Image Classification: Dense vs. Convolutional Neural Networks

FYS-STK4155: Project 3

Trygve Leithe Svalheim
Kristian Joten Andersen
github.com/trygvels/FYS-STK4155

December 12, 2019

Abstract

In this project we study the fundamental differences between dense neural networks (DNNs) and convolutional neural networks (CNNs) by constructing them from scratch in order to have full control of the underlying algorithm. For a concise and computationally feasible comparison, we implement a Multilayer Perceptron (MLP) of three layers, and a CNN with a single convolutional layer, feeding into an MLP of two layers.

We compare their ability to classify images from three different datasets: The CIFAR-10 data set with images of 10 different objects and animals, the MNIST data set with images of handwritten numbers, and the Street View House Number (SVHN) data set, images of house numbers taken from google streed view.

On CIFAR-10 our DNN had an accuracy of 0.497 while the CNN had an accuracy of 0.520. On MNIST our DNN had an accuracy of 0.977 and the CNN had an accuracy of 0.981. Lastly, on SVHN the DNN had an accuracy of 0.742 and the CNN had an accuracy of 0.793. We see that the CNN performs equal to or better than the DNN on all data sets, and therefore proves that convolution layers improve image classification ability.

For validation we implemented a structurally equal CNN algorithm using `Keras`, which slightly improved upone the results of the manual CNN. However, employing a more complicated architecture, computationally feasible thanks to the highly optimized `Tensorflow` toolbox allowed us to explore further improvements to the network architecture, such as additional convolutional layers, max-pooling and higher number of filters, both of which drastically improved the classification score.

Contents

1. INTRODUCTION

Up until now we have seen how we can use numerical algorithms in order to do linear regression and polynomial fitting or to do logical regression in order to solve binary classification problems. But, what happens when one has more than one class? What happens when data points are no longer (necessarily) independent?

One area of machine learning that have been heavily researched in the later years are image processing. Most common is the search for specific classes or objects in an image, but it can also be the classification of many images of single objects. One of the areas in modern society this has been implemented in recent time is in cars, where cameras scan the field of view looking for traffic signs and alerts the driver of what it has read.

A common problem for algorithms doing image processing is that the data usually have a high spatial correlation, something that older mathematical algorithms might struggle to deal with. In this project we therefore use two different algorithms, a convolutional neural network (CNN) algorithm and a dense neural network (DNN) algorithm, which we also utilized in the second project of this course [1]. The CNN algorithms have been developed specifically for image processing where the spatial correlation is heavily taken into consideration, where the DNN algorithms are more generalized algorithms meant to solve a broader range of classification problems.

To test if and how much the specific properties of CNNS improve the image classification relative to a similarly structured DNN, we assess the two algorithms' ability to classify images in three different data sets. Two of the data sets concerns classifying numbers from 0 to 9 in either handwritten form or in different fonts, both with limited resolution. The last data set concerns classifying images of objects and animals, each with their own class specific traits and variances, but also including variation of the backgrounds.

Because the aim of this project is to study the underlying algorithms of convolutional neural networks, we have coded both our algorithms from scratch. This gives us full control of how the code works. In order to validate that our code we implement a representative CNN algorithm with equal setup using the `Keras` and **Tensorflow** libraries in Python.

In this report we will first take a closer look at the different data sets we will be working with in Sec. 2. Then, in Sec. 3 we will take a look at the general problems of classification and what parameters we can use in order to measure how well the algorithms perform. After that, in Sec. 4 we will be presenting the algorithms we will use and how they work. Finally, in Sec. 5 we will present the results of the classification of our two algorithms for each data set and discuss strengths and weaknesses of the two.

2. DATA OVERVIEW

In this project we will be looking at three different data sets. All the data sets consist of images of different objects, and it will be the job of our neural network algorithms, which will be presented in Sec. 4 to classify each image correctly (or as many as possible).

The first set is called CIFAR-10 [2]. It is a data set consisting of 60000 color images of objects and animals divided into 10 classes with 6000 images of each. The images have a size of 32×32 pixels, resulting in a $32 \times 32 \times 3$ array of integer numbers on the range [0, 255] (inclusive). The classes and their label in the files can be seen in Tab. I and an example of how some of the images for each class look like can be seen in Fig. 1. According to Alex Krizhevsky [2], the images are mutually exclusive, meaning that no image contains any of the other classes than those specified. Additionally, in order to limit overlap/correlation between automobiles and trucks, pickup trucks are featured in neither of the classes.

The CIFAR-10 data set is divided into a test data set of 10000 images, containing 1000 randomly chosen images of each class, and a training data set which consists of the remaining 50000 images. In the training set the images are distributed randomly, i.e. a certain batch of neighboring indices may contain more images from some classes than others. From figure 1 we clearly see that the backgrounds of some of the classes are changing more than for other classes. It will be interesting to see how this might end up affecting the algorithms' predictions.

The second set is known as the MNIST data set

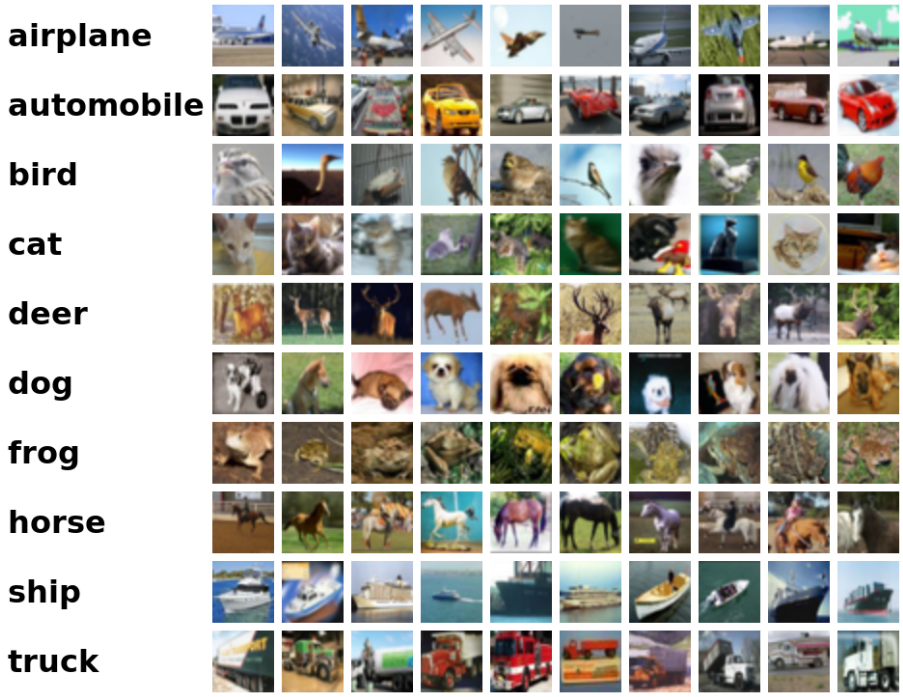

FIG. 1. Example images of the 10 classes in the CIFAR-10 data set. It is clear that the motives and the background may vary a lot within each class. Figure reference [2].

TABLE I. Overview of the CIFAR-10 classes' labeling and data value.

Class label	Class value in input data
Airplane	0
Automobile	1
Bird	2
Cat	3
Deer	4
Dog	5
Frog	6
Horse	7
Ship	8
Truck	9

[3]. It is a data set consisting of 70000 grey-scale images of handwritten numbers from 0 to 9, giving 10 different classes. The data set is, similar to CIFAR-10, split into a test set of 10000 images and a training set of 60000 images. All images are arrays of size $28 \times 28 \times 1$ pixels (1 due to grey-scale), where each pixel takes integer values on the range [0, 255]. 0 means background (white) while 255 means foreground (black), and in all cases the numbers are centered in the image. Furthermore, all images are mutually exclusive wrt. classes, just like the CIFAR-10 data set, meaning no image contain 2 classes. The classes and their label in the data can be seen in Tab. II and an example of how images for each class look like can be seen in Fig. 2.

The third set is the Street View House Numbers (SVHN) data set [5], provided by the Computer Science Department at Stanford University. This data set consists of color images of house numbers gathered from Google Street View. The complete set consists of a training set with 73257 digits, a test set with 26032 digits, and an unassigned set

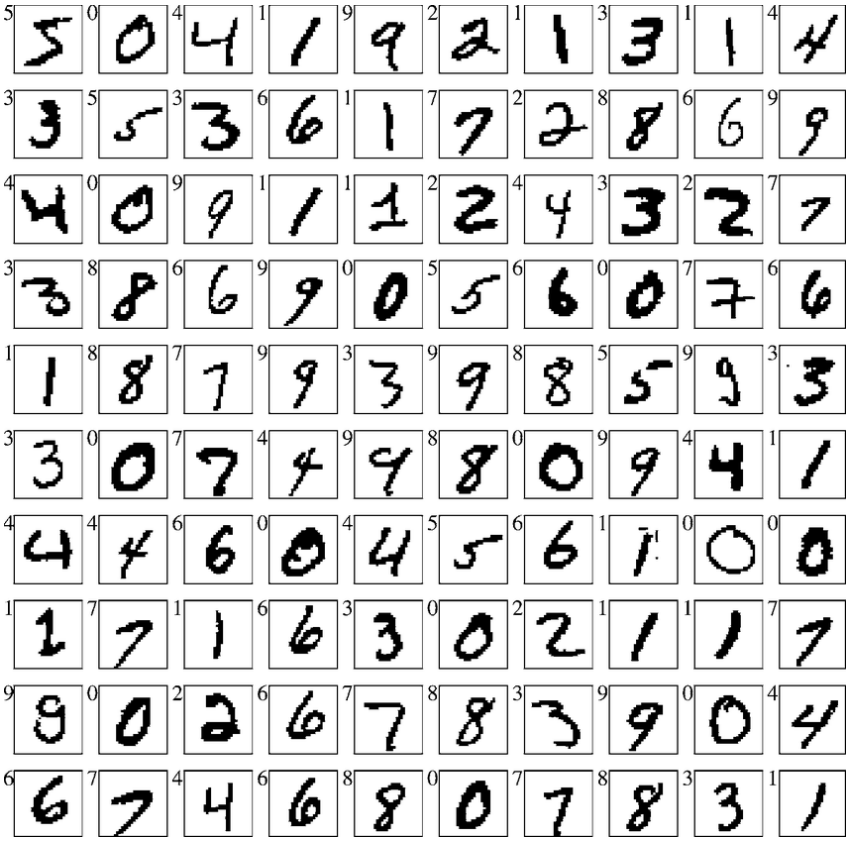

FIG. 2. 100 example images from the MNIST data set. The correct classification of each image is given in the top left corner of each image, One can clearly see that the handwritten numbers vary substantively, and one could easily mistake one for the other, e.g. between the digits '4' and '9'. Figure reference [4].

TABLE II. Overview of the MNIST classes' labeling and data value.

