Recognizing handwritten digits
INF283 - Project - 2

Sindre Eik de Lange

Executive Summary

The task was described as to produce a classifier able to predict labels of handwritten digits - a step in the direction of developing an AI system to automatically deliver mail.

Generally, when training such classifiers one needs relevant data. Such a dataset is provided by the company, consisting of approximately 70k images (handwritten digits), and their labels. Naturally, this dataset is used for training of our classifier.

As mentioned this dataset consists of approximately 70k images. In order to verify that the classifiers are general, that is, it is able to recognize unseen data, the data set was divided into three unique sets: *training*, *validation* and *testing*. Specifically, this makes it possible to optimize the classifier, using *training*- and *validation*-set, and then verify the aforementioned level of generality using the (unseen) *testing*-set.

Four classifiers has been implemented, with some what various results ranging from approximately 88-99%. It is worth mentioning here that obtaining 100% accuracy is not desirable. This assertion is based on information from http://www.codeproject.com/KB/library/NeuralNetRecognition/Collage-74-Errors.gif, where one can clearly see some faults in the data.

With the advances in the field of machine learning this task is relevant for using such techniques. Especially image classification has seen an increase in performance (https://medium.com/comet-app/review-of-deep-learning-algorithms-for-image-classification-5fdbca4a05e2). One big factor is that it is such a big domain, with many big corporations standing to gain from this increase, and the mass production of GPUs.

As a last note, for additional improvement of the classifiers - making them even better for our specific task, it is relevant to start creating our own dataset, consisting of examples from our system.

Technical Report

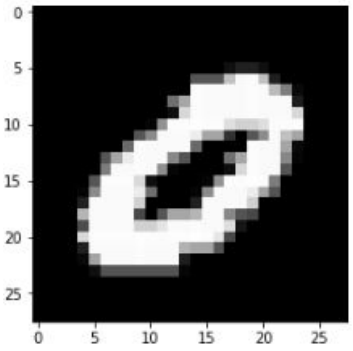
The task was described as to produce a classifier able to predict labels of handwritten digits - a step in the direction of developing an AI system to automatically deliver mail.

The data was supplied by the company - containing 70k images of handwritten images [0, 9], stored in two .csv files: *handwritten_digits_images.csv* and *handwritten_digits_labels.csv*. These were too big for GitHub, but as mentioned in the README.md file, can be downloaded from https://pjreddie.com/projects/mnist-in-csv/.

In order to verify that the classifiers are general, that is, it is able to recognize unseen data, the data set was divided into three unique sets: *training*, *validation* and *testing*. This was done explicitly for the deep learning models, and implicitly for the machine learning models. Specifically, this makes it possible to optimize the classifier, e.g. do hyperparameter tuning, using *training*- and *validation*-set, and then verify the aforementioned level of generality using the (unseen) *testing*-set.

More specifically, the training- and validation set is generally used for hyperparameter tuning, and model selection, then the training-set to get information how well the model will perform when deployed.

A computer sees an image as a combination of pixel values [0, 256], which is why one can store images as a *.csv* file. So while we humans see the images like this:



The computer sees this:

```
[0.        , 0.        , 0.        , 0.        , 0.        ,
 0.        , 0.        , 0.        , 0.        , 0.        ,
 0.        , 0.        , 0.        , 0.        , 0.        ,
 0.        , 0.        , 0.11372549, 0.16862746, 0.16862746,
 0.10980392, 0.        , 0.        , 0.        , 0.        ,
 0.        , 0.        , 0.        ],
[0.        , 0.        , 0.        , 0.        , 0.        ,
 0.        , 0.        , 0.        , 0.        , 0.        ,
 0.        , 0.        , 0.        , 0.        , 0.33333334,
 0.33333334, 0.33333334, 0.76862746, 0.98039216, 0.9882353 ,
 0.7607843 , 0.16470589, 0.        , 0.        , 0.        ,
 0.        , 0.        , 0.        ],
[0.        , 0.        , 0.        , 0.        , 0.        ,
 0.        , 0.        , 0.        , 0.        , 0.        ,
 0.        , 0.        , 0.        , 0.        , 0.9882353 ,
 0.98039216, 0.98039216, 0.98039216, 0.98039216, 0.9882353 ,
```

Note: A small snippet of the data that gives the number, above.

