A study of Convolutional Neural Networks

Kristian Aalling Sørensen[1]

1 Faculty of Applied Science, Department of Engineering, University of British Columbia Okanangan, 1137 Alumni Ave, Kelowna, BC V1V 1V7, Canada. e-mail: s154443@student.dtu.dk

November 23, 2019

ABSTRACT

Machine learning algorithms are used progressively more in a wide range of different applications. A Convolutional Neural Network(CNN) is an especially efficient method for pattern recognition and feature extraction of large arrays. In this article, the idea behind CNN will be described leading to a basic understanding of the essential components in a CNN. The theory will be implemented in a simple CNN model to illustrate the coding of such a CNN using the Python library Keras and Google's Tensorflow framework. The resulting model had an accuracy of 82%. This article shows how to reflect on CNN results in order to improve a model through many iterations. The implemented code showed how easy it is to build a intermediate CNN model, but also illustrates the difficulties in making an accurate and *efficient* model.

Key words. Deep learning – Convolutional Neural Network – Image Classification – Feature extraction - Machine Learning -

1. Introduction

Inference using Machine Learning(ML) methods is gaining more and more recognition in today's society. Tasks that was previously assigned to experts have been taken over by different ML algorithms depending on their nature. Previously, the detection of features or objects in e.g. images have been done by highly trained experts with specific domain knowledge taking time and costing money.
Convolutional Neural Network(CNN) is a type of Deep Neural Network(DNN) used specifically to learn features or detect patterns in large arrays of data, be it speech recognition or images. The true power of CNN is that it can detect patterns by utilizing local connections and weight sharing by applying convolutions instead of element-by-element operations as in conventional Neural Networks(NN), it therefore does not require any prior features.
In this paper, the theory of CNN will be given, describing some of the most conventional layers and methods used in the ML community starting with the general architecture of a CNN in section 2 followed by the most common layers; Convolutional layers, Pooling layers and Fully Connected(FC) layers. Thereafter, the methods for training the CNN is described including activation functions, loss functions and regularizations. Also, specific ways of optimizing a CNN is outlined.
In section 3, a CNN model will be implemented. A famous training set, the CIFAR-10 data set will be used to implement a simple CNN model illustrating the usage of CNN and how to improve it. In section 4, some of the many parameters in the CNN will be discussed.

2. Convolutional neural network

Convolutional Neural Network (CNN) is a type of Deep Neural Network(DNN), see e.g. [LeCun(2015)], consisting of several different layers. Like other Neural Networks(NN) it is inspired by the brain, and more specifically by the visual cortex[Aloysius(2017)]. The motivation for using a CNN instead of a NN, be it deep or shallow, is partly due to the efficiency problem with NN when working with arrays or large data sets, be it 1D e.g. speech recognition, 2D e.g. images or 3D e.g. volume or movies. In a NN, the number of neurons for e.g. a RGB image in the first layer alone will be too large to compute. For instance, for satellite images it could very well result in $\approx 2^{13}$ neurons. This motivates the uses of layers where weights are shared, see section 2.1. Not only is CNN more efficient that classic NN, it also outperforms all other NN in image analyses when enough layers are used, i.e. when it is *deep* enough, see [Sze(2017)]. A typical CNN architecture is illustrated in figure 1.
The general CNN architecture builds on the following ideas:

1. Local connections; Neighbouring pixels are correlated.
2. Shared weights; Weights in a feature map is shared and spatial information is kept, see section 2.1.
3. Pooling; Reduce dimensionality and generalize to unknown objects, see section 2.2.
4. Many layers; A deep CNN can detect increasingly more complex structures.

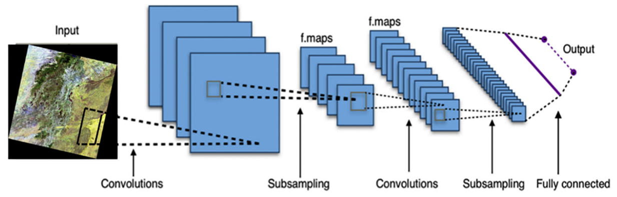


Fig. 1. Basic architecture of a CNN includes a convolutional layer followed by sub-sampling and an activation [Traore(2018)].

In figure 1, an image with several bands(channels) is used as input into the first convolutional layer which applies a con-

volution. Here, the input image is convoluted with different filters each initialized with random weights. Then, an activation function is used in order to acquire a feature map. Thereafter a pooling is applied to reduce the spatial dimensionality(the subsampling). Many of such *blocks* of layers can be used to increase the complexity. Lastly, a Fully Connected(FC) layer with an activation function is applied in order to classify the input.
The convolutional layer is described in section 2.1. The pooling layer will be described in section 2.2 and in section 2.3 the FC layer is described.[Sze(2017)]