Class label	Class value in input data
'0'	0
'1'	1
'2'	2
'3'	3
'4'	4
'5'	5
'6'	6
'7'	7
'8'	8
'9'	9

of 531131 somewhat less difficult samples to use as additional training data if necessary. The data set comes in two formats, a full scale image format, or a reduced and centered format (much like the MNIST data set) of size $32 \times 32 \times 3$ (3 due to RGB color images). In any case the pixel values are integers in the range 0 to 255, just like the other two data sets. We choose to look at the second format, i.e. the reduced and centered format. This is because of two reasons: First, the images have the same size, which makes coding simpler and the data volume manageable. Secondly, the class of each image is decided by the digit centered in the image, giving us 10 classes of the digits from 0 to 9.

The classes and their label in the data can be seen in Tab. III and an example of how images for each class look like can be seen in Fig. 3. We note that,

FIG. 3. Example images from the SVHN data set. One can clearly see that the images vary in resolution and the amount of distractors on the sides, i.e. other digits. Figure reference [5].

TABLE III. Overview of the SVHN classes' labeling and data value.

Class label	Class value in input data	Class value after read in
'0'	10	0
'1'	1	1
'2'	2	2
'3'	3	3
'4'	4	4
'5'	5	5
'6'	6	6
'7'	7	7
'8'	8	8
'9'	9	9

in contrast with the other two data sets, some of the images may contain "distractors" at the sides, i.e. part of a digit from another class. This unique feature might influence the accuracy of our algorithms, but it gives a much more realistic view of how one would implement classification algorithms for everyday use. In the input data '0' is given the value 10. We change this to 0 when we read in the data, so that '0' has the corresponding class value 0 before we one-hot encode the data set, see. Sec. 4.4 for more information on one-hot encoding.

The training sets of each dataset is very large. While we are fitting we would like to see how well the algorithm is doing without biasing the training with respect to the test data. We therefore take out a subset of 5000 random samples from each of the training sets which we call a *development* or *validation* set. This set we can use to measure how the algorithms prediction power is developing

throughout the training without biasing. We will get back to this in Sec. 4.

3. THEORY

3.1. Classification

Classification differs from normal linear fitting as the output takes discrete values, either the input data belongs to a class or it does not. Instead of outputting a value of a (presumed continuous) function, classification algorithms usually output the probability of the input data belonging to the specified class(es). In a binary classification problem, where the input data is classified to either belonging to the target class or not (single class), that which is most likely of the two would be the classification assigned to the input data. In the case of multiple classes, the classification problem changes slightly. Assuming that the classes are mutually exclusive, i.e. the input data can only belong to one class, then one has to compute the probability of the data belonging to any of the classes, where the sum should equal 100%. In some cases there is a 'none' class, i.e. the data don't belong to any of the other classes, but this is less common when calibrating the algorithm on the training data set, where the goal is for the algorithm to recognize specific classes. As with binary classification, given mutual exclusive data, the class with the highest probability would be the one assigned to the input data.

3.2. Goodness of fit

In order to test how well our algorithms preform, we introduce some of the same goodness of fit parameters as in project 2 [1]. For our classification problem we introduce the following three: area ratio or area under the curve, the F1-score, and accuracy.

Area ratio (AR) is defined from the Lift plot of the data set, see Fig. 4. According to the given model, one sorts the data from most likely to least likely to belong to the target class. Then one plots the number of targets in the first n number of data points, i.e. a cumulative function of the models predictive power. If the model is the best fit, all target data are more likely, given the model, to belong to the target class than any of the data not belonging to the target class. The cumulative function will then rise linearly from the origin to (N_t,N_t), N_t being the number of targets in the data set. If one were to sort the data at random (given that one knows the relative distribution), one would end up drawing a straight line (a baseline) from the origin to (N_d,N_t), as the number of targets in the first n data points are proportional to the relative number of targets in the full data set (i.e. N_t/N_d). The area ratio is then defined as

$$\mathrm{AR} = \frac{\text{Area between the model curve and the baseline}}{\text{Area between the best fit curve and the baseline}}. \tag{1}$$

We note that the area ratio can take values $\mathrm{AR} \in [-1, 1]$, and that negative values means that the model does worse than classifying target/non-target at random, again given that one knows the relative amount of targets.

This can be transformed into the also used goodness-of-fit parameter area under the curve (AUC) which takes the same principles as AR, but plots target along the y-axis and non-targets along the x-axis. The AUC is further defined as the relative ratio under the whole curve so that by transformation

$$\mathrm{AR} = 2(\mathrm{AUC} - 0.5), \tag{2}$$

$$\mathrm{AUC} = 0.5 + \mathrm{AR}/2. \tag{3}$$

We notice that $\mathrm{AR} = 1$ corresponds to $\mathrm{AUC} = 1$, $\mathrm{AR} = 0$ corresponds to $\mathrm{AUC} = 0.5$, and $\mathrm{AR} = -1$ corresponds to $\mathrm{AUC} = 0$.

In a single binary classification problem we can define the data to either be *positive*, i.e. the target, or *negative*, while the predictions from our model we call *predicted positive* (PP) or *predicted negative* (PN) respectively. We can now divide the data into 4 groups: true positive = data that is both positive and predicted positive, true negative = data that is both negative and predicted negative, false positive = data that is both negative and predicted positive, and false negative = data that is both positive and predicted negative.

We now introduce two new terms, the true positive rate (TPR) and the positive predictive value (PPV), which are also known as *recall* and *precision* respectively. They are defined as

$$\mathrm{TPR} = \frac{\sum \text{true positive}}{\sum \text{predicted positive}}, \tag{4}$$

$$\mathrm{PPV} = \frac{\sum \text{true positive}}{\sum \text{positive}}, \tag{5}$$

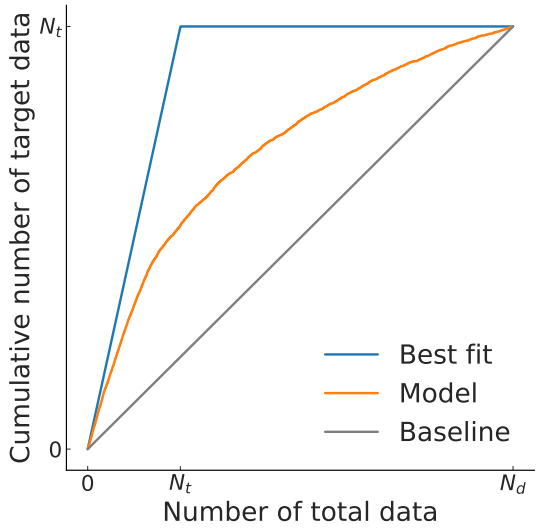

FIG. 4. Lift plot of a model to a data set. N_t is the number of targets in the data set and N_d is the total amount of data in the data set.

and together they define the F_1-score for the *positive* target

$$F_1 = \frac{\mathrm{TPR} \cdot \mathrm{PPV}}{\mathrm{TPR} + \mathrm{PPV}}. \tag{6}$$

We can also utilize this score for classification problems with more than a binary class, which is the case for our data set that we are working with here. What we have to do is simply calculate the F_1-score for each class separately, treating the given target class as the positive outcome and all the other classes as the negative outcome. In the case of multiple classes there is the possibility that no classes have more than 50% likelihood for a given image. In any case we choose to classify any image as its most likely class when computing the F_1-score and also the accuracy.

The *accuracy* (AC) of a model is defined as

$$\mathrm{AC} = \frac{\sum \text{correctly predicted data}}{\sum \text{total population}}, \tag{7}$$

which is the ratio of the total population the model correctly classifies. The accuracy, like many other statistical measures, is known under different names, one being the correct classification rate (CCR).

All these parameters have their own strengths and weaknesses. The area ratio is a good measure if one wants to optimize the use of resources and only act on parts of the data. In our case, if we were to only find images with say the number 4, we would like our algorithms to pick out as many images with the number 4 before any images with other numbers. A high area ratio is therefore good for the model. However, a high area ratio doesn't necessarily mean that our model is accurate. If our model predicts all data to belong (i.e. have the highest probability) to the target data, only that the true target data has slightly higher probabilities, we loose out on all the other classes, giving a very low accuracy if the majority of the data is negative. This also affects the PPV and therefore also the F_1-score of the target class. As the accuracy looks at the whole data set, and the F_1-score looks at only one of the classes, the latter has more details.

In our work, we would like our models to do well with all goodness-of-fit parameters for all classes. Since we have several classes, it is more important that a class with higher occurrence is weighted more, therefore we introduce a weighted average F_1-score of the total data set. This weighted F_1-score is the sum of all individual class' F_1-score multiplied with the class' density in the whole population, i.e.

$$\overline{F}_1 = \frac{\sum_i F_{1,i} \cdot N_i}{\sum_i N_i}, \tag{8}$$

where $F_{1,i}$ is the F_1-score and N_i is the population of class i. We see that we may also use this weighting for the AR as well, simply by substituting $F_{1,i}$ with AR_i. Additionally, by comparing the scores of our test and training data we will see whether we over-fit or under-fit our model. Over-fitting is indicated by large differences between test and training data while under-fitting is indicated by both test and training data not performing well.

The main difference between the accuracy/F_1-score and the AR/AUC is that while the first only take into account the most likely classification of individual data point (e.g. images) the latter take into account the likelihood of all data within any given class. In other words, accuracy only measures if the strongest prediction of each individual input

data point is correct, while the area ratio measures the strength of the likelihoods across all input data. The cost/loss function of a logical/classification algorithm looks primarily on the magnitude of the likelihood predicted for the true classification for the given data point, which is somewhat more similar to the area ratio than the accuracy/F_1-score. To reiterate, the accuracy and F_1-score are good when describing the performance on the whole data set, while the AR/AUC is good when one later wants to act upon a subset of the data set, where by example each data point acted upon has some cost related to it.

4. METHODS AND ALGORITHMS

We will now present the two algorithms we have used to classify the data sets presented in Sec. 2, namely dense neural networks (DNN) and convolutional neural networks (CNN). First we will talk about activation functions, then we'll explain the general outline of each algorithm and give the more specific setups of the algorithms for each of the three data sets at the end of each section.

4.3. Activation functions

An important aspect of neural networks is activation functions. These functions are carefully chosen depending on the type of data in question. The activation function is what gives Neural networks complexity, as their non-linear nature prevents the network from becoming a linear mapping from input to output, or a polynomial of degree one. In principle, one can use any non-linear function that is continuous, bounded and non-constant. Another important property of the activation function is that it needs to be differentiable, if not, we will not be able to backpropagate as we will see later.

In project 2 [1] we presented a number of activation functions, and we will recapitulate them here. The most common activation function, especially for classification problems, is the *sigmoid* function

$$g(x) = \frac{1}{1+e^{-x}}, \tag{9}$$

which truncates the input between 0 and 1. Historically, this activation function has been very common, but after the advent of deep neural nets, a major flaw was discovered, namely the *vanishing gradient problem*. The vanishing gradient problem is the problem of gradients going to zero as a result of the partial derivatives truncated to small values being multiplied l times, which for a high number of hidden layers goes to zero. This means the first layers learn a lot slower, and in the worst case scenario, not at all. For our purposes, using more than one hidden layer, this may prove to be an issue, and we therefore do not use this.