Worth mentioning that some algorithms expects the data to be reshaped into 2- or 3D arrays - more about this later.

Accuracy on the validation set was used for performance metric because this is a straight forward “right or wrong” classification case, where one does not generally care about anything other than optimizing the models ability to classify an image correct. With that said, a *confusion matrix* provides information about which numbers the model has problems with recognizing, and which numbers it most often confuses for other specific numbers, e.g. 1 and 7. Considering this, a confusion matrix were plotted, but only for the chosen optimal model.

For your information, a portion of the code has been refactored into different *.py* files in an attempt to prettify the notebook, and make everything more readable. It is important to note that these files is predicated on the data-files being stored in the *data/*-folder, images and labels, respectively.

Preprocessing & Candidate Algorithms

I have decided to combine these two points, because what algorithm one chooses dictates the necessary/relevant preprocessing steps, so that one is able to run the algorithms.

However, there were chosen 4 candidates models:

1. Random Forest
2. XGBoost
3. Neural Network
4. Convolutional Neural Network

which means that two *classes* of classifiers were chosen: *machine learning*- and *deep learning* classifiers. This results in (somewhat) different preprocessing steps.

Generally, neural networks work better on image classification: https://towardsdatascience.com/deep-learning-vs-classical-machine-learning-9a42c6d48aa,

which is why the main focus has been on the deep learning algorithms. With that said, it is interesting to test some of the most powerful machine learning algorithms.

Other machine learning algorithms were left out based on the reason mentioned above; they are generally inferior when processing images. It would've been fun to implement more complex neural networks, such as ResNet or ResNext, but for this task it seems like overkill.

Random Forest

Random Forest is a bagging classifier for, amongst other things, classification. It combines multiple *tree classifiers* or *Decision Trees* in order to get a more accurate and stable prediction than the *Decision Trees* could get by themselves.

Random Forest is well suited for multiclass classification problems for a number of reasons, but the most important one is that it is able to calculate feature importance at every step of the training, and therefore optimizes itself when training.

(Important) Hyper parameters:

- **n_estimators**: number of trees the algorithm builds before calculating the answer, e.g. taking the maximum vote, or averages of predictions.
- **max_features**: number of features to consider when looking for the best split. NOTE: This was not set under initial training, because *Sklearn*'s default value seemed satisfactory based on their documentation (and experience).
- **random_state**: The seed used by the random number generator, e.g. how one can make sure that results are reproducible.
- **n_jobs**: number of processors it is allowed to use - affects only computational time.

In order to train a Random Forest, one needs to use *One Hot Encoding* on the *labels*, because it is a *categorical variable*. This causes an increase in the dimensionality of the feature representations, which again leads to increase in compute time. Furthermore, in order to decrease compute time one can *normalize* the *training data*, i.e. make all of the pixel values between 0 and 1. `X = X / 255` . Note that the pixel values are divided by 255 because this is the maximum value a pixel can have. One can hard code this value because this is constant for images.

Random Forest divides the training set into both training and validation, by default, which is the reason why one does not need to explicitly pass in both sets. Then the explicit validation set has been used to score the model

```
Scores:  0.8957857142857143
 Training time:  15.95  seconds
```

Model performance:
This is an ok score, considering the amount of time it took, and the fact no Grid- or Random-Search were implemented. As mentioned earlier, machine learning algorithms are generally inferior to deep learning algorithms at tasks such as image classification, which is why this was decided not to be implemented.

XGBoost

XGBoost, or *eXtreme Gradient Boosting*, is a popular tool for Kaggle Competitions, because it works on tabular data, but on image classification, as mentioned, it falls short. However, there are implementations where one combines CNNs with XGBoost, by using the CNN as a trainable feature extractor, and XGBoost as recognizer in the top level of the network which produces the results.

As I will show later, this model performed ok, accuracy-wise, however it used approximately 40 minutes longer on training than the other models. Considering this, XGBoost does not seem as a viable model for this problem, and will not be discussed further.

General Neural Network

A *General Neural Network* is a standard feed forward neural network which is fed input, that is run through the network, outputs a prediction, and then updates its weights using backpropagation. It is a powerful tool, and works good for a variety of problems, also image classification.