2.1. Convolutional layer

The name Convolutional Neural Network comes from the algebraic expression of convolution which, in the regime of arrays, is a filtering[Skalski(2019)]. The convolutional layer is given by:

$$\mathbf{z}^{(n)} = \mathbf{F}^{(n)}\mathbf{A}^{(n-1)} + \mathbf{b}^{(n)}, \tag{2.1}$$

where $\mathbf{z}^{(n)}$ is the intermediate output values, or *activity*, for layer n, $\mathbf{F}^{(n)}$ is a tensor containing all the filters for layer n with specific heights and widths. The amount of filters for a given input is called the layer's *depth*. $\mathbf{A}^{(n-1)}$ is the previous layer. This is either the input layer, or the output from a previous layer, in which case it is called a *feature map*[1]. The amount of feature maps, $\mathbf{z}^{(n)}$ is thus the same amount of applied filters. $\mathbf{b}^{(n)}$ is the bias added to the feature map.
Taking the activity from equation (2.1) and applying an activation function, $f(\cdot)$, will result in the feature map, $\mathbf{A}^{(n)}$ which will be passed on to the next layer. See section 2.4.

$$\mathbf{A}^{(n)} = f(\cdot)^{(n)}\left(\mathbf{z}^{(n)}\right). \tag{2.2}$$

Each feature map from a layer illustrates different attributes with increasing complexity. The first convolutional layer could e.g. have extracted horizontal/vertical lines. Next time a convolutional layer is used, edges are found. Then edges are combined into patterns, which will be combined into parts, which is used by a convolutional layer to find objects. By using many convolutional layers, complex features can thus be extracted.
All elements in a feature map shares their weights[2]. This *weight sharing* reduces the number of parameters with increasingly larger data sets, as we saw in section 2. By applying the weight sharing, the CNN is less sensitive to change in location, i.e. a filter detects e.g. vertical lines no matter the location in the 2D array due to the convolution. This independence of location is highly important in e.g. image classification.[LeCun(2015)]

2.2. Pooling layer

A pooling layer *pools* neighbouring elements together and offers two advantages. Most importantly, the pooling layer reduces the spatial dimensionality of the information in the feature map which is essential when working with either large data-sets or deep networks since the computing power needed would increase exponentially. Secondly, the pooling layer ensures that extracted features can be generalized to unknown data, see [Sze(2017)]. Different kinds of pooling methods exits, each performed on a single feature map. In equation (2.3), the Max-Pooling is shown with a $(2x2)$ kernel. Here, it can be seen just how a pooling layer reduceds the dimensionality.

$$\begin{vmatrix} 1 & 3 & 5 & 1 \\ 1 & 1 & 2 & 1 \\ 5 & 4 & 2 & 1 \\ 5 & 9 & 8 & 1 \end{vmatrix} \xrightarrow[\text{Pooling}]{\text{Max}} \begin{vmatrix} 3 & 5 \\ 9 & 8 \end{vmatrix}. \tag{2.3}$$

Often, a *stride* is applied to the pooling layer. This offers the advantage of skipping neighbouring pixels defined by the stride value. In equation (2.3), a stride value of 2 is used to skip every second pixel. Larger stride offers a larger reduction of dimensionality. Typically, the stride is equal to size of pooling or larger, meaning that there is no overlap. It has though been argued that an overlap would increase the accuracy, see e.g. [Krizhevsky(2018)].

2.3. Fully connected Layer

When a CNN is used for classification, a FC layer is often used as a classification schema. This is the last layer, placed after the last pooling layer, as shown in figure 1, see [Traore(2018)]. The FC layer is a vector of numbers, with the same size as the reduced output layer from the last pooling. The input is hence the last pooling layer, and the output is a vector of the same size as the wanted classes[3] see Sakryukin (2018). In short, this works much like a traditional Neural Networks, see e.g [MacKay(2003)]. The activation function in such a FC could e.g. be the log softmax activation function as given by:

$$f(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_i)}, \tag{2.4}$$

here the input x_i is evaluate at every class and thus found to belong to a single class. The output is hence turned into a probability distribution, and a class can therefore be evaluated.

2.4. Activation function

In equation (2.2) we saw the usage of an activation function in order to get the feature map. Many different activation functions exists. The convolutional-, pooling -and the FC layer are all linear operations, it is therefore the activation function's task to introduce a non-linearity that enables the approximation of complicated non-linear transformations. A widely used activation function in the CNN regime is the Rectified Linear Unit(ReLU) as given by[Gadosey(2019)]:

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.5}$$

where the input x is mapped as being x if the value from the feature map is 0 or above and 0 otherwise. Similarly, the Sigmoid- and Tanh functions are given by:

$$f_{Sigmoid}(x) = \frac{1}{(1+\exp(-x))}, \quad f_{Tanh}(x) = \frac{2}{(1+\exp(-2x))} - 1. \tag{2.6}$$

The Sigmoid- and the Tanh functions are both saturated whereas the ReLU function is non-saturated. If the input for the Sigmoid(Tanh) function is very low or very high, the output "saturates" to 0 or 1(-1) which will result in small gradients and thus poor convergence, see section 2.5. Moreover, it has been

[1] This feature map is comparable to neurons in a classic NN
[2] Comparable to synapses in a classic NN

[3] There can be more than one FC layer, as it just describes a layer connected to each node in its input layer.

shown that precaution in choosing the initial weights must be taken to avoid problem where neurons aren't learning using saturated function, see [Zhang(2018)].
Not only is the ReLU function non-saturated, it has also been shown that using this activation function will result in faster convergence and same accuracy as other functions, see [LeCun(2015)] and [Sze(2017)].