Another related activation function is the hyperbolic tangent function or *tanh*.

$$f_{\mathrm{t}}(x) = \tanh(x) = \frac{\mathrm{e}^{x} - \mathrm{e}^{-x}}{\mathrm{e}^{x} + \mathrm{e}^{-x}}. \tag{10}$$

which is a scaling of a sigmoid, centering it at zero, so that the function truncates values between -1 and 1. It has a steeper derivative so it updates gradients faster, and because it is centered on zero it is easier to optimize. However, calculating exponentials are more computationally expensive, and it still has the problem of vanishing gradient and thus we do not use it here.

On of the most popular activation functions these days is the rectified linear units (ReLU), takes the input and returns all values above zero,

$$f_{\mathrm{ReLU}}(x) = \max(0, x). \tag{11}$$

It does not have vanishing gradient problem and is very computationally cheap. But on the other hand, a neuron which goes to zero has little chance of recovery, which results in the *dying ReLU* problem. Additionally, because it has no upper bound, ReLU also has the issue of exploding gradients, crashing the weight update for high learning rates.

Another variation of the relu is the is the exponential linear unit

$$f_{\mathrm{ELU}}(x;\alpha) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\mathrm{e}^{x} - 1) & \text{if } x < 0. \end{cases} \tag{12}$$

which has been designed to perform better on classification problems. While the ELU generally performs better than a ReLU, it requires more calculations, and is therefore more computationally costly. In circumstances where a network does not suffer from vanishing gradients but still runs slow, a simple ReLU might be preferable.

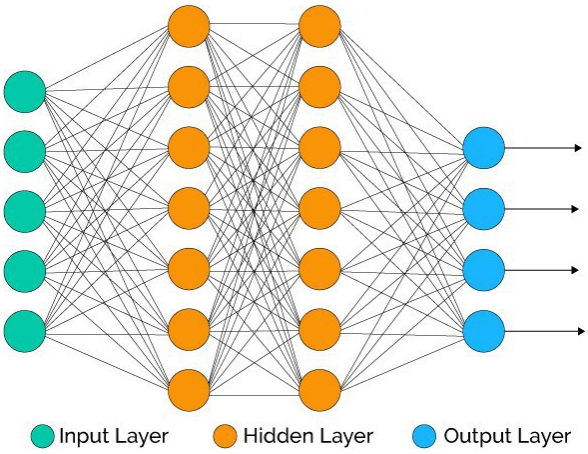

FIG. 5. Illustration of a dense neural network with an input layer with 5 nodes, two hidden layers with 7 nodes and an output layer with 4 nodes. Figure reference [6].

4.4. Dense Neural Networks

In the second project [1] of this course we described a dense neural network for a binary classification problem. We will therefore only recapitulate the relevant properties of this type of neural networks and refer the interested reader to the other report for some more details. A dense neural network is build up by an *input* layer, an *output* layer, and one or more *hidden* layers between the input and output layers. Generally, an algorithm of a DNN can be written

$$\text{Input} \rightarrow \text{Hidden} \cdot h \rightarrow \text{Output}, \tag{13}$$

where h is denoting the number of hidden layers. An illustration can be seen in Fig. 5

The input layer is per definition the input data points, or the image in the case for our data sets, and has a fixed size depending on the data. The output layer is just a vector, with length equal to the number of classes, which returns the input's probabilities of belonging to the given classes. The hidden layers are a layers made up of a range of *nodes* and are not bound by any dimension, meaning that they can take any size or any number of nodes.

The special property of DNNs is that every node of each layer is connected to the all the nodes in the previous layer (and therefore also to the nodes in the next layer). In each layer, each node computes the response of the previous layer's output to a set of initially random *weights* and *bias* before it passes it through an *activation gate*. An activation gate is a non-linear function that transforms the data in some suitable way, for example truncating it between values of 1 and -1. This activation gate is crucial in the neural network as it prevents the network from becoming a simple linear mapping from input to output.

Mathematically, each node in a hidden layer performs the following calculation:

$$z_k^l = \sum_{j=1}^{N^{l-1}} \left(w_{jk}^l a_j^{l-1}\right) + b_k^l, \tag{14}$$

$$a_k^l = g(z_k^l), \tag{15}$$

where w_{jk} is the weight, a the output of the previous layer, and b is the bias. The superscript denotes the layer and the subscripts j and k denote the indices of nodes in respectively layer $l-1$ and l. Lastly N denotes the number of nodes in the given layer, i.e. layer $l-1$, and g is the activation function.

As each node has a unique weight for each of the connections (i.e. nodes) to the previous layer, the number of weights can be quite large, e.g. in the CIFAR-10 data sets the images has sizes $32 \times 32 \times 3$ giving 3072 weights for each of the nodes in the first hidden layer. This limits the number of nodes in the first hidden layer, as the processing power and memory requirement would quickly become too large for most computers.

In the last step of the DNN we pass on the output of the last hidden layer through the output layer with another set of weights and biases. The last step then applies another activation function, which transforms the data into the desired shape, which for a classification problem, might be a probability that the input data belongs in a certain category. In the case of a multi-class classification problem, this activation function should return the probabilities of each class between 0 and 1, with the sum normalized to 1. One such function is the Softmax function

$$g(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \tag{16}$$

where the sum runs over all z values of the output layer. The derivative of the Softmax function with respect to an arbitrary z_k can be shown to be

$$\frac{\partial g(z_j)}{z_k} = g(z_j)(\delta_{jk} - g(z_k)), \tag{17}$$

where δ is the Dirac delta function. In the case of binary classification this reduces to $i = j$.

We want to define a cost function for our classifications. First we redefine an images true class $y_i = 1, 2, 3...$ through one-hot encoding so that in classification problem with 10 classes

$$y_i = 1 \to \hat{y}_i = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$
$$y_i = 4 \to \hat{y}_i = (0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$$

We have that the probability of input data (image) i belonging to a class c is

$$P(y_{ci} = 1 \,|\, \mathcal{D}_i, \hat{\theta}) = \frac{e^{z^L_{ci}}}{\sum_{k=1}^{K} e^{z^L_{ki}}}, \tag{18}$$

where we have used the Softmax function in Eq. (16). $\mathcal{D}_i$ is the i-th input image, $\hat{\theta}$ is the complete set of weights and biases in all nodes, and z^L_{ci} and z^L_{ki} is the values of node c and k in the final layer L. The i in the right side of the equation denotes that the values are due to the i-th input. Defining the joint likelihood likelihood function of all data as

$$P(\mathcal{D} \,|\, \hat{\theta}) = \prod_{i=1}^{N} \prod_{c=1}^{K} \left[P(y_{ci} = 1)\right]^{y_{ci}}, \tag{19}$$

where N is the number of input images and K is the number of classes. We then define our cost function as

$$\mathcal{C}(\hat{\theta}) = -\ln P(\mathcal{D} \,|\, \hat{\theta}). \tag{20}$$

Using the mathematical properties of logarithms, we get

$$\mathcal{C}(\hat{\theta}) = -\sum_{n=1}^{N} \sum_{c=1}^{K} y_{ci} \ln P(y_{ci} = 1). \tag{21}$$

If written on vector form, we get

$$\mathcal{C}(\hat{\theta}) = -\sum_{n=1}^{N} \hat{y}_i^T \cdot \ln P(\hat{y}_i = 1). \tag{22}$$

Backpropagation

In order to fit the weights and biases to minimize our cost function we need to know how the cost function change with respect to each of them. Before we write down the equations, we note that $P(y_{ci} = 1)$ is the same as a^L_{ji}, i.e. the output of the activation function of node j in the last layer $l = L$. From our second project [1] we now have for the weights of the final layer

$$\begin{aligned}\frac{\partial \mathcal{C}}{\partial w^L_{jk}} &= \frac{\partial \mathcal{C}}{\partial a^L_k}\left[\frac{\partial a^L_k}{\partial w^L_{jk}}\right] \\ &= \frac{\partial \mathcal{C}}{\partial a^L_k}\left[\frac{\partial a^L_k}{\partial z^L_k}\frac{\partial z^L_k}{\partial w^L_{jk}}\right] \\ &= \frac{\partial \mathcal{C}}{\partial a^L_k} g'_L(z^L_k) a^{L-1}_j, \end{aligned} \tag{23}$$

$$\tag{24}$$

where k is the node index in the layer $l = L$, and j is the index of the node in the previous layer that node k is weighting w^l_{jk} with. g'_l is the partial derivative of the activation function (of the given layer) with respect to z, and should ideally be easily defined. Similarly for the bias we have

$$\begin{aligned}\frac{\partial \mathcal{C}}{\partial b^L_k} &= \frac{\partial \mathcal{C}}{\partial a^L_k}\left[\frac{\partial a^L_k}{\partial b^L_k}\right] \\ &= \frac{\partial \mathcal{C}}{\partial a^L_k}\left[\frac{\partial a^L_k}{\partial z^L_k}\frac{\partial z^L_k}{\partial b^L_k}\right] \\ &= \frac{\partial \mathcal{C}}{\partial a^L_k} g'_L(z^L_k). \end{aligned} \tag{25}$$

The partial derivative of the Cost function with respect to a^L_k is

$$\begin{aligned}\frac{\partial \mathcal{C}}{\partial a^L_k} &= -\frac{\partial}{\partial a^L_k}\sum_{c=1}^{K} y_c \ln(a^L_c) \\ &= -\frac{y_k}{a^L_k}. \end{aligned} \tag{26}$$

In the partial equations above we have omitted the sum over the input data, i.e. there should be a sum over the N inputs and so each y, a and z should have an input specific summation index as well, i.e. i as in Eq. (21).

From equation (17) we get that

$$g'_L(z^L_k) = g(z^L_k)\left(1 - g(z^L_k)\right) = a^L_k(1 - a^L_k), \tag{27}$$

as $j = k$ in Eq. (17). combining all equations, this leaves us with

$$\frac{\partial \mathcal{C}}{\partial w_{jk}^L} = -\sum_{i=1}^{N} y_{ki}(1 - a_{ki}^L)\, a_{ji}^{L-1} \tag{28}$$

$$\frac{\partial \mathcal{C}}{\partial b_k^L} = -\sum_{i=1}^{N} y_{ki}(1 - a_{ki}^L), \tag{29}$$

where we now have included the sum over input data images i. We see that the derivatives are all equal to or less than zero, as y and a take values between 0 and 1.