The model looks like this:

```
def get_nn_model(self, learning_rate=1e-3, regularizer=0.01,
                 loss='categorical_crossentropy', metrics='accuracy'):
    optimizer = self.optimizer(lr=learning_rate)
    regularizer_l2 = regularizers.l2(regularizer)
    model_nn = Sequential()
    model_nn.add(Dense(256, input_dim=784, activation='relu', kernel_regularizer=regularizer_l2))
    model_nn.add(Dense(128, activation='relu', kernel_regularizer=regularizer_l2))
    model_nn.add(Dense(60, activation='relu', kernel_regularizer=regularizer_l2))
    model_nn.add(Dense(10, activation='softmax'))
    model_nn.compile(loss=loss, optimizer=optimizer, metrics=[metrics])
    return model_nn
```

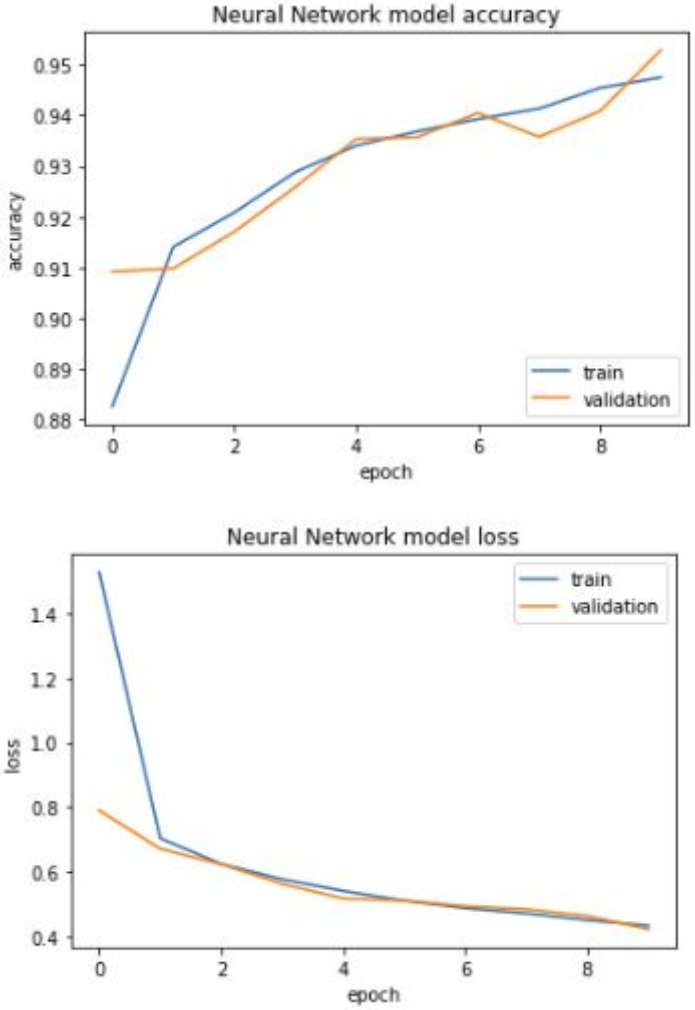
The model was built using the Keras framework, more specifically, its *Dense layer* component. A dense layer is another term for *fully connected layer* which basically means that it is a linear operation where each input is connected to every output by a weight. These are usually followed by a non-linear activation function, which in our case is a *rectified linear unit* or *ReLu*. This basically just means that each negative (pixel) value is set to 0, and the other are remained untouched. An alternative to *relu* is *tanh*, which used to be the standard, however, today most project use *relu*.

The *input_dim* is equal to the dimensionality of the input, which on this case is 28x28 = 784.

As for the number of neurons, I have yet to find some documentation on the optimal number for the hidden layers, except that is should somewhat reflect the complexity of the input. As for the last layer, the output layer, always has to equal the number of classes in the data.

The *kernel regularizer* is implemented because the model was overfitting, so by implementing a regularizer, one combats this, by applying penalties on layer activity during optimization. More specifically one regularizes the weights in the neural network when doing backpropagation. This can cause a decrease in model accuracy, but this is often a trade off when working with neural networks: accuracy vs. overfitting. Nne generally wants to avoid overfitting, even if it means sacrificing some accuracy.

