An Object Oriented Implementation of a Convolutional Neural Network in Python*

Erling Nupen
Ole Kristian Rustebakke
Brage Andreas Trefjord
Jan Tor Fredrik Stensson

June 7, 2024

Abstract

In this project we have developed object oriented python code for a convolutional neural network (CNN) from scratch, i.e., without the use of existing machine learning packages. We have tested our network on two separate image datasets, the MNIST dataset containing handwritten numbers, and the CIFAR-10 dataset containing colored images of animals and vehicles. We got a maximum validation accuracy of 91% on the MNIST when training on 6 000 images with learning rate $\eta = 0.1$ and regularization parameter $\lambda = 0.1$, using the LRELU activation function, 20 convolutions filters of size 3×3, and a hidden fully-connected layer of 20 neurons. The CIFAR-10 dataset performed poorly for all parameters, with a maximum accuracy of only 14%, likely because we had to use a very limited number of only 500 training images as the inclusion of color channels made the model training very computationally heavy. If we had more time it would have been beneficient to implement the option to choose stride and padding in the convolution layer. In addition, it would be interesting to further optimize the code, for instance using a low-level language such as C++, and see if this would give better results for the CIFAR-10 dataset.

Contents

1 Introduction **3**

2 Theory **3**
2.1 The Machine Learning Task 3
2.2 The CNN Architecture 4
2.2.1 Layers 5
2.2.2 Convolution 5

*GitHub Repository: `https://github.com/Bragit123/FYS5429/tree/main/Project`

2.2.3 Pooling . . . 8
2.2.4 Fully Connected . . . 8
2.2.5 Flattening . . . 9
2.3 Training the Neural Network . . . 9
2.3.1 Backpropagation . . . 13
2.3.2 Backpropagation in the Fully Connected Layer . . . 13
2.3.3 Backpropagation in the Pooling Layers . . . 14
2.3.4 Backpropagation in the Convolutional Layer . . . 14

3 Method **17**
3.1 The CNN Architecture . . . 17
3.2 Feed Forward . . . 18
3.3 Backpropagation . . . 18
3.4 Flatten . . . 19
3.5 Optimization . . . 19

4 Results and Discussion **19**
4.1 The MNIST Dataset . . . 19
4.2 CIFAR-10 dataset . . . 25

5 Conclusion **27**

6 Appendix **27**
6.1 Gradient Calculation . . . 27
6.2 Convolution . . . 28
6.3 Optimized convolution . . . 28
6.4 CNN backpropagation derivation . . . 31
6.4.1 Gradient with respect to the bias . . . 31
6.4.2 Gradient with respect to the kernel . . . 32
6.4.3 Gradient with respect to the input . . . 32

1 Introduction

An important part of deep learning is image analysis. It can for example be used in self driving cars [1], and is even actively used in hospitals for X-ray images [2]. In our report we looked at the popular Cifar10 [3] and MNIST [4] datasets.

The MNIST handwritten digits dataset contains 60 000 training images and 10 000 test images, of size 28×28 pixels in grayscale, and the Cifar10 dataset contains 50 000 training images and 10 000 test images of size 32×32, and 3 color channels, each pixel has a value between 0 and 255. The cifar10 dataset contains images of different objects in 10 different classes, like cats, dogs etc. in different motions and poses. Every class has an equal amount of training and test data, i.e. 5000 training images, and 1000 test images each.

For such image analysis tasks it is conventional to use a Convolutional Neural Network (CNN). For this project, we set our main challenge to be implementing such a network from scratch, instead of using imported networks from a library like TensorFlow[5]. This is a great exercise to better understand how a CNN works in detail, and to see some of the challenges involved in coding such a model.

Section 2.1 introduces the machine learning task from a statistical point of view. We consider the distribution we try to replicate and give a bayesian interpretation of the CNN architecture. Section 2.2 explains the architecture of a CNN, by going through the different layers of the network. Further, Section 2.3 describes how we train the network. We discuss backpropagation in the different layers and the activation and cost functions. In Section 3 we go from the general theory to our actual model. Then we present and discuss our results in Section 4. We plot the performance of our network for various parameters and comment on heuristics used to guide us when selecting the parameters. To get an idea of the relative performance of our model we compare it to TensorFlow. Finally, Section 5 gives a conclusion of the project, including some notes on further improvements to our model that we did not have the time to implement.

2 Theory

2.1 The Machine Learning Task

The general task in machine learning is to given covariates $X = x$ and (usually) targets $Y = y$ estimate $f(X)$, which should form the relation $Y = f(X) + \epsilon$, where ϵ is some random noise in the data [6, p. 28]. In example, we want to model the structural part of the data $f(X)$ while not capturing the noise, we denote our estimate by $\hat{f}(X)$. To assess how good our model is at capturing the structural part it is useful to compare the models performance on data used during training, *training data* and

new unseen data, *validation data*.