Now we need to take a look at the other hidden layers and compute the derivatives of the cost function with respect to the weights and biases in those layers, starting with the layer closest to the final layer and working backwards.

$$\begin{aligned}
\frac{\partial \mathcal{C}}{\partial w_{jk}^l} &= \frac{\mathcal{C}}{\partial z_k^l}\frac{\partial z_k^l}{\partial w_{jk}^l} \\
&= \frac{\partial \mathcal{C}}{\partial a_k^l}\frac{\partial a_k^l}{\partial z_k^l}\frac{\partial z_k^l}{\partial w_{jk}^l} \\
&= \left(\sum_m \frac{\partial \mathcal{C}}{\partial z_m^{l+1}}\frac{\partial z_m^{l+1}}{\partial a_k^l}\right)\frac{\partial a_k^l}{\partial z_k^l}\frac{\partial z_k^l}{\partial w_{jk}^l} \\
&= \left(\sum_m \frac{\partial \mathcal{C}}{\partial z_m^{l+1}} w_{km}^{l+1}\right) g_l'(z_k^l)\, a_j^{l-1},
\end{aligned} \tag{30}$$

where the partial derivative of the cost function with respect to z_m^{l+1} is known from the calculation of that layer as it would have been computed earlier. For the biases we get

$$\begin{aligned}
\frac{\partial \mathcal{C}}{\partial b_k^l} &= \frac{\partial \mathcal{C}}{\partial z_k^l}\frac{\partial z_k^l}{\partial b_k^l} \\
&= \frac{\partial \mathcal{C}}{\partial a_k^l}\frac{\partial a_k^l}{\partial z_k^l} \\
&= \left(\sum_m \frac{\partial \mathcal{C}}{\partial z_m^{l+1}}\frac{\partial z_m^{l+1}}{\partial a_k^l}\right)\frac{\partial a_k^l}{\partial z_k^l} \\
&= \left(\sum_m \frac{\partial \mathcal{C}}{\partial z_m^{l+1}} w_{km}^{l+1}\right) g_l'(z_k^l),
\end{aligned} \tag{31}$$

where we have used that z is linear w.r.t. b. Again, we have omitted the sum over the input data images, so a summation is needed like in Eqs. (28) and (29).

Once we reach the input layer we have calculated all derivatives of the cost function with respect to the weights and biases, and we then update each of them using the delta rule like we did in project 2 [1]

$$w_{jk,\text{new}}^l = w_{jk,\text{old}}^l - \lambda \frac{\partial \mathcal{C}}{\partial w_{jk,\text{old}}^l} \tag{32}$$

$$b_{k,\text{new}}^l = b_{k,\text{old}}^l - \lambda \frac{\partial \mathcal{C}}{\partial b_{k,\text{old}}^l}, \tag{33}$$

where λ is the learning rate. As might be clear, if one have more input images the cost function and therefore it's derivative might become large. To normalize this we choose to add a normalization factor $1/N$ to the cost function so that we don't have to change λ if we change the number of input data images. Note that this will also affect equations (21), (28), and (29) which also gains a $1/N$ factor.

When we have finished updating the weights and biases we run the network forward again, producing new estimates before again doing the backpropagation until we are satisfied with the results.

ADAM optimization

When running through the network and updating weights and biases, a static learning rate as described using stochastic gradient descent in Eq. (32) might prove inefficient. In recent years, a lot of progress has been made in developing optimization algorithms cut down on training time in neural networks. Because training is by far the most time consuming part of we want to make sure that we don't take any unnecessary steps, or that we step to far or too short. In wandering through parameter space we want to take long steps when gradients are small, and short steps when gradients are high. Furthermore, we want to avoid local minima and converge on the optimal solution.

Adaptive Moment Estimation or Adam [7], for short, is a popular method for calculating adaptive learning rates on a per-feature basis. Like RMSprop, one of its predecessors, it calculates the average of past squared gradients v, while at the same time keeping an exponentially decaying average of past gradients m like momentum gradient descent. Momentum gradient descent adds a velocity term to the vanilla gradient descent, which decays and

accellerates like a ball on a slope. Adam on the other hand, behaves like a heavy ball with friction, which prefers flat minima in the error surface [8].

For each feature we calculate the adam parameters

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \end{aligned} \tag{34}$$

where g_t is the gradient at step t. Both m and v are initialized to zero. The authors of the paper have observed that the parameters are biased towards zero, and must therefore be corrected at every step before updating features θ

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{35}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t. \tag{36}$$

The authors of the paper reccomend setting $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. While *alpha* remains a free parameter to be tuned.

Specification of the DNN algorithm parameters

Due to the large number of weights needed for each node in the first hidden layer, we have chosen to use the ReLU function, Eq. (11), as our activation function as it will limit the large computational requirement of calculating exponential functions. Furthermore, it has a well defined gradient, being equal to 1 if the argument is larger than zero, and 0 if the argument is smaller than zero.

As mentioned in Sec. 2, both the CIFAR-10 and the SVHN data bases have input images of size $32 \times 32 \times 3$, giving 3072 weights per node in the first hidden layer. For the MNIST data base it is smaller, as the images are $28 \times 28 \times 1$, giving 784 weights per node in the first hidden layer.

Due to the large number of weights we chose to use $h = 3$ number of hidden layers, with the number of nodes in each hidden layer being $N_h = (128, 128, 64)$, for all data sets. The output layer had 10 nodes, corresponding to the number of classes in all data sets.

We are dealing with large datasets in this work, and using all inputs during training will make for a very slow algorithm. In our second project [1] we were also dealing with a large data set. We solved that by assuming that the cost function and its derivative can be represented by a subset, a.k.a. a batch, of the input data. We do so in this work as well, drawing a random batch from the training data for each loop of the training of the algorithm. The batch size has to be large enough to be representative of the training set while not being too large so that the training takes too long. We chose to use $N = 128$ as our batch size when training our DNN algorithm.

One training loop of the algorithm can be generalized as

1. Draw a random batch of size N_b from the training set.

2. Forward-propagate the batch through the DNN and compute the class probabilities.

3. Compute the cost and the accuracy of the batch.

4. Backward-propagate tha batch through the DNN, computing the partial derivative of the cost function with respect to all the weights and biases.

5. Update the weights and biases using ADAM optimization.

To decide on when to stop training, we would like to see how well our algorithm performs while training. The batch size is too small for this and it is also biased. In the data overview section, Sec. 2, we mentioned that we split off a development set of size 5000 from each of the training sets in order to have a set which is unbiased of both the training and the test data. Thus, every 100 loop (or step) in the training we forward-propagate the development set and compute the total accuracy of this set. This will have to be done in batches, as there are too many numbers (weights, biases, nodes etc.) to store in memory for the entire set. It is important to note that we do not back-propagate using this development set, as we want it to stay unbiased of the training.

The weights and bias of each node in layer l of the DNN algorithm were initialized using the following

equations.

$$\eta^l = \frac{2}{n^{l-1}}, \tag{37}$$

$$w^l_{jk} = \mathcal{N}\left(0, \sqrt{\eta^l}\right), \tag{38}$$

$$b^l_k = 0, \tag{39}$$

where n^l is the number of nodes in layer l ($l = 0$ is the input layer) and $\mathcal{N}(\mu, \sigma)$ is a normal distribution with mean μ and standard deviation σ.

4.5. Convolutional Neural Networks

A convolutional neural network, or ConvNet or CNN for short, is a type of neural networks that is especially good at picking up spatial correlation in the input data, making it ideal for image classification. The core of the network, which differs it from dense neural networks, is the *convolution operation* (hence its name). Instead of each node being connected to all nodes in the previous layer, each with their own weights, a node in a CNN is only connected to a subset of the nodes in the previous layer. In addition, all nodes in a layer share the same weights and it is the convolution of these weights over the previous layer that picks up on the spatial structure in the data.

Before going into details, a general CNN can be written

$$\begin{aligned}\text{Input} \to &[\text{conv} \to \text{activ}] \cdot n \to \\ &[\text{FC} \to \text{activ}] \cdot k \to \text{Output},\end{aligned} \tag{40}$$

where n is the number of convolution layers and activation layers and k is the number of fully connected and activation layers before the output layer. There are several additions that can be made to the full convolutional network. When working with convolutional layers such as the one implemented in this project, it is commonplace to accompany it by a "Pooling layer", which takes the output of the feature layers from the convolutional layer and chooses the highest activations. This reduces the dimensionality of the network, because we typically chose the 2 maximum values of the pooling grid (Max-pooling), which halves the size of the layer. Furthermore, pooling contributes to what is known as local spatial invariance, in that it shifts pixels around inside the two-by-two grid. There are also other improvements that can be made to convolutional networks such as batch normalization and dropout, which we will not study in detail. Once again, we stress that the focus of this project is to study the specific increase in accuracy by switching out a dense layer with a convolutional layer.

The *convolution layer* is the layer where one convolves the previous layer with a set of weights. A set of weights is known as a *filter*, and each layer may have an arbitrary number of filters. The filters have a size $(W_\text{f}, H_\text{f}, D_\text{f})$ corresponding to width, height and depth respectively. For the input image and the convolved layers we have likewise $(W_\text{in}, H_\text{in}, D_\text{in})$ and $(W_\text{c}, H_\text{c}, D_\text{c})$, and we define the number of filters for each convolvolution layer as as N_f. The three dimensional structure of the layers in CNN is crucial, as the network is built to pick up on spatial correlations/structure. The convolution operation consists of scanning the filter(s) over the previous layer in the width and height dimensions (i.e. the spatial dimensions), weighting the nodes with the filter(s), and at each position outputting the sum plus a bias to a node in the convolution layer (as is done for all nodes in a DNN). Figure 6 shows an illustrated example of this. What the figure doesn't show is that the the weighting is done on the whole depth of the previous layer, giving $D^1_\text{f} = D_\text{in}$ and $D^l_\text{f} = N^{l-1}_\text{f}$ ($l > 1$). The second of these conditions derives from the fact that each filter gives a 2D layer of convolved nodes and therefore we stack the layers from each filter in the depth direction so that $D^l_\text{c} = N^l_\text{f}$. Therefore, all nodes in a depth column "looks" at the same spatial part of the input, only through different filters.

As the edges of the filter can not go outside the defined edges of the input layer (or the previous convolved layer) then the width and height of the convolved layers would gradually decrease through the layers of the CNN if the filter size is larger than 1, which it always should be. To counteract this one may use what is known as *zero-padding*, where one adds one or more zero to the (spatial) edges of the input/previous layer before convolving with the filters. If we denote the thickness of the zero-padded layer as Z the width and height of the convolved layer will be [10]

$$\mathrm{W_c}^l = W^{l-1}_\text{c} - W^l_\text{f} + 2Z + 1, \tag{41}$$

$$\mathrm{H_c}^l = H^{l-1}_\text{c} - H^l_\text{f} + 2Z + 1, \tag{42}$$

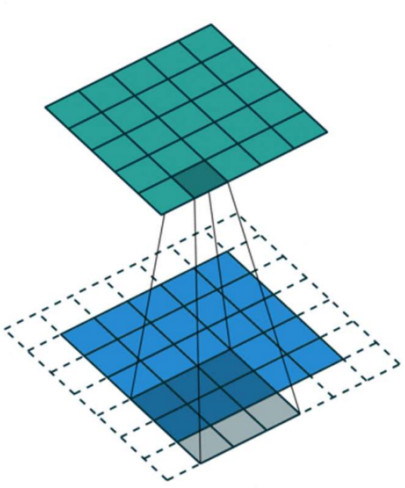

FIG. 6. An illustration of the convolution in a convolutional neural network. The input/previous layer (blue) is scanned with a filter (grey) resulting in a node value in the convolved layer (green). The white squares are zero-padding around the input/previous layer. Figure reference [9].

where in the case $l = 1$, $W_{\mathrm{f}}^0 = W_{\mathrm{in}}$. In the case of a filter size of $3 \times 3 \times D_{\mathrm{f}}$, a zero padding of $Z = 1$ would return a convolved layer of equal spatial size as the previous layer. In figure 6 this is the case, where the zero-padded values are outlined as white dotted squares.