Here one can see a plot of the models training



Which shows that the model converges somewhat, and is therefore a solid model for image classification. Furthermore, one can see that the model is not overfitting, it is rather underfitting, so the regularization was successful.

Convolutional Neural Network

A Convolutional Neural Network, or CNN, is a specific type of Neural Network, in which one uses *convolutions* or *filters* or *kernels* (usually 3x3, or 5x5 matrices) to slide over the input data to recognize specific features. It is still a feed forward network, in which the data is fed forward through the network, and then is trained/optimized by using backpropagation. The difference between the aforementioned (General) Neural Network and CNN is that instead of

each neuron's weight being updated, it is the values of the *convolutions* that are being updated.

This is how the model looks like

```
def get_cnn_model(self, learning_rate=1e-3, loss='categorical_crossentropy', metrics='accuracy'):
    optimizer = self.optimizer(lr=learning_rate)
    model_cnn = Sequential()
    model_cnn.add(Conv2D(32, (5, 5), input_shape=(28, 28, 1), activation='relu'))
    model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
    model_cnn.add(Dropout(0.5))
    model_cnn.add(Flatten())
    model_cnn.add(Dense(128, activation='relu'))
    model_cnn.add(Dense(10, activation='softmax'))

    model_cnn.compile(loss=loss, optimizer=optimizer, metrics=[metrics])
```

There are a lot of information in one picture, but will try to explain the most important aspects. First, there is an input layer = **Conv2D**, which has been explained earlier, with using convolutions to traverse over the input (this one has dimensions 5x5). This layer needs to specify the expected input size: (28, 28, 1), which is a 2D *tensor*, and normally how images are represented in CNNs. This required a reshape of the data:

```
X = X.reshape([-1, 28, 28, 1]).astype('float32')
```

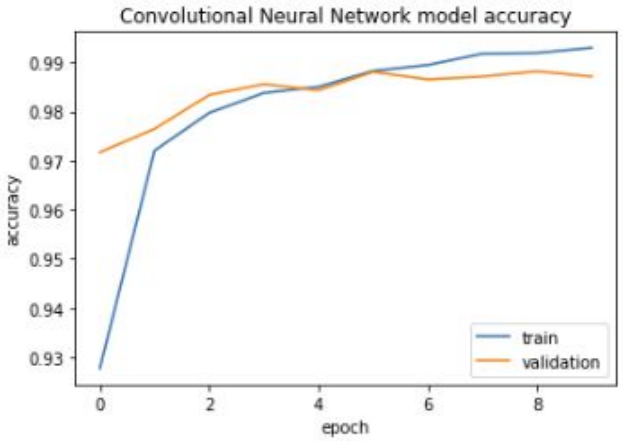
MaxPoolLayer2D is basically a down sizing/scaling layer, which takes a matrix with size = pool size, identifies the max number in that matrix, and deletes the other ones. Other alternatives are *AveragePooling*, *GlobalPooling*, etc.

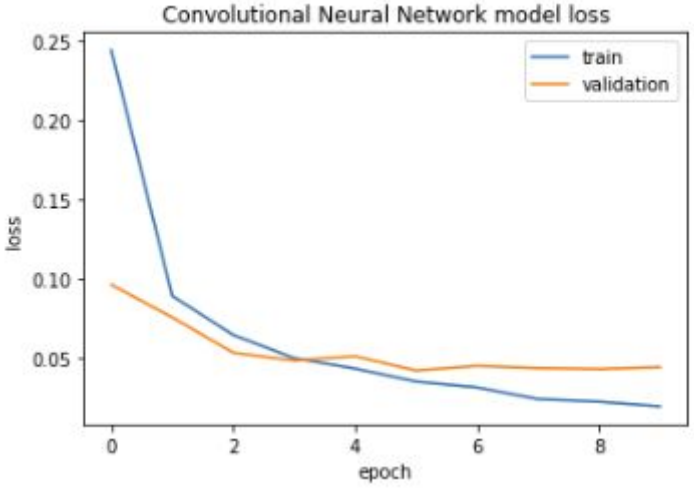
Dropout has a regularizing effect, similar to NN's *kernel regularizer*, but this works in the way that the activations in the input (to dropout layer) has a *dropout*% of getting deleted. This is totally random, so it forces the model to generalize, and not to rely on specific activations to make predictions.