2.5. Loss function

By comparing the output from the CNN, i.e. the result from the FC layer, with the target value from the input, i.e. known classes, we can find the error in the network's estimate using a cost function. This error, or *loss*, can then be used to update the random initialized weights to increase the model performance. The loss function acts in both the forward- and the backwards direction;
In the forward propagation, the loss function finds the loss from the CNN output and the target input (label).
In the back-propagation schema, this cost function is used to calculate error gradients in a method called Gradient Descent. Here, the weights and biases in the previous layers are update accordingly in a method called Backward Pass. Firstly, the loss function has to be defined. This could be any differentiable function, since a gradient will be found. Commonly used expression are the cross entropy as seen in equation (2.7), Root Mean Square, L2-Norm or the Kullback-Leibler divergence(relative entropy) as seen in equation (2.8). The cross entropy is given as: [Sakryukin (2018)]

$$G(\mathbf{w}) = -\sum_{j}^{M} t_n \ln\left(y(\mathbf{x}^{(n)}, \mathbf{w})\right), \tag{2.7}$$

in which M is the number of classes, t is the vector of true labels and y is the output of e.g. a FC layer. Similarly, the Kullback-Leibler error function is given by[MacKay(2003)]

$$G(\mathbf{w}) = -\sum_{n} \left[t^{(n)} \log\left(y(\mathbf{x}^{(n)}, \mathbf{w})\right) + \left(1 - t^{(n)}\right) \log\left(1 - y(\mathbf{x}^{(n)}, \mathbf{w})\right)\right], \tag{2.8}$$

where $\mathbf{x}$ is the data, $\mathbf{y}$ the output of e.g. a FC layer, and t the true labels. From the error function, e.g. equation (2.7) or (2.8), the error gradients with respect to each weight are found, starting from the last layer. For e.g. the Kullback-Leibler loss function, the gradient is:

$$L = \frac{\partial G(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{eq(2.8)} = -\sum_{n} \mathbf{x}^{(n)}(t^{(n)} - y^{(n)}) = \sum \mathbf{g}^{(n)}. \tag{2.9}$$

The Gradient Descent methodology is often used in CNN problems [Traore(2018)]. By applying the Backward Pass multiple times until a convergence criteria is reached, the weight and biases will thus be adjusted.

2.6. regularization

When updating the weights according to section 2.5, there is a risk of the weights, $||\mathbf{w}||$ blowing up. As with saturated activation functions, described in section 2.4, this can result in over-fitting. For these reasons, a regularization is made. Different kinds of regularizations exits, e.g. penalizing the weights by adding a bias term to the loss function.
One such penalty is the weight decay regularizer, which modifies the loss function to:

$$G(\mathbf{w})_{regu} = G(\mathbf{w}) + \alpha_k \sum \mathbf{w}_k^{in} + \beta_k \sum \mathbf{w}_k^{bias} + \gamma_k \sum \mathbf{w}_k^{out}. \tag{2.10}$$

Here, the weights for input- and output layers and the bias are regularized differently to avoid over-fitting where α, β, γ are their corresponding penalty.
Another regularization technique is called dropout. Using this regularization, feature maps are ignored at random in the training phase, defined by a certain probability, i.e. *dropped*. This is especially useful in FC layers where feature maps are too dependant and thus prevents over-fitting or when there is a over representation of one class.

2.7. Batch learning

A group of inputs can be used in conjunction to update their weights by calculating gradients according to equation (2.9). This is called batch learning and can *potentially* improve the weights and speed. The number of inputs in a batch is called the batch size. If one input is used to calculate the gradient, equation (2.9) is called *stochastic* gradient descent. If a batch is used, equation (2.9) is called mini-batch learning. The amount of times the same mini-batch learning is done is called *epochs*.

2.8. Batch normalization

As seen in section 2.4, a CNN updates its weights by adding gradient errors as found from the back propagation. Therefore, if the input data isn't scaled, the ranges of the extracted features might differ with different error updates in each feature map resulting in potential over-fitting of one feature, and under-fitting of another. For this reason, a normalization is performed. Moreover, performing a normalization has been shown to speed up training and improve accuracy, see [Sze(2017)]
Different forms of normalization exist, including Batch normalization, Weight Normalization, Layer Normalization, Instance Normalization, Group Normalization and many more, see e.g. [Kurita(2018)]. In Batch normalization, the normalized value for each batch is scaled and shifted according to:

$$y = \left(\frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\right)\gamma + \beta, \tag{2.11}$$

where (β, γ) are the scaling and shifting parameter respectively, as found from training. x the input from the mini-batch m and $\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$, where m is the size of the batch. σ^2 is the mini-batch variance, $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$. ϵ is a small constant used to avoid numerical instabilities. The first part of equation (2.11) is thus the normalization, and the latter part the scaling and shifting. The weight are thus constrained by the mean and variance.[Sze(2017)]

2.9. Limitations of CNN

As stated in section 2.1, a CNN can detect increasingly more abstract structures with more layers and more filters. It is however not possible to add an infinitely large amount of layers due to the limitations in computation. Too complex CNN architectures are too computational intensive to be efficient so when designing a CNN, efficiency must be a key factor.[Zhang(2018)]

3. Methodology

There are many modifications of a CNN one can make depending on the complexity on ones problem and data sets. LeNet-5 uses 5 layers for document recognition [LeCun(1998)], AlexNet uses 8 layers for image recognition [Krizhevsky(2018)] and ResNet uses 152 layers for object detection [He(2016)]. Each of these CNN models have different features but all share the overall idea of a CNN network]Zhang(2018)]. The CNN model in this paper is mainly used for illustrative purposes - it shares the same ideas as better CNNs, as described in section 2, but does not have as good an accuracy.