Another convolution parameter of the CNN is the *stride* S, which is the number of indices or nodes one moves the filter at a time when convolving. In other words, $S = 1$ means to move the filter one step/index at a time, while $S = 2$ would mean to move 2 steps and so on. In effect one skips over $S - 1$ indices for each step, resulting in a smaller convolved layer when $S > 1$. Including the stride, one can show that Eqs. (41) and (42) become [10]

$$\mathrm{W_c}^l = \left(W_{\mathrm{c}}^{l-1} - W_{\mathrm{f}}^l + 2 \cdot Z\right)/S^l + 1, \tag{43}$$

$$\mathrm{H_c}^l = \left(H_{\mathrm{c}}^{l-1} - H_{\mathrm{f}}^l + 2 \cdot Z\right)/S^l + 1. \tag{44}$$

We now see that not all values for width, height, zero padding, and stride give integer values for the height and width of the convolved layer, e.g. for some random values $(6-3+2\cdot 1)/2+1 = 3.5$ which would not work. There are some solutions to solve this problem if the filter size specified is the one that is preferable, adding zero-padding on one side or truncating the input are those that are mostly used. We see that a stride $S > 1$ is effectively downgrading the input/previous layer. This might be useful for large images where details are more likely to be separated by several pixels (i.e. nodes) than if the image was smaller.

After convolving the previous layer, the convoluted values are passed through an activation gate, just like the fully connected layers in a DNN, which doesn't change the size of the layer. After the activation gate, the convolved/activation layer may be passed to a new convolution layer or to the first fully connected layer of the final structure of the CNN (a DNN like structure).

A third option is to pass the activation/convolved layer to a *pooling layer* which downsamples the layer, usually by performing a mean or a maximum on a subset of $P_1 \times P_2$ nodes (in spatial dimensions). This is an other way of reducing the size of the data being passed through the CNN without having to use a stride $S > 1$, though this does make backpropagating a little more technical, as one needs to know which nodes in the previous layer that contributed to the pooled node. The pooled layer may then be passed back to a convolution layer or to the first fully connected layer.

Like dense neural networks the last hidden layers of a CNN are fully connected layers with activation gates in-between. In practice, a CNN is simply one or more convolutional layers which outputs into a DNN. Which means that the output layer also has dimensions equal to the number of classes and an activation function that brings the output values to the desired range. Like with our DNN algorithm, we choose to use the Softmax function here as well, see Eq. 16.

The filter sizes and the number of filters are more often sampled. A filter of spatial size 3×3 will see correlations between 9 pixels at a time, and connecting pixels to a maximum of 24 other pixels (all that are 1 or 2 pixel widths and/or heights away). A filter of size 5×5 will correlate 25 pixels at a time, maximum connect 1 pixel to neighbors ≤ 4 pixel widths and/or heights away. It might be easier to think of this in one dimension. If we now were to set two 3×3 filters after one another, i.e. two convolution layers as seen in Fig. 7, then the first layer connects 3 nodes/pixels from the input layer, while the second layer connects 5 nodes from

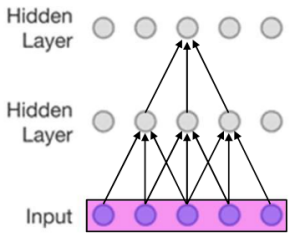

FIG. 7. An illustration of the "field of view" for nodes in a hidden layer of a CNN using a filter sie of 3. Figure reference [9].

the input layer. Additionally, the nodes in the input layer closer in spatial position to the node in the second layer contribute more than nodes further to the side, effectively weighting local correlations more. If taking this to two dimensions, two 3×3 filters have $2 \cdot 9 = 18$ weights and 2 biases, while one $5x5$ filter has 25 weights and 1 bias. In other words, we get more local spatial correlations/structure, or we recover more complexity, with several layers of smaller filters than with one large filter, and as a bonus there are fewer weights and biases which may speed up the algorithm.

CNNs come in many different shapes and forms, with several hyper parameters like filter size and numbers, stride, pooling and more. These are all tuned depending on the data in question, and many studies have been done on optimal network architectures. In recent years, there have been many important developments in network architectures, such as discovering the benefits of deep neural networks using many hidden layers.

Forward and backward propagation in a convolution layer

We will now look at the mathematical equations that makes a CNN. First, we define

$$r_{\mathrm{w}} = (W_{\mathrm{f}} - \mathrm{mod}(W_{\mathrm{f}}, 2))/2, \tag{45}$$

$$r_{\mathrm{h}} = (H_{\mathrm{f}} - \mathrm{mod}(H_{\mathrm{f}}, 2))/2, \tag{46}$$

where 'mod' is the modulo operator, so that for a $3 \times 3 \times D_{\mathrm{f}}$ filter one gets $r_{\mathrm{h}} = r_{\mathrm{h}} = 1$. Then, each node in a convolution layer performs the following calculation for each filter.

$$z^l_{jk} = \sum_{d=1}^{D^l_{\mathrm{f}}} \sum_{m=-r_{\mathrm{w}}}^{r_{\mathrm{w}}} \sum_{n=-r_{\mathrm{h}}}^{r_{\mathrm{h}}} w^l_{dmn} a^{l-1}_{d,j+m,k+n} + b^l, \tag{47}$$

$$a^l_{jk} = g(z^l_{jk}), \tag{48}$$

where d runs over the depth, and m and n runs over the spatial dimensions. j and k are also denoting spatial dimensions, and l denotes the layer. It is assumed that the values of a, where the indices fall outside the input/previous layer, are zero (proper zero-padding). Due to the number of indices to keep track of, we have omitted an index for the filter for the weights w and bias b, as well as the convolved sum z, a is the activation layer and g is the activation function. This would be repeated until one reaches a pooling layer or the first fully connected layer in the DNN like structure of the CNN, see Eq. (40). In the data sets we work with the spatial size of the images is quite small, maximum 32×32. Therefore, we choose to not include any pooling layers in our algorithm as this simplifies the backpropagation, in addition to the fact that we get a more pure comparison to the DNN. We also choose to use stride $S = 1$ which again simplifies even more.

Backpropagation in a CNN differs from that of DNN. The convolution makes it a little more complicated, though there are substantially fewer weights and biases of which one has to find the derivative of the cost function with respect to. To do the derivations necessary we change to a more code like notation (Python). A second note is that we do not include indices for the set of input images that one would need to sum over, but is is fairly straight forward to do, therefore we omit it for better readability. We define for the weights and bias

$$\mathtt{w}[\mathtt{f}, \mathtt{d}, \mathtt{m}, \mathtt{n}], \tag{49}$$

$$\mathtt{b}[\mathtt{f}], \tag{50}$$

where `f` represents the filter index, `d` represents the depth index (of the filter), `m` and `n` represent the spatial indices (of the filter). Further for the convolved values z^l

$$\mathtt{z}[\mathtt{f}, \mathtt{j}, \mathtt{k}], \tag{51}$$

where $\mathtt{j}$ and $\mathtt{k}$ represent the spatial indices, and $\mathtt{f}$ represents the depth (corresponding to the filter index). Finally, for the input to the convolution layer a^{l-1}

$$\mathtt{a}[\mathtt{d},\mathtt{j},\mathtt{k}], \tag{52}$$

where $\mathtt{j}$ and $\mathtt{k}$ represent the spatial indices, and $\mathtt{d}$ represents the depth (corresponding to the depth of each filter). Note that the letters/parameters used for indexing will change positions from what is defined above in the following equations!

When we take the partial derivative of the cost function wrt. an arbitrary parameter y of w, b or a we get

$$\frac{\partial \mathcal{C}}{\partial y} = \sum_{z_x} \frac{\partial \mathcal{C}}{\partial z_x}\frac{\partial z_x}{\partial y}, \tag{53}$$

where z_x are all the convolved values in the layer that depend on y.

We start by taking the partial derivatives of the biases b. As all z-values in the f-th depth of the convolution layer depend on $\mathtt{b}[\mathtt{f}]$, we need to sum over the entire spatial extent of the f-th depth of the layer, thus

$$\begin{aligned}\frac{\partial \mathcal{C}}{\partial \mathtt{b}[\mathtt{f}]} &= \sum_j \sum_k \frac{\partial \mathcal{C}}{\partial \mathtt{z}[\mathtt{f},\mathtt{j},\mathtt{k}]}\frac{\partial \mathtt{z}[\mathtt{f},\mathtt{j},\mathtt{k}]}{\partial \mathtt{b}[\mathtt{f}]} \\ &= \sum_j \sum_k \frac{\partial \mathcal{C}}{\partial \mathtt{z}[\mathtt{f},\mathtt{j},\mathtt{k}]},\end{aligned} \tag{54}$$

where we have used that z is linear wrt. b. Moving on to the weights, all z-values in the f-th depth of the convolution layer depend on $\mathtt{w}[\mathtt{f},\mathtt{d},\mathtt{m},\mathtt{n}]$ so we need to sum over the entire spatial extent of the f-th depth of the layer, and we get

$$\begin{aligned}\frac{\partial \mathcal{C}}{\partial \mathtt{w}[\mathtt{f},\mathtt{d},\mathtt{m},\mathtt{n}]} &= \sum_j \sum_k \frac{\partial \mathcal{C}}{\partial \mathtt{z}[\mathtt{f},\mathtt{j},\mathtt{k}]}\frac{\partial \mathtt{z}[\mathtt{f},\mathtt{j},\mathtt{k}]}{\partial \mathtt{w}[\mathtt{f},\mathtt{d},\mathtt{m},\mathtt{n}]} \\ &= \sum_j \sum_k \frac{\partial \mathcal{C}}{\partial \mathtt{z}[\mathtt{f},\mathtt{j},\mathtt{k}]}\mathtt{a}[\mathtt{d},\mathtt{j}+\mathtt{m},\mathtt{k}+\mathtt{n}].\end{aligned} \tag{55}$$

In equations (54) and (55) we are left with one unknown; the partial derivative of the cost function with respect to z. We now see if we can solve this in a similar way

$$\begin{aligned}\frac{\partial \mathcal{C}}{\partial \mathtt{z}[\mathtt{f},\mathtt{j},\mathtt{k}]} &= \frac{\partial \mathcal{C}}{\partial \mathtt{a}^{\mathtt{l}}[\mathtt{f},\mathtt{j},\mathtt{k}]}\frac{\partial \mathtt{a}^{\mathtt{l}}[\mathtt{f},\mathtt{j},\mathtt{k}]}{\partial \mathtt{z}[\mathtt{f},\mathtt{j},\mathtt{k}]} \\ &= \frac{\partial \mathcal{C}}{\partial \mathtt{a}^{\mathtt{l}}[\mathtt{f},\mathtt{j},\mathtt{k}]}g'(\mathtt{z}[\mathtt{f},\mathtt{j},\mathtt{k}]),\end{aligned} \tag{56}$$

where we have specified a^l because it gives us the gradient of the activation function. The partial derivative of the cost function w.r.t. a^l is given by

$$\frac{\partial \mathcal{C}}{\partial \mathtt{a}^{\mathtt{l}}[\mathtt{f},\mathtt{j},\mathtt{k}]} = \sum_{z_x^{l+1}} \frac{\partial \mathcal{C}}{\partial z_x^{l+1}}\frac{\partial z_x^{l+1}}{\partial \mathtt{a}^{\mathtt{l}}[\mathtt{f},\mathtt{j},\mathtt{k}]}, \tag{57}$$

where z_x^{l+1} is all z-values in the next layer that is depending on a^l. If it is the last convolved layer before the fully connected layer, all z^{l+1} depend on a^l and the latter derivative in Eq. (57) is

$$\frac{\partial \mathtt{z}^{\mathtt{l}+\mathtt{1}}[\mathtt{i}]}{\partial \mathtt{a}^{\mathtt{l}}[\mathtt{f},\mathtt{j},\mathtt{k}]} = \mathtt{w}^{\mathtt{l}+\mathtt{1}}[\mathtt{i},\mathtt{f},\mathtt{j},\mathtt{k}]. \tag{58}$$

Here we assume that the fully connected layer has one dimension, and the partial derivative of the cost function with respect to z_i^{l+1} is found through the same equations as described for DNNs, see back-propagation in Sec. 4.4.