Flatten layer takes the 3D tensor passed through the CNN, and makes it into a 1D tensor. This is necessary because the rest of the network is standard NN, which means that it needs 1D tensor.

Then the rest of the model is a standard NN, which has been explained earlier.

Here are some plots of the models training:



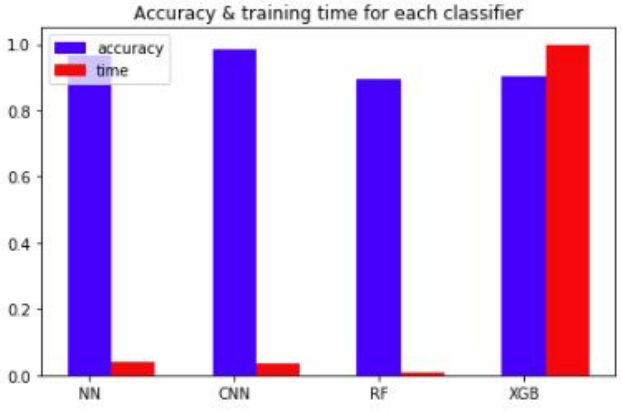


As one can see, the model is clearly overfitting, but by increasing the dropout rate this is somewhat trivial to combat.

Model Selection Scheme

As mentioned earlier, the chosen performance metric is *accuracy*, but one always needs to consider the time aspect as well - because of limited resources.

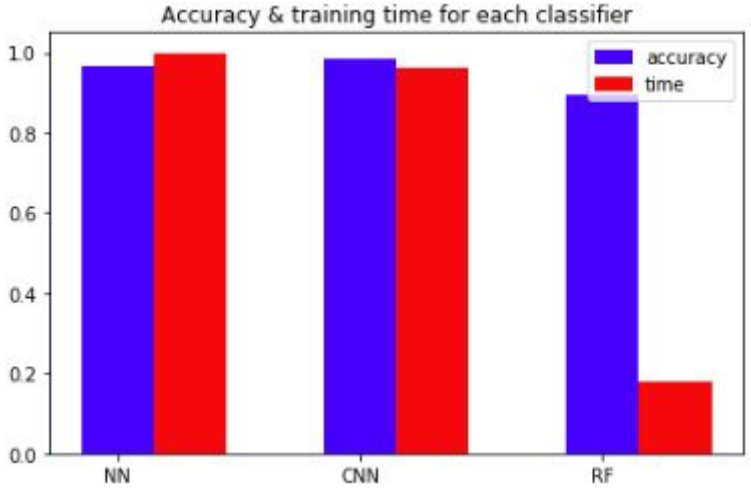
Here you can see the 4 different models plotted against each other.



For your information, the time-value has been normalized for all of the models in order to make it possible to plot it next *accuracy*.

```
normalizing_constant = max(time_list)
for i in range(len(time_list)):
    time_list[i] = time_list[i]/normalizing_constant
    print(time_list[i])
```

It is easy to see that the XGBoost model is polluting the plot, so by removing this, one gets a better look at the other models.



It seems as though the neural network models gets better accuracy, but the random forest has a significantly shorter training time. Let's look at the actual numbers:

```
'NN': {'accuracy': 0.9527976190476191, 'time': 66.88},
'CNN': {'accuracy': 0.9871428571428571, 'time': 279.143},
'RF': {'accuracy': 0.88, 'time': 12.476},
```

The neural networks outperforms the random forest with ~ 7-10%, and considering the relatively small increase in time, it is clear that the neural networks are better suited for this task. Furthermore, the CNN provides both an increase in accuracy, and a decrease in time, compared to the NN.

One can therefore conclude that the CNN is the “final classifier”, which fits with published results for image classification tasks.