3.1. Data set

The used data set is the Canadian Institute For Advanced Research(CIFAR)-10 data set. It is a collection of images widely used in the machine learning community to train NNs and especially CNNs, see [Krizhevsky(2019)] for details. It is used here for two reason, firstly is consists of RGB images that can be used for classification and secondly it is easily accessible on APIs such as Keras. The CIFAR-10 data set was split up into 50000 training samples and 10000 testing samples, both includes 10 classes. The training samples are used to train the model where after the testing data is applied to the CNN in order to find its accuracy. The classes used for classification are airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

3.2. CNN model

The model is made in Python using the Keras API in order to utilize Google's Tensorflow Machine Learning framework which includes the various functions and layers described in section 2, it moreover uses a Tensor Processing Unit (TPU) which supports tensors (i.e many filters) with both integers and floating points[4]. The TPU offers the advantage of running comparatively fast independently of CPU or GPU[5]. To make the model, hyper parameters have been chosen and iterated through to optimize the CNN. In table 1, a summery of the hyper parameters is displayed.

Depth	**Pooling size**	**Stride**	**Epochs**
32	$(2x2)$	$(2x2)$	90
Learning Rate	**P[DropOut]**	**Batch size**	
0.1-0.001	0.25	90	

Table 1. Hyper parameters used in the CNN model. Many other parameters can be defined and tweaked for optimum performance.

The CNN model is build using the layers, functions and hyper parameters as described in section 2 and is seen in Appendix A. The model consist of 6 blocks of layers with a convolutional layer, an activation function, a batch normalization and a pooling layer. The used activation function is in all instance the ReLU function. In the FC layer, a softmax activation function was used as classifier. In short, making a CNN model require the decision of many design parameters.

[4] `https://www.tensorflow.org/tutorials/keras/classification`

[5] This model is made using Google Drive's CodeLabs which offers the option of running code on GPU.

3.3. Results

After training the model using the hyper parameters in table 1 a model is made. This model is saved and evaluated using the testing data and is shown in Appendix A. By applying different schemes, different model accuracy have been found with a resulting accuracy of 82% as we can see in figure 2. We see that the accuracy increases with more epochs. Also, looking at table A.1, we see see how it in general increases with each layer.

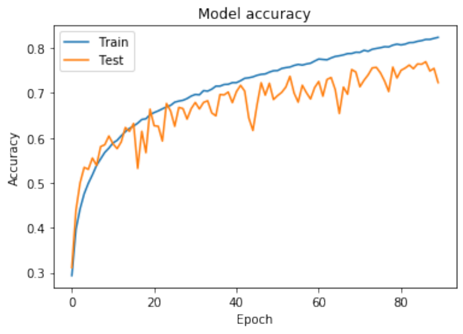


Fig. 2. The accuracy of the training can be seen here throughout different epochs. It can be seen that the Accuracy increases meaning the CNN indeed learns. Moreover, it could look as if the model started overfitting, since the training accuracy is better than the test accuracy.

Moreover, in figure 3, a confusion matrix is shown illustrating the confidence of each class. Here, the amount of true/false positives/negatives can be seen. The classifier is not perfect, but performs well on distinctive classes such as cars. Similarly, in table A.1 the precision and recall is shown showing the same tendencies.

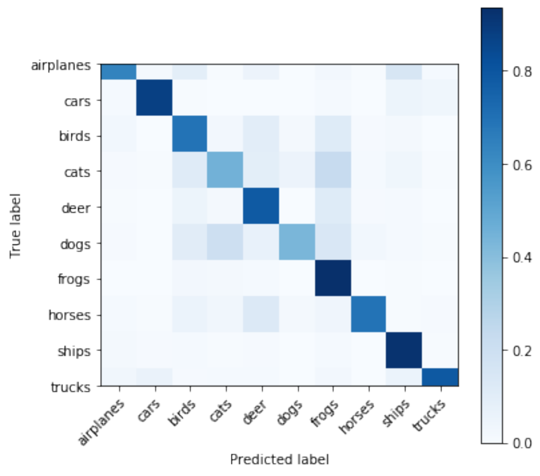


Fig. 3. A confusion matrix showing the accuracy of each class. Moreover, the amount of false positive and false negatives are seen on the off diagonal. Here, we e.g. see horses have comparatively many false positives of deer. This is probably due to the similarities of the two.