If we are not at the last convolved layer before the fully connected layer, the z^{l+1} who depend on $\mathtt{a}^{\mathtt{l}}[\mathtt{f},\mathtt{j},\mathtt{k}]$ are not straight forward to compute. In any case, all filters will scan $\mathtt{a}^{\mathtt{l}}[\mathtt{f},\mathtt{j},\mathtt{k}]$, but some of the filters will only only use parts of their filter on a and it is all depending on filter size and zero-padding. Stride $S > 1$ would complicate this even further. Assuming $S = 1$ and zero-padding that conserves the spatial size, one would for Eq. (57) get the following, omitting the layer indices l for a and $l+1$ for z, w, and the filter hyperparameters

$$\begin{aligned}&\frac{\partial \mathcal{C}}{\partial \mathtt{a}[\mathtt{f},\mathtt{j},\mathtt{k}]} = \\ &= \sum_{d=1}^{D_\mathrm{f}} \sum_{m=-r_\mathrm{w}}^{r_\mathrm{w}} \sum_{n=-r_\mathrm{h}}^{r_\mathrm{h}} \frac{\partial \mathcal{C}}{\partial \mathtt{z}[\mathtt{d},\mathtt{j}+\mathtt{m},\mathtt{k}+\mathtt{n}]} \cdot \\ &\qquad \frac{\partial \mathtt{z}[\mathtt{d},\mathtt{j}+\mathtt{m},\mathtt{k}+\mathtt{n}]}{\partial \mathtt{a}[\mathtt{f},\mathtt{j},\mathtt{k}]} \\ &= \sum_{d=1}^{D_\mathrm{f}} \sum_{m=-r_\mathrm{w}}^{r_\mathrm{w}} \sum_{n=-r_\mathrm{h}}^{r_\mathrm{h}} \frac{\partial \mathcal{C}}{\partial \mathtt{z}[\mathtt{d},\mathtt{j}+\mathtt{m},\mathtt{k}+\mathtt{n}]} \cdot \\ &\qquad \mathtt{w}[\mathtt{d},\mathtt{f},\mathtt{j}-\mathtt{m},\mathtt{k}-\mathtt{n}],\end{aligned} \tag{59}$$

where $\partial \mathcal{C}/\partial \mathtt{z}^{\mathtt{l}+\mathtt{1}}[\mathtt{d},\mathtt{j}+\mathtt{m},\mathtt{k}+\mathtt{n}]$ is defined from back-propagation of earlier layers, except for where $(j+m, k+n)$ is outside of the spatial dimensions of the layer (i.e. in the zero-padding) where it is

by definition zero. In the case of a single convolution layer, this last part would not be used in back-propagation.

Finally, when all gradients have been computed we update the weights and biases of the CNN in the same manner as for the DNN, see ADAM optimization in Sec. 4.4, before running the network in forward-propagation again. Like with DNN we stop the training when we reach convergence of the network.

Specification of the CNN algorithm parameters

In this work we implement CNN and DNN algorithms from scratch without the use of any other Python libraries like **Scikit-Learn** or **Tensorflow** to perform computations for us. This way we have *full* control of all the steps in the algorithm, but as a downside the computations speed is slow. The code architecture is inspired by mandatory excercise 1 in in9400, written by Ole-Johan Skrede [11] but modified to a working convolutional network. All propagations algorithms etc. have been implemented by us.

Furthermore, coding in Python is not ideal for large data sets, especially when running through multiple for loops as is nessecary for the backpropagation algorithm. Therefore, we have implemented these in its pures form, in order to be interprated by **numba**'s *jit* compiler to try to speed up the forward- and back-propagation. However, the code is still significantly slower then what it could be if implemented in say C++/fortran with proper array multiplications. All in all this limits the number of filters, filter sizes, and the number of hidden layers we can use in our CNN. This is not an issue to us, however, as the focus of the project is to study the simplest type of convolutional network vs. a dense.

We chose to (initially) implement our CNN network with only one convolution layer. In comparison to the DNN algorithm, this layer will effectively replace the first layer of the DNN. This we do as a benchmark test to see if a single convolutional layer can pick up on some complexity from the input images that a fully connected layer can not, and therefore increase the accuracy of the classification relative to the DNN algorithm. Having only one layer does speed up the code, as the fully connected layers are faster to process, especially with the back-propagation. The drawback of using only one layer is one does not nessecarily pick up the hierarchical structure often present in these data types, ie. structure within structure, and the local extent of these complexities are limited to the filter size.

The filter size we chose for the convolution layer $(N_\mathrm{f}, W_\mathrm{f}, H_\mathrm{f}) = (3, 3, 3)$, i.e. we used 3 filters in total of spatial size 3×3. This is far from optimal, one would typically use a filter number on the order of 10s, such as 32, but for our study, this has proven too time consuming. The depth of each filter matches the depth of the input layer, see Sec. 2 for further information. Additionally, we used 1 layer of zero-padding so that our first convolved layer would end up with the same dimensions as the input image. Furthermore, we chose to use 2 fully connected layers with the same number of nodes as the last 2 layers of the DNN algorithm, i.e. $N_h = (128, 64)$. The output layer had of course 10 nodes, equal to the number of classes. Also, like the DNN algorithm, we used the ReLU function, Eq. (11), as the activation function for all layers except the output layer which used the Softmax function, Eq. (16).

The training loop of the CNN is identical to that of the DNN, see last part of Sec. 4.4, although the forward- and back-propagation through the convolution layers differ. To best compare the two algorithms we use the same batch size for the subset of training data for each training step.

5. RESULTS AND DISCUSSION

5.6. CIFAR-10 results

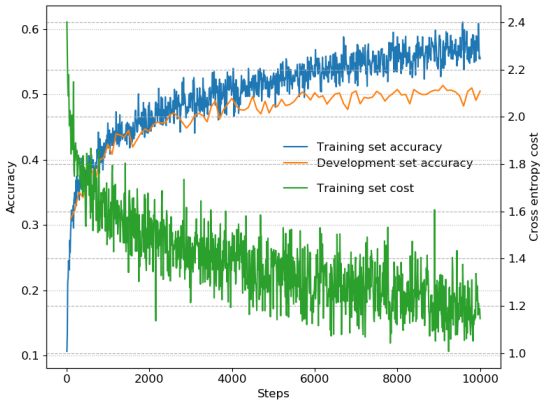

FIG. 8. Running cost and accuracy for the training of the dense neural network (DNN) on the CIFAR-10 data set. The training set values are calculated from the batch used in each training step, while the development accuracy is calculated on the whole 5000 image development set.

With the setups specified in the methods section, Sec. 4, we let each network train for 10000 batches of 128 images from the CIFAR-10 data before the development set started to converge. The running accuracy of the training batches and the development set as well as the running cost for both networks can be seen in Figs. 8 and 9. We also implemented a CNN algorithm using `Keras` with the same layer and filter structure as our CNN. The training was slightly different as the code computes the values of each data set's accuracy after an epoch, usually corresponding to the training set's size. The accuracy of the development set converged after 10 epochs and the running values for

TABLE IV. Overview of the accuracy for the algorithms on three subsets of the CIFAR-10 data set.

Data set	DNN	CNN	Keras
Training	0.5849	0.7425	0.7734
Development	0.5050	0.5278	0.5522
Test	0.4969	0.5202	0.5460

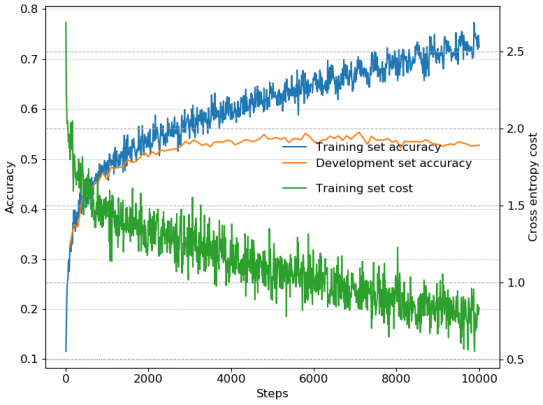

FIG. 9. Running cost and accuracy for the training of the convolutional neural network (CNN) on the CIFAR-10 data set. The training set values are calculated from the batch used in each training step, while the development accuracy is calculated on the whole 5000 image development set.

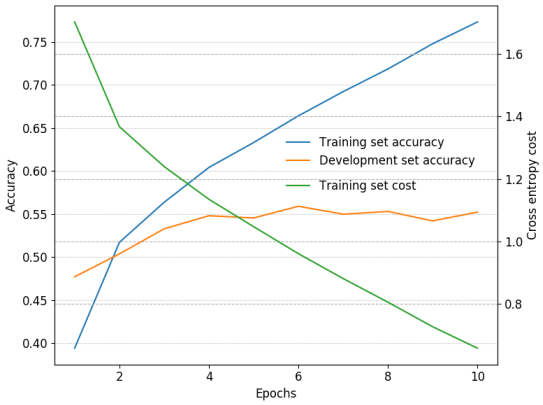

FIG. 10. Running cost and accuracy for the training of the `Keras` implemented convolutional neural network (CNN) on the CIFAR-10 data set. The training set values are calculated from the whole 45000 image training set for each epoch, while the development accuracy is calculated on the whole 5000 image development set.

TABLE V. Overview of the goodness-of-fit parameters for the algorithms on the CIFAR-10 test data. The columns show the class labels, number of input data for each class, and F_1-score and area ratio for each algorithm.