Because this is the final classifier, one can try to optimize its performance by implementing Grid- or Random-search. However, it seems that Keras has some problems implementing GridSearchCV with Keras: https://stackoverflow.com/questions/48717451/keras-kerasclassifier-gridsearch-typeerror-cant-pickle-thread-lock-objects, which is why it was decided to implement a custom one. Lastly, get a sense of how well is performs on unseen data, e.g. how well it will perform in production.

```
start_cnn_optim = time.time()
counter = 0
for bs in batch_size:
    for epoch in epochs:
        for lr in learning_rate:
            optim = Adam(lr=lr)
            for pool_s in pool_size:
                for activat in activation:
                    for drop in dropout:
                        start_cnn_optim_each_model = time.time()
                        model_cnn_optim = create_cnn_model(optimizer=optim,
                                                           activation=activat,
                                                           pool_size=pool_s,
                                                           dropout=drop)
                        history_cnn_optim = model_cnn_optim.fit(X_train_cnn,
                                                y_train_cnn,
                                                validation_data=(X_val_cnn, y_val_cnn),
                                                epochs=epoch,
                                                batch_size=bs)

                        scores_cnn_optim = model_cnn_optim.evaluate(X_test_cnn, y_test_cnn)

                        end_cnn_optim_each_model = time.time()
                        total_cnn_optim_each_model = end_cnn_optim_each_model - start_cnn_optim_each_model

                        classifier_cnn[counter] = {'parameters':
                                                   {
                                                       'batch_size': bs,
                                                       'epochs': epoch,
                                                       'learning-rate': lr,
                                                       'pool-size': pool_s,
                                                       'activation': activat,
                                                       'dropout': drop
                                                   },
                                                   'scores':
                                                   {
                                                       'loss': scores_cnn_optim[0],
                                                       'accuracy': scores_cnn_optim[1]
                                                   },
                                                       'time': total_cnn_optim_each_model,
                                                       'history': history_cnn_optim
                                                   }
                        print(classifier_cnn[counter])
                        counter += 1


end_cnn_optim = time.time()
total_cnn_optim = end_cnn_optim - start_cnn_optim
```

Important to note here that the optimizer was not changed, only its learning rate. This is because as far as I understand, Adam is the go to optimizer today, because it combines the best of two world: *RMSProp* and *Momentum*.

The parameters used in the GridSearch:

```
batch_size = [50, 100]
epochs = [6, 12]
learning_rate = [1e-3, 1e-4]
pool_size = [(3, 3)]
activation = ['relu']
dropout = [7e-1, 5e-1]
classifier_cnn = {}
```

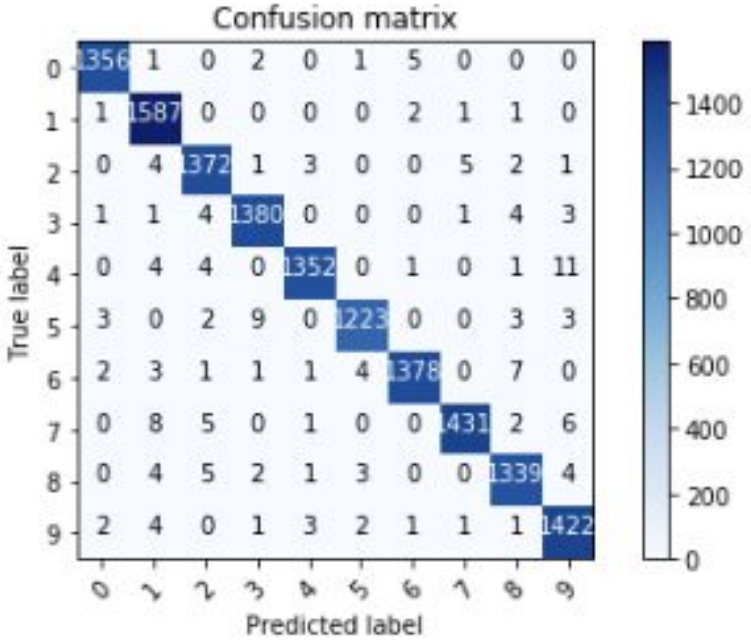
The optimal (accuracy) model was then chosen:

```
{'parameters': {'batch_size': 50,
  'epochs': 6,
  'learning-rate': 0.001,
  'pool-size': (3, 3),
  'activation': 'relu',
  'dropout': 0.4},
 'scores': {'loss': 0.038795391272237925, 'accuracy': 0.988},
 'time': 131.85629200935364,
```

Which also shows the score of the model. An accuracy of 0.988 is decent, but as we will see later, could be improved by more epochs, and more hyperparameters for the Grid Search.