If our model performs substantially better on the training data compared to the validation data, this may indicate that our model $\hat{f}(X)$ also models ϵ. The problem can be resolved by making the model less complex, which is typically done by using fewer parameters. For neural networks we can add a penalty term depending on weight size. The result is that we are left with a new optimization problem, which has to find a trade-off between goodness of fit and weight size.

However, if our model is complex enough, one usually assumes that minimizing the loss function with respect to the training data yields an overparametrized model. This motivates *early stopping*, where we train our model until no improvement in goodness of fit is seen with respect to validation data. We use the optimal values for the validation data, and hope that they generalize well [7, pp. 241–249].

In the case of image classification, we may think of the target Y as a categorical, multinomial distributed random variable taking values in $\mathcal{C} = \{class1, ..., classN\}$. Our task becomes to model the probability, $Pr(Y = k|X = x)$ for all $k \in \mathcal{C}$. That we are in fact modelling probabilities will be crucial when we later choose the loss function 3.

CNNs have shown to be particularly successful in image processing. Contrary to a general neural network, CNNs utilize convolutions and pooling to extract the most important features of an image. These operations allow for less flexibility than a general neural network, but being restrictive in this sense is what makes the CNNs so effective. It can be shown that a Neural Network is a universal function approximator, meaning essentially that for any function f there exists a neural network which is able to approximate f arbitrarily well. Still, one does not know which architecture to use to estimate the function. The CNN architecture can from a bayesian point of view be seen as posing an infinitely strong prior on our function $\hat{f}$ and as it turns out, this prior is usually quite good. [7]

2.2 The CNN Architecture

A CNN architecture is centered around processing image data. The challenge with conventional Feed Forward Neural Netoworks (FFNNs) lies in the immense number of parameters required for image classification tasks. For instance, a single image from the MNIST database, constituting a 28×28 pixel square, demands 784 weights for each hidden neuron, solely to parametrize its pixels [8]. This scalability issue swiftly renders conventional FFNNs impractical for higher quality or color images due to the linear relationship between image pixel dimensionality and weights. However, in CNNs, parameter efficiency is achieved through weight sharing via convolution kernels. Weights are distributed and reused across the input space, mitigating the burden of parameters becoming too many. The subsequent section 2.2.1 will delve

into the distinct layers comprising a CNN architecture.

2.2.1 Layers

In a CNN, there are three primary layers, see Figure 1: the convolutional layer, pooling layer, and fully connected layer. Among these, the convolutional layer holds particular significance, as it is fundamental to the network's architecture and lends the CNN its name.

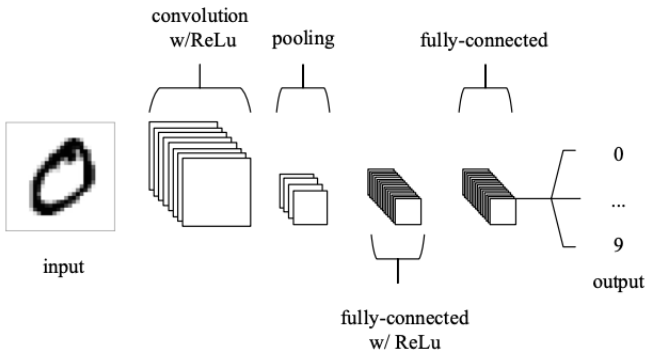


Figure 1: A novice CNN architecture, displaying one instance of each layer. Image gathered from Figure 2.1, in article [8]

Figure 1 gives an overview of what a basic CNN architecture looks like. Figure 1 is an example of a simplistic CNN, however, it is possible to add several instances of each layer throughout the network. More advanced aspects of CNN architecture is discussed in the subsequent sections, whilst various relevant activation functions, are discussed in Section 2.3.

2.2.2 Convolution

At its general form, a convolution can be defined as an operation $*$ between two real-valued functions f and g given by:

$$(f * g)(t) = \int_0^t f(a)g(t-a)da$$

To unpack the expression we consider Figure 2. We see that in practice the convolution corresponds to reflecting g around the second axis and slide it over f, giving a contribution to $f * g$ depending on the overlap.