Class label	Amount	F_1-score			Area Ratio		
		DNN	CNN	Keras	DNN	CNN	Keras
Airplane	1000	0.5292	0.5963	0.6333	0.8068	0.8286	0.8448
Automobile	1000	0.6044	0.5995	0.6090	0.8619	0.8455	0.8762
Bird	1000	0.3117	0.4035	0.4347	0.6406	0.6796	0.6930
Cat	1000	0.3353	0.3354	0.4088	0.6584	0.6602	0.6863
Deer	1000	0.4286	0.4795	0.4937	0.7146	0.7409	0.7573
Dog	1000	0.4132	0.4275	0.4103	0.7093	0.7219	0.7156
Frog	1000	0.5524	0.5753	0.6191	0.8222	0.8263	0.8581
Horse	1000	0.5442	0.5476	0.6018	0.7744	0.7832	0.8122
Ship	1000	0.6104	0.6472	0.6823	0.8498	0.8718	0.8945
Truck	1000	0.5597	0.5500	0.5922	0.8336	0.8021	0.8368
Weighted average		0.4889	0.5162	0.5485	0.7672	0.7760	0.7975

each epoch can be seen in Fig. 10. The final accuracy of each of the algorithms on the different data sets can bee seen in Tab. IV, while the F_1-score and area ratio of the test data can be seen in Tab. V.

We see that the accuracy of the DNN algorithm is about 2.3% lower than our CNN algorithm, which again is 2.6% lower than the `Keras` CNN algorithm. The difference between the algorithms are significant, but looking at the trend of the development set, which acts like a test set during training, we see that the CNN algorithm averages a higher accuracy (and lower variance) than the DNN over the last 4000 steps, see Figs. 8 and 9. Due to this we are confident that the difference in accuracy of the test set is significant. The bigger question is whether or not we are over-fitting the algorithms to the training sets, by which the difference between the accuracy of the training set and the development set is a clear indication of. Taking a look at the accuracy of the `Keras` CNN algorithm during training, see Fig. 10, shows the same indications as the other algorithms, namely that the training set is being over-fitted while the accuracy of the development set flattens out. The accuracy of the development set is slightly better than our CNN, which to us indicates that `Keras`' training algorithm is doing a better job than our CNN, since the structure of the CNNs is the same.

In table V we see the same tendencies as for the accuracy. The DNN is performing the worst on the test set, the `Keras` CNN is performing the best, while our CNN algorithm lies somewhere in the middle, but the differences are only ∼2-3%. A more interesting result, which is shared by all algorithms, is that the F_1-score and area ratio per class is a lot higher for certain classes (e.g. automobiles and trucks) than for others (e.g. birds and cats). This shows us that some classes are easier to recognize and classify than others. Why is this the case? There may be several factors: position, colors, object shapes, intraclass variance, viewing angle, symmetry, backgrounds, and more. For image examples see Fig. 1.

With positioning we mean where the class object it positioned in the image. If it is to the left-/right/top/bottom and so on. Since the DNN algorithm is spatially dependent, i.e. it relies on the fact that the features of a class is in the same position (pixel), moving it around can impact this code substantially. Furthermore, it doesn't see spatial correlation between pixels which may further decrease the prediction power. As a CNN algorithm can pick up on local features, this might not be as big of a problem.

Next on the list; colors, object shapes, and intraclass variance (the first two a part of the latter) may also throw off the algorithms. If the size of an object or its color change a lot between classes, then the class may be harder to pinpoint for the

algorithms. One example would be ships vs. birds, where the birds tend to change more in size. This also impacts the background, another thing on the list, because the algorithms might classify the images based on the background rather than the object itself if the background for one of the classes are typically different from all other classes. Again, one example here is the ships, where one would usually have water in the background. Other classes, like the animals have more diverse backgrounds in general (see. Fig. 1). We're not saying backgrounds are everything, but having varying background for all classes would be important for tackling this potential 'bias'.

Lastly, we mentioned symmetry and viewing angle. Some objects are very symmetric, others less so. Also, the angle from which one usually views the objects vary between the classes. Ships are repeating in this comparison, where they are usually more symmetric and more often viewed from the same angle(s) than most others.

5.7. MNIST results

FIG. 11. Running cost and accuracy for the training of the dense neural network (DNN) on the MNIST data set. The training set values are calculated from the batch used in each training step, while the development accuracy is calculated on the whole 5000 image development set.

For the MNIST data set we let each algorithm train for 2000 steps before the accuracy of the de-

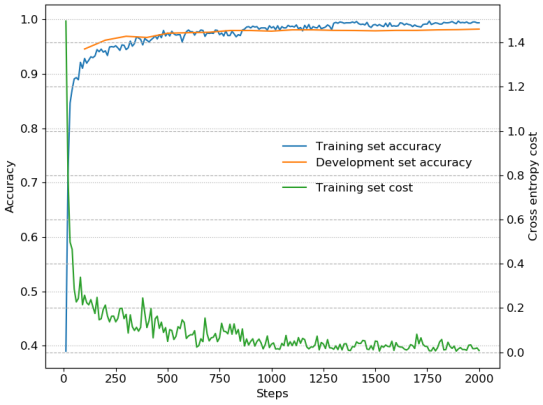

FIG. 12. Running cost and accuracy for the training of the convolutional neural network (CNN) on the MNIST data set. The training set values are calculated from the batch used in each training step, while the development accuracy is calculated on the whole 5000 image development set.

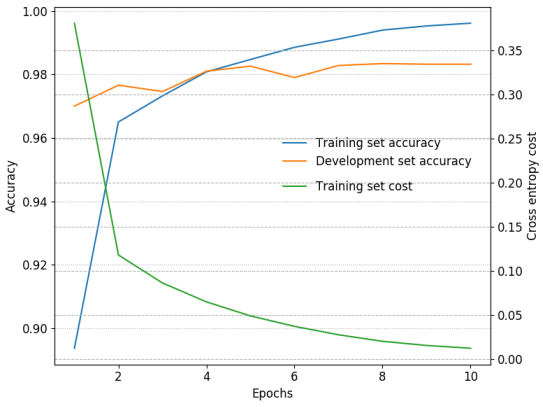

FIG. 13. Running cost and accuracy for the training of the `Keras` implemented convolutional neural network (CNN) on the MNIST data set. The training set values are calculated from the whole 55000 image training set for each epoch, while the development accuracy is calculated on the whole 5000 image development set.

TABLE VI. Overview of the accuracy for the algorithms on three subsets of the MNIST data set.

Data set	DNN	CNN	Keras
Training	0.9905	0.9969	0.9961
Development	0.9766	0.9822	0.0832
Test	0.9772	0.9806	0.9795

velopment converged. The running accuracy of the training batches and the development set as well as the running cost for both algorithms can be seen in Figs. 11 and 12. The `Keras` CNN algorithm was run for 10 epochs, like was done for CIFAR-10, and the evolution of the training can be seen in Fig. 13. The final accuracy of each of the algorithms on the different data sets can bee seen in Tab. VI, while the F_1-score and area ratio of the test data can be seen in Tab. VII.

For this data sets, all networks obtain very good results. None of the networks are less than 97.7% accurate. Furthermore, for this analysis, our network also scored higher than both the `Keras` implementation and the DNN with a score of 98.06%, which is close to the result obtained by a more complex CNN algorithm in the Deep Learning lecture by Michael Nielsen (2019) [12] which got 0.9967% (9967/10000) of test data correct.

The reason for the high scores on this data set stems from many different factors. First, the background is all white with no distractions (unlike the housing number data set) and a single color channel. Second, all numbers are centered which diminishes the need for spatial invariance in training and boosts the DNNs classification ability. Third, there is very little intraclass variance; each number looks very similar, which can be seen by the fact that all classes have almost identical F1 score in table VII, unlike the different types of dogs in the CIFAR data set, for example.

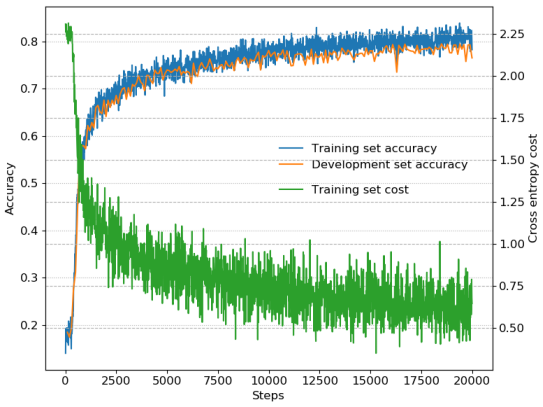

FIG. 14. Running cost and accuracy for the training of the dense neural network (DNN) on the SVHN data set. The training set values are calculated from the batch used in each training step, while the development accuracy is calculated on the whole 5000 image development set.

5.8. SVHN results

For the SVHN data set we let each algorithm train for 20000 steps before the accuracy of the development converged, which was the longest number of training steps for any of the data sets (suggesting that the Adam learning rate could be optimized further). The running accuracy of the training batches and the development set as well as the running cost for both algorithms can be seen in Figs. 14 and 15. The `Keras` CNN algorithm was run for 20 epochs, twice as what was done for the other two data sets, and the evolution of the training can be seen in Fig. 16. The final accuracy of each of the algorithms on the different data sets can bee seen in Tab. VIII, while the F_1-score and area ratio of the test data can be seen in Tab. IX.

From the data we see that the algorithms' performance is ok, but not great. 74-81% accuracy for all, though the CNN algorithms are close (1.6% difference) and so do better than the DNN algorithm. Talking about the F_1-score and area ratio the case is the same. `Keras` CNN performs best, followed closely by our CNN, and finally the DNN algorithm is again ∼3-5% behind. Unlike the CIFAR-10 data and more like MNIST, these scores are not

TABLE VII. Overview of the goodness-of-fit parameters for the two algorithms on the MNIST test data. The columns show the class labels, number of input data for each class, and F_1-score and area ratio for each algorithm.

Class label	Amount	F_1-score			Area Ratio		
		DNN	CNN	Keras	DNN	CNN	Keras
'0'	980	0.9878	0.9878	0.9863	0.9997	0.9997	0.9997
'1'	1135	0.9908	0.9921	0.9890	0.9998	0.9998	0.9998
'2'	1032	0.9788	0.9811	0.9703	0.9986	0.9995	0.9992
'3'	1010	0.9746	0.9773	0.9802	0.9989	0.9993	0.9997
'4'	982	0.9774	0.9848	0.9837	0.9993	0.9997	0.9997
'5'	892	0.9700	0.9746	0.9831	0.9991	0.9995	0.9992
'6'	958	0.9752	0.9801	0.9854	0.9991	0.9995	0.9990
'7'	1028	0.9741	0.9785	0.9759	0.9991	0.9994	0.9990
'8'	974	0.9737	0.9721	0.9710	0.9987	0.9993	0.9989
'9'	1009	0.9673	0.9757	0.9698	0.9974	0.9992	0.9992
Weighted average		0.9772	0.9806	0.9795	0.9990	0.9995	0.9994

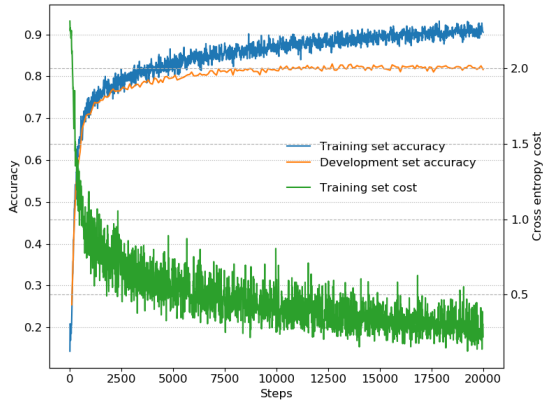

FIG. 15. Running cost and accuracy for the training of the convolutional neural network (CNN) on the SVHN data set. The training set values are calculated from the batch used in each training step, while the development accuracy is calculated on the whole 5000 image development set.