One would assume that the model will perform somewhere around this number when deployed, because of the fact that the this score is from the *test*-set, i.e. unseen data. However, it is important to update the model, when it is in production.

This optimal model used to plot its confusion matrix, in order to learn what labels the model struggle with:



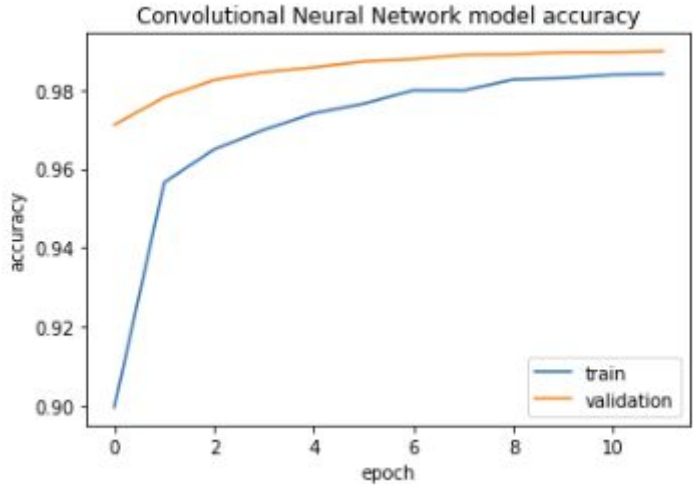
So, it seems like the model struggles with 9 in general, especially when the actual label is 0, 4, 5, and 7. Other interesting notes:

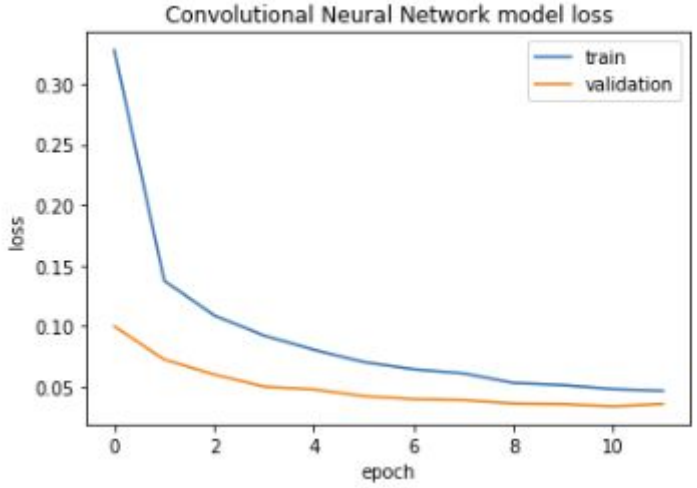
- 6 vs. 7
- 5 vs. 8
- 2 vs. 7

With that being said, it seems that the model is doing pretty ok:

	precision	recall	f1-score	support
0	0.99	0.99	0.99	1365
1	0.98	1.00	0.99	1592
2	0.98	0.99	0.99	1388
3	0.99	0.99	0.99	1394
4	0.99	0.98	0.99	1373
5	0.99	0.98	0.99	1243
6	0.99	0.99	0.99	1397
7	0.99	0.98	0.99	1453
8	0.98	0.99	0.99	1358
9	0.98	0.99	0.99	1437
micro avg	0.99	0.99	0.99	14000
macro avg	0.99	0.99	0.99	14000
weighted avg	0.99	0.99	0.99	14000

and it's training history plotted:





This does seem somewhat ok, but with potential for improvement, such as decreasing underfitting, which is a perfect segway to the next point.

Increased Resources

Given more resources it would've been optimal to try more values in the CNN GridSearch, such as decreased dropout and increased learning rate, and test if more layers are relevant. Furthermore, given more time one could clean up the dataset, as mentioned in the Executive Summary, and maybe extend the dataset.

When deployed

To verify the models performance in production, it was scored on unseen data - the *test*-set:

```
0.9885714285714285
```

This seems like a satisfactory starting point.