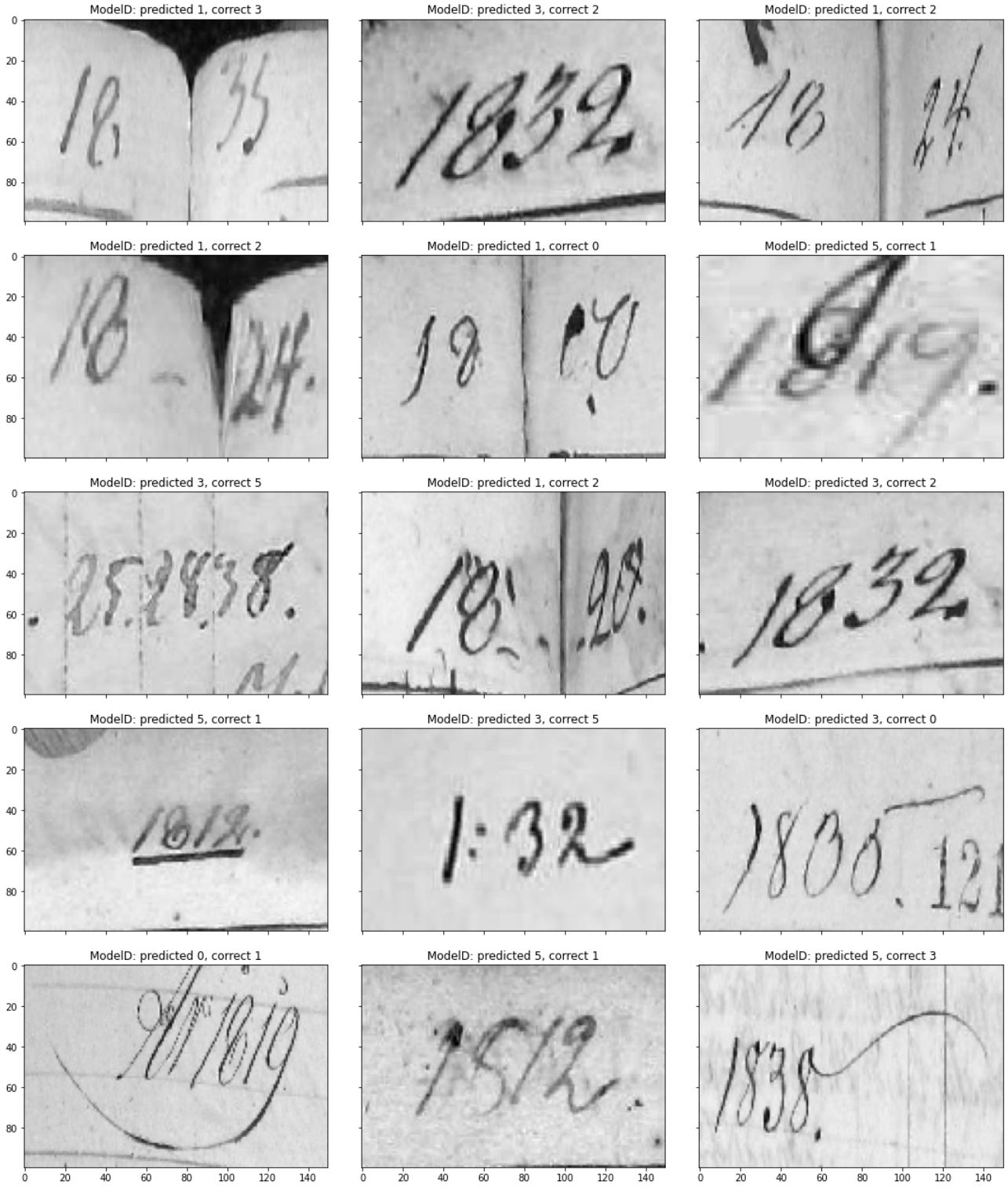


Figure 32: ModelD misclassifies 15 out of the 1200 test images.

Figure 33: Model Y misclassifies 26 out of the 1200 test images.