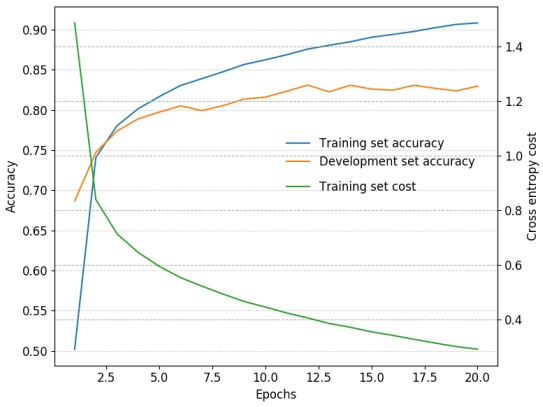

FIG. 16. Running cost and accuracy for the training of the `Keras` implemented convolutional neural network (CNN) on the SVHN data set. The training set values are calculated from the whole 68257 image training set for each epoch, while the development accuracy is calculated on the whole 5000 image development set.

TABLE VIII. Overview of the accuracy for the algorithms on three subsets of the SVHN data set.

Data set	DNN	CNN	Keras
Training	0.7851	0.9155	0.9084
Development	0.7654	0.8164	0.8296
Test	0.7417	0.7929	0.8079

very different between the classes, though they differ somewhat more than the MNIST set. Also, if we look at Fig.14 it might look like the algorithm is not optimally trained after 20000 steps, but would need more steps to converge. If this would result in slightly better estimates is not certain, but only running the training for longer would prove this. Simultaneously, as the variance of the development set is a couple percent, the accuracy on the test data might still vary, and the low accuracy of the test data for the DNN might be on the lower side of this fluctuation. It is worth mentioning that the CNN algorithm already started to converge after 10000 steps, which is half of that of the DNN and probably even less if the DNN needs even more. The CNN also has a smaller variance in the accuracy of the development set through the training as well.

What we do see is that the representation of each class, i.e. the number of images of each class in the training data, is very uneven. Consequently, it looks like the classes with the highest representation perform better than the others. If the distribution is similar in the training data, then based on the way we are training the algorithms, the more represented classes will be sampled more often and therefore the algorithms learn to classify these 'more' than the others. This correlation looks to be more than just coincidence. A quick check shows that the difference in relative representation of a class between the training set and the test set is less than 10% (of the relative representation), which underlines the assumption made.

As for the other two datasets, we also need to look at the background and intraclass variances when considering the performance of the algorithms on the SVHN dataset. From figure 3 we see that the background do change between images, but they are still fairly homogeneous with only the color changing. It is maybe more likely that the different shapes (i.e. fonts) of the digits are throwing off the algorithms more than the background. The position of the digits are more likely not a reason for the non-optimal accuracy, as the digits have been centered in the images (like the MNIST dataset).

5.9. General discussion

In this section let us compare the algorithms' performance on the different datasets. It is clear that the MNIST is the easiest dataset to classify, with almost perfect accuracy for each algorithm. On the contrary, the CIFAR-10 dataset proves to be the hardest to classify, with the best algorithm managing to classify just over half of the images. From what we have seen, it looks like the the backgrounds and complexity of the class objects' spatial characteristics are heavily impacting the accuracy of the algorithms. This does not come as a shock, as extracting higher complexity from an image usually would require more complex structure in the algorithms. Improvements to the algorithms will be discussed later, in Sec. . The dataset that used the longest time to train was the SVHN dataset, taking more than twice that of CIFAR-10. This might be because the complete data set was larger and it would take longer to sample the whole set.

The common things we see across all three datasets is that the our CNN algorithm performs close to or as well as the `Keras` CNN algorithm, and that they both perform better than our DNN algorithm by ∼1-5% depending on the type of measure. Another common characteristic is that all algorithms tend to predict not just the total datasets but aslo each class equally well or equally bad. This is best seen in Tab. V where both the F_1-score and the area ratio vary between the classes, but are almost consistent across algorithms. This shows that all algorithms are struggling with the same class characteristics or background contamination of the images. Still, from the results we have seen we can say with confidence that the convolution layer of the CNN proves better than a simple fully connected layer in the DNN to classify images, though the difference for the layer we implemented was limited, giving at best a 5% increase in accuracy.

TABLE IX. Overview of the goodness-of-fit parameters for the two algorithms on the SVHN test data. The columns show the class labels, number of input data for each class, and F_1-score and area ratio for each algorithm.

Class label	Amount	F_1-score			Area Ratio		
		DNN	CNN	Keras	DNN	CNN	Keras
'0'	1744	0.7016	0.7796	0.7940	0.9059	0.9400	0.9522
'1'	5099	0.8388	0.8671	0.8779	0.9424	0.9596	0.9633
'2'	4149	0.7995	0.8384	0.8680	0.9164	0.9409	0.9577
'3'	2882	0.6675	0.7320	0.7555	0.8647	0.8876	0.9060
'4'	2523	0.8003	0.8071	0.8249	0.9276	0.9418	0.9479
'5'	2384	0.6970	0.7799	0.7927	0.8943	0.9330	0.9339
'6'	1977	0.6720	0.7301	0.7507	0.9043	0.9244	0.9371
'7'	2019	0.7454	0.8088	0.8314	0.9201	0.9537	0.9598
'8'	1660	0.6001	0.6953	0.7104	0.8749	0.8966	0.9189
'9'	1595	0.6669	0.7124	0.6961	0.8940	0.9126	0.9325
Weighted average		0.7420	0.7926	0.8091	0.9095	0.9332	0.9442

5.10. Possible improvements

As mentioned in the introduction we set up a simple, and structurally equal CNN using `Keras` and `Tensorflow`. This allowed us to compare our results to a different implementation of the same algorithm. Furthermore, the fact that `Tensorflow` is highly optimized and runs a lot quicker (On top of being extremely easy to use), allows us to explore the possible improvements that could have been made to our own implementation.

As mentioned in the discussion on activation functions, we know that the elu activation is designed to work well on classification problems. Therefore, we ran the `Keras` network using elu activation on the most challenging dataset; CIFAR-10, which only very slightly improved the result. We have decided to not use elu in our implementation because of the computational cost increase, and the fact that the problem it is designed to solve (vanishing gradients), is not an issue with such a small network.

Although the focus of this project is to explore the addition of a single convolutional layer from scratch, we have explored the addition of another convolutional layer as well as max-pooling using `Keras` on CIFAR-10. The setup was thus one conv. layer with 5 filters of size 5×5, followed by a max-pooling layer with a grid of $(2, 2)$, followed by another convolutional layer with 3 filters of size 3×3 and another max-pooling layer. This again improved the accuracy of the network by at least $\sim$10% (it was stopped short), but drastically slowed down the network, as expected. The improvement from the addition of additional convolutional layers and max-pooling is no surprise but a welcome additional result.

Finally, the most impactful improvement that was allowed by the `Keras` implementation, and perhaps most relevant, was the ability to increase the number of filters. By running the simple CNN using one convolutional layer and no max-pooling with 32 (3×3) filters, as opposed to only 3, we increased the classification accuracy by $\sim$9%. Which is not surprising considering the fact that we expected 3 filters of $(3 \times 3 \times 3)$ or 81 weights, to hold information on 10 different categories. However, the small amount of filters still managed to beat the the dense layer.

6. CONCLUSION

In our work we have studied how the addition of a convolutional layer can improve the classification accuracy of a neural network on three different image data sets.

By constructing a convolutional neural network (CNN) on its simplest form, we have tested it against a simple dense, fully connected neural network and proved that without tuning and suboptimal parameters, we obtain better scores on three independent data sets by three different statistical measures, namely F_1-score, area ratio and accuracy; (0.516, 0.793, 0.981), (0.776, 0.933, 0.9995), and (0.520, 0.793, 0.981) respectively for the (CIFAR-10, SVHN, MNIST) datasets, compared to (0.489, 0.742, 0.977), (0.767, 0.910, 0.9990), and (0.497, 0.742, 0.977) for a dense neural network DNN).

We can therefore with confidence conclude that the addition of convolutional layers improves the classification ability of neural networks on image data. This stems from the unique ability of the filter weights to capture spatially correlated features in images.

REFERENCES

[1] Trygve Leithe Svalheim and Kristian Joten Andersen. *Classification and Regression: Neural Networks vs. Logistic/linear Regression*. Nov. 2019.

[2] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. *The CIFAR-10 dataset*. 2010. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[3] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *The MNIST database of handwritten digits*. 2013. URL: http://yann.lecun.com/exdb/mnist/.

[4] Dennis Decoste and Nello Cristianini. "Training Invariant Support Vector Machines". In: *Machine Learning* 46 (Aug. 2003). DOI: 10.1023/A:1012454411458.

[5] Yuval Netzer et al. *The Street View House Numbers (SVHN) Dataset*. 2011. URL: http://ufldl.stanford.edu/housenumbers/.

[6] Michael Sheinman. *THyperparameters for Neural Networks*. 2018. URL: https://medium.com/udacity-pytorch-challengers/hyperparameters-for-neural-networks-c50ab565ee3d.

[7] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[8] Martin Heusel et al. *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. 2017. arXiv: 1706.08500 [cs.LG].

[9] Tollef Jahren. *IN5400 Machine learning for image classification. Lecture 5 : Convolutional neural networks*. 2019. URL: https://www.uio.no/studier/emner/matnat/ifi/IN5400/v19/material/week5/in5400_2019_week5_convolutional_nerual_networks.pdf (visited on 12/10/2019).

[10] Fei-Fei Li, Justin Johnson, and Serena Yeung. *CS231n Convolutional Neural Networks for Visual Recognitions*. 2019. URL: http://cs231n.github.io/convolutional-networks/ (visited on 12/10/2019).

[11] Ole-Johan Skrede. *Part of mandatory assignment 1 in IN5400 - Machine Learning for Image analysis*. 2019. URL: https://www.uio.no/studier/emner/matnat/ifi/IN5400/v19/index.html (visited on 12/12/2019).

[12] Michael Nielsen. *Neural Networks and Deep Learning, Chapter 6: Deep Learning*. 2019. URL: http://neuralnetworksanddeeplearning.com/chap6.html (visited on 12/11/2019).