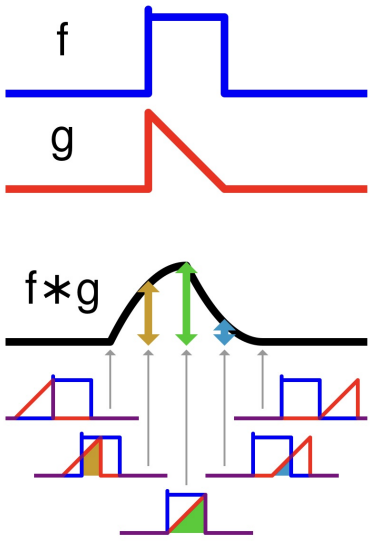


Figure 2: Functions f and g, and the resulting convolution $f * g$[9]

In the context of image processing convolution takes a slightly different form. As pixels are point objects, we now work with discrete functions. If we let f and g be real valued discrete functions, the convolution is given by:

$$(f * g) = \sum_{a=0}^{t} f(a)g(t-a)$$

We refer to the functions f and g as respectively the input and the kernel. On a computer f and g are usually represented by multidimensional arrays, and one assumes that the functions are zero outside of these arrays.

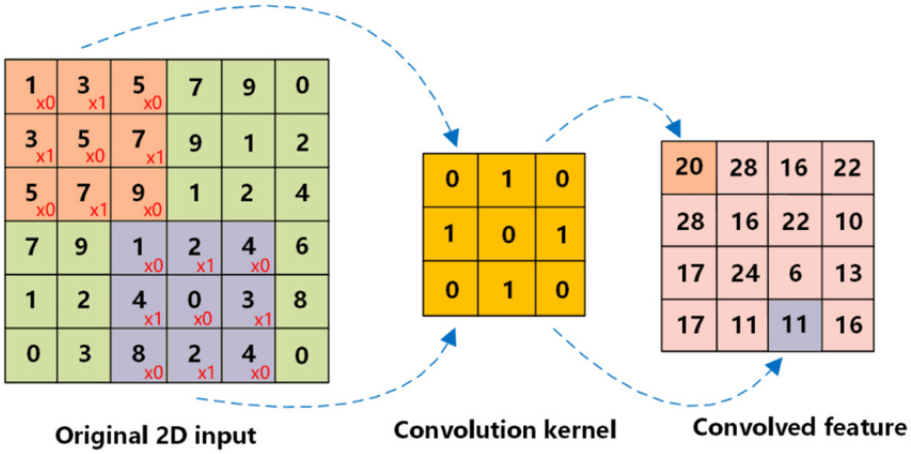


Figure 3: Convolution of a 6×6-matrix, with a 3×3 kernel. The orange and blue submatrices of the input, result in the orange and blue entries of the convolved feature. [10]

The convolution can be seen as taking the Hadamard product of the 180° rotated kernel K and submatrices of the input, I: $\sum_i \sum_j I^{sub}_{i,j} K_{i,j}$, resulting in the convolved feature matrix. Figure 3 shows an example of discrete convolution on a 6×6 input with a 3×3 kernel, and the corresponding output feature map.

There are some non-trivial parameterizations for this operation, i.e. how far we should move the kernel over the input. This length is known as the *stride* of the convolution and is typically denoted as s. Choosing a larger stride, results in a smaller output matrix, hence more dimensionality reduction. A larger stride is also more computationally efficient since we compute fewer Hadamard products. On the downside, a larger stride can result in the loss of some important information as some pixels will be weighted less. This can result in worse model performance. However, a stride of size one will also favour centralized pixels over those close to the boundary. A solution to this is so called *zero padding*. This simply means adding extra zeros around the border of the matrix. Adding no zeros is often called *valid padding*, while adding zeros such that the dimension of the input array equals that of the output is called *same padding*. Lastly, adding zeros such that every pixel appears in the same number of Hadamard products is called *full padding*, this is typically not used as it causes the output to be larger than the input. Often, one chooses something between valid- and same padding, as this typically gives good accuracy, while still reducing the dimensionality.

The values of the kernel is part of what the neural network learns, this process is described in 2.3. In practice, we will implement correlation, which is closely related to convolution. They differ in that correlation does not rotate the kernel, which

should not alter the result. Hence, correlation is often preferred in CNNs as it is more computationally efficient.

2.2.3 Pooling

Another layer that is often featured in a CNN is a pooling layer. Given an input matrix I, the pooling layer acts as a filter traversing areas of the matrix, giving an output based on some rule. The two most common is to take the maximum element over the region or the average of the elements, as shown in Figure 4. Once again, this is a way of reducing the dimensionality of the input. Note that this layer is quite inflexible compared to other neural network layers, as there is no learning feature in this layer.