In figure 4, the classification of a car is made. According to the confusion matrix, the model should be accurate with that class with an accuracy of 82%. The picture was taken at the parking lot of University of British Columbia, re-scaled to have same size as the input pictures and classified using the model.

It was found that the CNN indeed classified a picture of a car as belonging in the car class. It thus illustrated

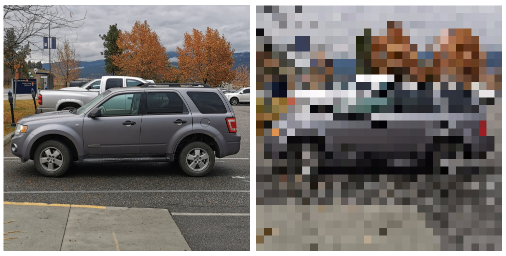


Fig. 4. (left): Picture of a car taken from the parking lot at UBCO. (Right): The down-sampled image feeded into the CNN. It is possible to feed the model various sizes, but that is a tedious task as it has been trained on one size.

the power of a CNN - using trained data it has learned to detect cars (and other classes). For Python notebooks, go to `https://github.com/aalling93/ENGR562_CNN.git`. Here, the entirety of the CNN can be seen.

4. Discussion

Making a CNN is a craft. There is no 'best' solution, and often it is needed to iterate through the parameters many times. Before that can be done, many subjective decisions must be made. Some of these choices are discussed in the following:

1. Filter size: A filter size of $(3x3)$ has been used for the filtering. The images used are size $(32x32x3)$ and it is therefore expected to be enough for filtering. If larger images are used, e.g. satellite image, much larger filters should be applied.

2. Learning rate: A parameter used in the Keras framework is the learning rate, i.e. how much the weights are changed based on the gradient error. In figure 2, a learning rate of 0.0001 is used. In figure B.2 the default learning rate of 1 is used. Here, it can be seen that the accuracy and loss fluctuates around a value. This is because the the value the weight are changed every time it too large. Therefore, a 'low enough' learning rate should be used. A too low learning rate take a lot of time to teach.

3. Epochs: In figure 2, 90 epochs have been used to find the model loss and the model accuracy for both the training- and test data. We see that the accuracy increases as the loss decreases, but we also see that the test accuracy fluctuates quite a lot. A smaller learning rate could thus be used. In the same figure, it would be possible to see a possible over-fitting, is the trained model accuracy was better than the test accuracy, and similarly is the train loss was smaller than the test loss. In figure B.2, 90 Epochs have been used, with a learning rate of 0.0005 and a batch size of 900. This is an extreme example, but illustrates how one parameter changes the accuracy. We see in figure B.3, that the accuracy is 60%, i.e. lower than the found 82%.

4. ReLU: The ReLU function has been used for all the convolutional layers. This is accepted in the ML community as being better than the saturated function, e.g. Tanh. In figure B.1, we can see the model loss of the data using a Tanh-, Sigmoid and ReLU respectively. Also, the model loss using no activation function is used as reference. For many epochs we see no change, but when running the model, the ReLU activation function converged faster(time wise).

5. Depth: The depth of all the convolution layers were chosen to be 32, i.e. 32 feature maps were generated in the first layer. This is quite deep for such a small model, but for more complex system, this should be deeper.

6. Batch size: Varying batch sizes have been tried out, but generally, the same size as the depth has been used. When the batch size was small, fluctuations were seen in the accuracy curve. This could be due to the variances between individual batches. When the batch size was large, it too much longer time to compute the model and the model converged earlier. Empirical results showed, that using too large of a batch size would reduce the accuracy of the model.

7. Pooling: throughout the pooling layers, a MaxPooling has been used. There exists many different ones, but study suggest this is the most stable one, see e.g. [Traore(2018)], different strides have been tried with the last result being a non-overlap.

8. Loss function: The cross entropy has been used as a loss function. It resembles the Kullback-Leibler function, but where the cross entropy calculates the total entropy over distributions, the Kullback-Leibler is calculated over two distribution. For this reason, the cross entropy is used as a loss function.

9. Batch normalization: A batch normalization was used. In order to see if that indeed improved the accuracy, the same model was made with, and without batch normalization. Still, it is being debated if if actually works or not, see [Santurkar(2019)].

10. Blocks: A block consisted of a convolutional neural network, an activation function and a pooling layer. Using 3 blocks, the best accuracy found was 64% (by changing various parameters), using 5 blocks an accuracy of 82% was found and with 6 blocks, is was 67%.

In general, making a good CNN model takes time. Using only the theory in this paper, one can get a long way with a simple model. But tweaking the model and improving the accuracy just 1% takes time - here one parameter is changed at a time to find the best mix, i.e. setting the depth at 32, then training the model followed by a change of the depth to e.g. 28 and yet another training and choosing the best depth by e.g. looking at the accuracy or cross validation.

5. Conclusion

In this paper Convolutional Neural Networks have been studied and explained. The theory in section 2 introduces the most important elements of a CNN, useful for beginners in the Machine Learning community; The most important layer, the convolutional layer which convolves the input data with filters resulting in a weight sharing and spatial Independence in the feature map. The activation function which introduces a non linearity. The Pooling layer which down samples the information in the feature map. The loss function which is used to train the CNN. Other methods for optimising the CNN was also described such as regularization, batch learning and normalization.
The theory was also implemented on a CNN model using a simple data set to illustrate how easy it is to develop a CNN. It should be clear that making a CNN is easy, but making it good is

difficult. The CNN model made in this paper had an accuracy of 82% which is good, but not good enough to replace the human experts. Therefore, more iterations should be made before this is useful in practise, e.g. add more layers, deeper layers, tweak parameters of change functions, many possibilities exists.

Acknowledgements. Building a CNN is a craft. Highly specific articles and too general books often falls short in a field that is constantly changing. I therefore would like to acknowledge the people writing blog articles on e.g `https://medium.com/` and `https://mlexplained.com/`.

References

MacKay, D. J. C. et alt. 2003, Information Theory, Inference, and Learning Algorithms, ISBN:0521642981

Aloysius, N. et alt. 2017, A review on deep convolutional neural networks, `https://10.1109/ICCSP.2017.8286426`

Traore, BB. et alt. 2018, Deep convolution neural network for image recognition, `https://doi.org/10.1016/j.ecoinf.2018.10.002`

LeCun, Y. et alt. 2015, Deep Learning, `https://doi.org/10.1038/nature14539`

Zhang, Q. et alt. 2018, Recent advances in convolutional neural network acceleration, `https://doi.org/10.1016/j.neucom.2018.09.038`

LeCun, Q. et alt. 1998, Gradient-based learning applied to document recognition , `https://doi.org/10.1109/5.726791`

Krizhevsky, A. et alt. 2012, ImageNet classification with deep convolutional neural networks, `http://dl.acm.org/citation.cfm?id=2999134.2999257`

He, K. et alt. 2016, Deep Residual Learning for Image Recognition

Zhang, Q. et alt. 2018, Recent advances in convolutional neural network acceleration, `https://doi.org/10.1016/j.neucom.2018.09.038`

Sze, V. et alt. 2017, Efficient Processing of Deep Neural Networks: A Tutorial and Survey, `https://doi.org/10.1109/JPROC.2017.2761740`

Santurkar, S. et alt. 2019, How Does Batch Normalization Help Optimization?, `https://arXiv:1805.11604`

Andrey Sakryukin, 2018,Under The Hood of Neural Networks. Part 1: Fully Connected. `https://towardsdatascience.com/under-the-hood-of-neural-networks-part-1-fully-connected-5223b7f78528`, visited on the 11/03/2019

Piotr Skalski, 2019, Gentle Dive into Math Behind Convolutional Neural Networks, `https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9`, visited on the 11/07/2019

Pius Gadosey, 2019, A beginner's guide to NumPy with Sigmoid, ReLu and Softmax activation functions, `https://medium.com/ai-theory-practice-business/a-beginners-guide-to-numpy-with-sigmoid-relu-and-softmax-activation-functions-25b840a9a272`, visited on the 11/01/2019

Kurita, K, 2018, An Overview of Normalization Methods in Deep Learning, `https://mlexplained.com/2018/11/30/an-overview-of-normalization-methods-in-deep-learning/`, visited on the 11/01/2019

Krizhevsky, A, 2019, The CIFAR-10 dataset, `https://www.cs.toronto.edu/~kriz/cifar.html`, visited on the 10/21/2019

In Appendix A, the output of the resulting model is shown. In figure A.1, the resulting model is shown, wherein information on all layers are shown. In table A.1, the precision for individual layers are shown. In figure A.2, a summary of layers are shown including their parameters.

In Appendix B output from the parameter analyses is shown including the usage of activation functions, epochs and learning rates.

For the full notebooks, the reader is referred to the Github repository `https://github.com/aalling93/ENGR562_CNN.git`, Using these, developing ones own CNN model should be easy.

Appendix A: Model results

Table A.1 shown the intermediate values from each layer.

-	precision	recall	f1-score
0	0.82	0.49	0.61
1	0.83	0.71	0.77
2	0.61	0.41	0.49
3	0.47	0.28	0.35
4	0.54	0.58	0.56
5	0.58	0.51	0.54
6	0.36	0.97	0.53
7	0.90	0.47	0.62
8	0.68	0.84	0.75
9	0.79	0.72	0.75
accuracy	-	-	0.6
macro avg	0.66	0.60	0.6
weighted avg	0.66	0.60	0.6

Table A.1. $Precision = \frac{TP}{TP+FP}$ is shown as well as $Recall = \frac{TP}{TP+FN}$ and $f1 = 2\frac{Precision \cdot Recall}{Precision + Recall}$ for each layer. I.e. precision is the amount of correct positive identifications, Recall the proportion of correctly identified positives $f1$ is a measure of accuracy. Interestingly enough, the $f1$ score is largest for layer 1 and converges at layer 8.

The figure A.1 displays the layout out the CNN. The different layers are shown with their parameters and shapes.
Figure A.2 shows the summary of the CNN. Here, the details of the layers are shown, i.e. which layers are shown and how big they are. Moreover, the amount of parameters are shown. In this CNN, a total of 34.912 parameters are used, of which 34.656 parameters are trained.