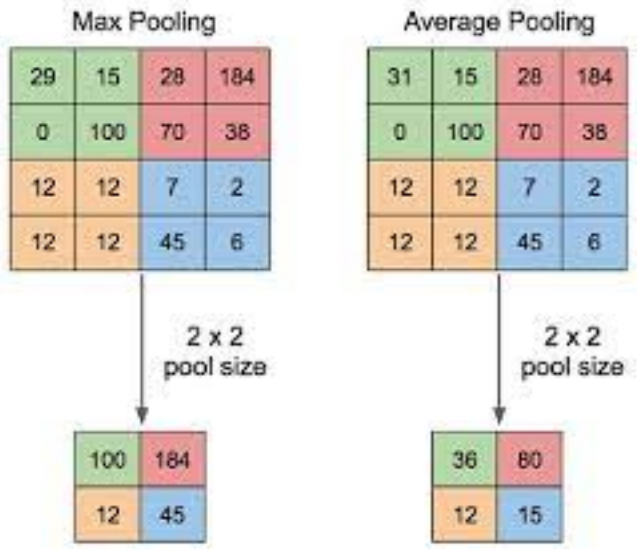


Figure 4: Max and average pooling [11]

2.2.4 Fully Connected

The fully connected layer, or the dense layer, usually constitutes the final layer of the network, where the potentially huge number of high-level features learned by earlier layers, is reduced down to the desired output format, such as class probabilities. Each neuron is connected to the previous layer of neurons output via parameters, or weights, with a bias term, as illustrated in Eq. (1)

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l \tag{1}$$

Where M is the number of neurons, w_{ij}^{L} is a weight at position i, j in the l'th layer in the neural network, b_j^l is the bias term. This output from the neuron is then sent through an activation function to produce the final output of the neuron for that layer l. Eq (2) illustrates the final output of one neuron, which is then sent to the next layer of the network, or the output layer.

$$a_j^l = f\left(z_j^l\right) \tag{2}$$

Where f is the activation function.

2.2.5 Flattening

Since the fully connected layers are one dimensional, but the output from the convolution and pooling is two dimensional (three dimensions if you include depth), we have to flatten the input to the fully connected layers. This is the only thing that is done in the flattening layer.

2.3 Training the Neural Network

There are several key aspects to consider when training a neural network. These include data preprocessing, architecture selection, choice of cost function, activation function, and the back propagation algorithm. There are more aspects of training to consider, however, these are the most important ones, and worth discussing in detail. We will first cover how data preprocessing can affect the accuracy of a neural network.

The MNIST database comprises preprocessed images specifically tailored for machine learning applications. Each image is standardized, normalized, and grayscale with dimensions of 28x28 pixels. This uniformity simplifies tasks like resizing, color normalization, and noise reduction. Designed for ease of use, the MNIST dataset minimizes the need for extensive data manipulation, enabling researchers to dedicate more time to algorithm development rather than data preprocessing.

After preparing the images for input into your neural network, the next crucial step is selecting the appropriate architecture. The structural design of the neural network dictates the type and number of hidden layers to include. There's no definitive answer regarding how to structure these layers or how many to include. However, the architecture significantly impacts the performance of the network. Therefore it is important to choose the right architecture for the classification problem at hand. Since we want to classify MNIST dataset images, several existing networks are benchmark examples. Some of these pre-existing networks are mentioned in Section 3. The specific architecture we decided to go with, is detailed in the Method Section 3.

According to the seminal work by Yann LeCun et al., "*Convolutional Networks combine three architectural ideas to ensure some degree of shift, scale, and distortion invariance: 1) local receptive fields; 2) shared weights (or weight replication); and 3) spatial or temporal subsampling*" [12]. These key ideas are important aspects to consider when building the architecture. LeNet's structure, and LeCun's three principles are illustrated in Figure 5.

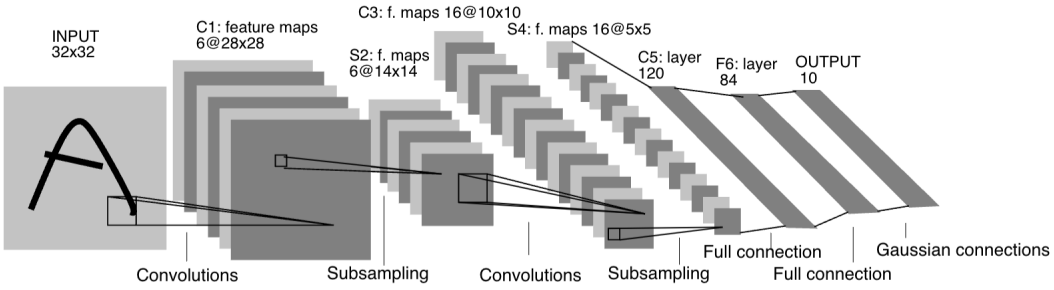


Figure 5: LeNet5's layer structure. Image gathered from [12].