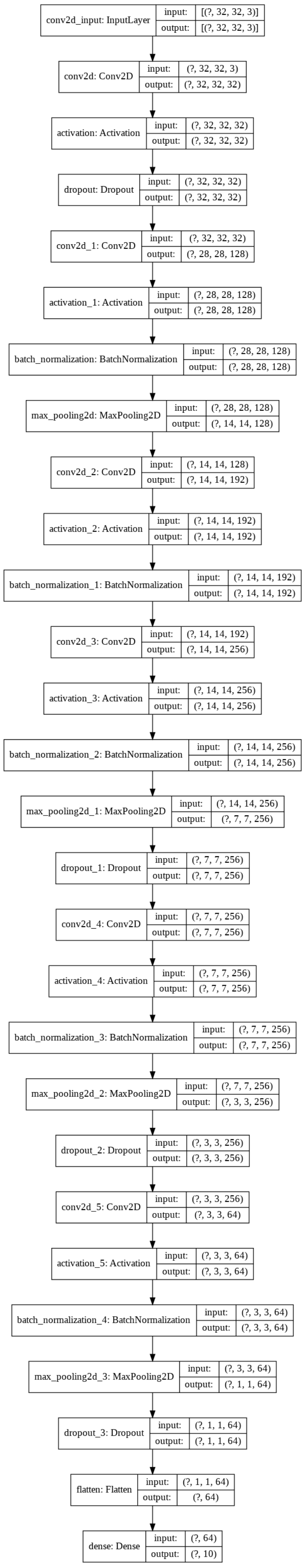
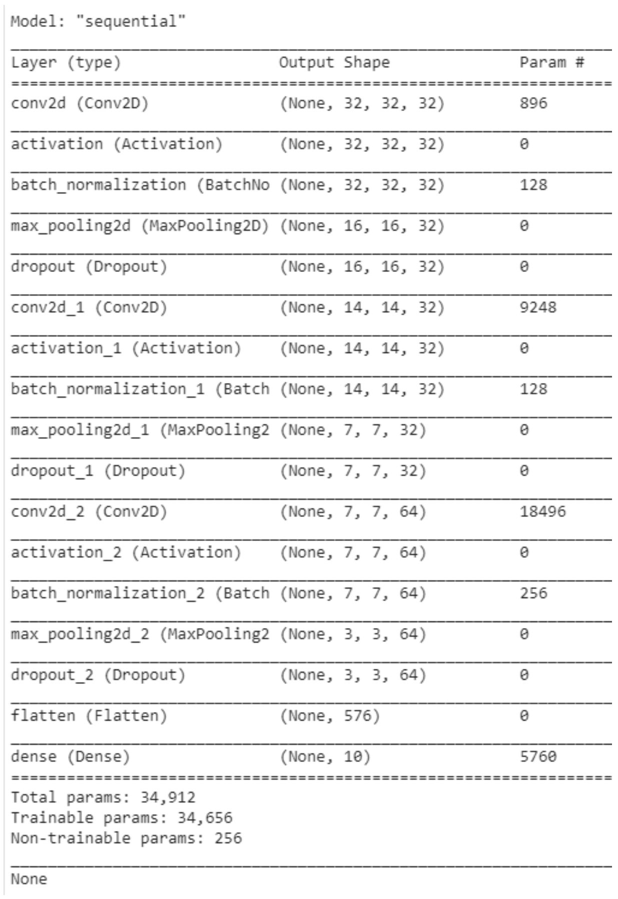

conv2d_input: InputLayer input: [(?, 32, 32, 3)] output: [(?, 32, 32, 3)]
conv2d: Conv2D input: (?, 32, 32, 3) output: (?, 32, 32, 32)
activation: Activation input: (?, 32, 32, 32) output: (?, 32, 32, 32)
dropout: Dropout input: (?, 32, 32, 32) output: (?, 32, 32, 32)
conv2d_1: Conv2D input: (?, 32, 32, 32) output: (?, 28, 28, 128)
activation_1: Activation input: (?, 28, 28, 128) output: (?, 28, 28, 128)
batch_normalization: BatchNormalization input: (?, 28, 28, 128) output: (?, 28, 28, 128)
max_pooling2d: MaxPooling2D input: (?, 28, 28, 128) output: (?, 14, 14, 128)
conv2d_2: Conv2D input: (?, 14, 14, 128) output: (?, 14, 14, 192)
activation_2: Activation input: (?, 14, 14, 192) output: (?, 14, 14, 192)
batch_normalization_1: BatchNormalization input: (?, 14, 14, 192) output: (?, 14, 14, 192)
conv2d_3: Conv2D input: (?, 14, 14, 192) output: (?, 14, 14, 256)
activation_3: Activation input: (?, 14, 14, 256) output: (?, 14, 14, 256)
batch_normalization_2: BatchNormalization input: (?, 14, 14, 256) output: (?, 14, 14, 256)
max_pooling2d_1: MaxPooling2D input: (?, 14, 14, 256) output: (?, 7, 7, 256)
dropout_1: Dropout input: (?, 7, 7, 256) output: (?, 7, 7, 256)
conv2d_4: Conv2D input: (?, 7, 7, 256) output: (?, 7, 7, 256)
activation_4: Activation input: (?, 7, 7, 256) output: (?, 7, 7, 256)
batch_normalization_3: BatchNormalization input: (?, 7, 7, 256) output: (?, 7, 7, 256)
max_pooling2d_2: MaxPooling2D input: (?, 7, 7, 256) output: (?, 3, 3, 256)
dropout_2: Dropout input: (?, 3, 3, 256) output: (?, 3, 3, 256)
conv2d_5: Conv2D input: (?, 3, 3, 256) output: (?, 3, 3, 64)
activation_5: Activation input: (?, 3, 3, 64) output: (?, 3, 3, 64)
batch_normalization_4: BatchNormalization input: (?, 3, 3, 64) output: (?, 3, 3, 64)
max_pooling2d_3: MaxPooling2D input: (?, 3, 3, 64) output: (?, 1, 1, 64)
dropout_3: Dropout input: (?, 1, 1, 64) output: (?, 1, 1, 64)
flatten: Flatten input: (?, 1, 1, 64) output: (?, 64)
dense: Dense input: (?, 64) output: (?, 10)

Fig. A.1. Illustration of the CNN model. Even this relatively simple CNN consist of many layers and parameters. This is illustrated here.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896
_________________________________________________________________
activation (Activation)      (None, 32, 32, 32)        0
_________________________________________________________________
batch_normalization (BatchNo (None, 32, 32, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 32)        9248
_________________________________________________________________
activation_1 (Activation)    (None, 14, 14, 32)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 14, 14, 32)        128
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 32)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 7, 7, 32)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 64)          18496
_________________________________________________________________
activation_2 (Activation)    (None, 7, 7, 64)          0
_________________________________________________________________
batch_normalization_2 (Batch (None, 7, 7, 64)          256
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 3, 3, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 3, 3, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 576)               0
_________________________________________________________________
dense (Dense)                (None, 10)                5760
=================================================================
Total params: 34,912
Trainable params: 34,656
Non-trainable params: 256
_________________________________________________________________
None
```

Fig. A.2. Summary of each layer. E.g the first convolutional layer has a depth of 32

Appendix B: Parameter discussion

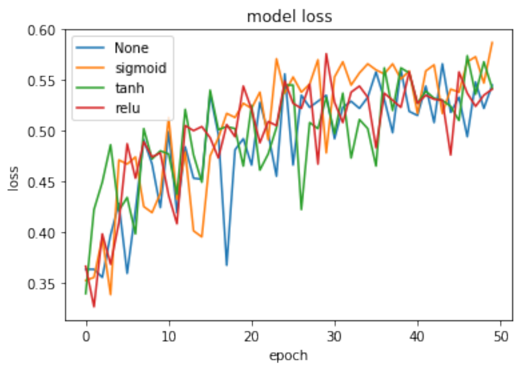


Fig. B.1. Illustrations of how all activation functions showed the same tendencies. Since ReLU is faster to train, this is used.

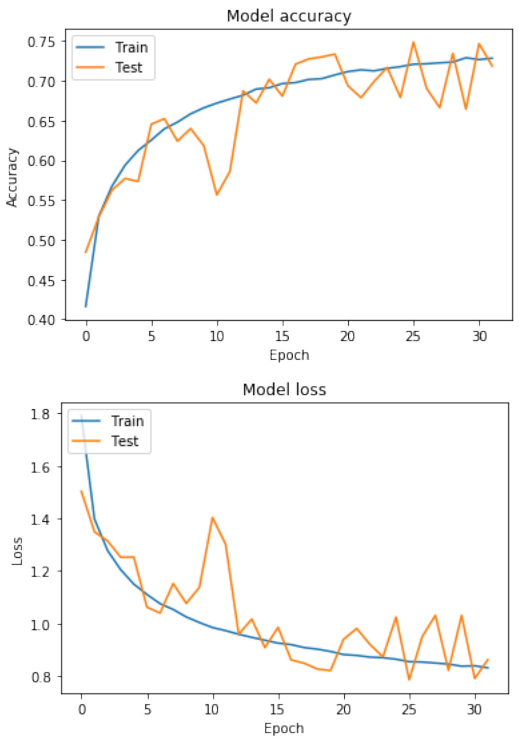


Fig. B.2. The model loss and model accuracy is seen to fluctuate around a certain number. This might be due to a high learning rate(how much the weights are changed based on the gradients). Here, a learning rate 1 is used.

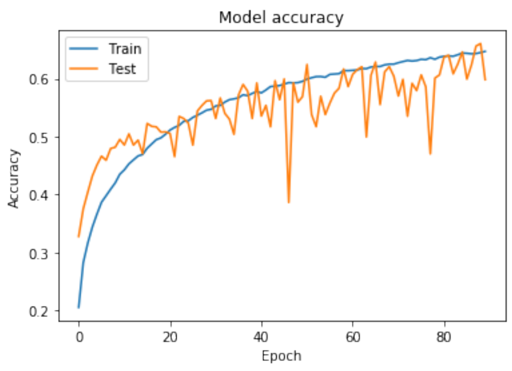


Fig. B.3. 90 epochs and a learning rate of 0.0005.