The figure illustrates how the original image is sent through various layers, all with a unique purpose. The convolutional layers extract relevant features in a spatially hierarchical manner, followed by subsampling layers (equivalent to our pooling layers) to reduce spatial dimension. The last layers are the fully connected layers, reducing the feature maps of the previous layer, down to an output that is suitable for the classification task.

For contrast, the architecture of AlexNet [13], is given in Figure 6.

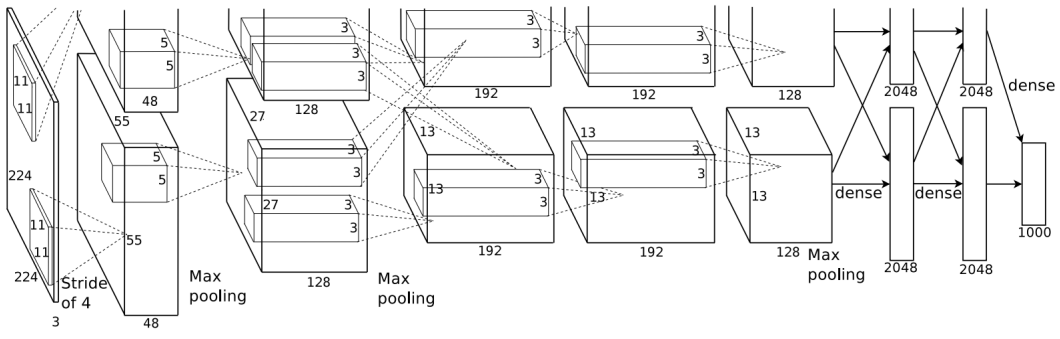


Figure 6: AlexNet layer structure, image gathered from [13]

As described in the article [13], and illustrated in Figure 6, the AlexNet contains eight layers, five convolutional and three fully-connected layers. The pooling layers

are not explicitly shown as layers in this architecture, but there are essentially three pooling layers in addition to the learned layers. Both networks uses a convolutional layer first, followed by pooling, and lastly dense layers. This order of layers is what we ended up using, and is described further in the Method Section 3.

AlexNet's architecture is deeper and more complex compared to LeNet-5's architecture. It also uses two pairs of kernels, 11×11, and 3×3 which are more adept at capturing complex data due to their increased receptive field. Smaller kernels like 5×5 and 2×2 used in LeNet-5, can miss out on important contextual information, due to their smaller receptive field.

The two networks illustrate the evolution of CNN architecture over the years and how AlexNet made drastic improvements over the initial LeNet-5 with its deeper and more complex architecture[1].

When it comes to the choice of cost, and activation function, there are several to choose from, each with its advantages and drawbacks. However, for classification analysis using images, there are two clear choices. For cost function where the goal is to obtain a classification output, log-loss is best, because it can compare predicted probabilities with the true labels. For the activation function, the optimal function is Rectified Linear Unit (ReLU) or Leaky ReLU. Log-loss is given in Eq. (3), ReLU is defined as Eq (5), leaky ReLU is defined in Eq. (4). Another popular activation function worth mentioning, is the sigmoid function Eq. (6). In addition we use the softmax function, Eq. (7), on the last layer, to get a normalized output containing the probabilities of each class according to the model.

$$\text{Log Loss } \mathcal{C}(\boldsymbol{W}) = -\sum_{i=1}^{n} \left(t_i \log a_i^L + (1 - t_i) \log \left(1 - a_i^L \right) \right), \tag{3}$$

$$\text{Leaky ReLU: } f(x) = \begin{cases} 0.01x & x < 0 \\ x & x \geq 0 \end{cases} \tag{4}$$

$$\text{ReLU: } f(x) = \max(0, x), \tag{5}$$

$$\text{Sigmoid: } \sigma(x) = \frac{1}{1 + e^{-x}}, \tag{6}$$

$$\text{Softmax: } \sigma(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}. \tag{7}$$

Where t_i is the target variable, and a_i^L is the output. Log loss is a smooth and differentiable function, making it perfect for gradient descent algorithms. (3) ReLU

[1]Much of this improvement comes from the fact that AlexNet was one of the first CNNs to use the computational power of GPUs for training[13].

is optimal due to the reduced computational overhead, due to its simplicity, in backward and forward passes.

Before we discuss the forward and backward passes in our neural network, let's discuss various optimization algorithms suitable for updating our weights.

Gradient descent algorithms are the most widely used optimization algorithms in machine learning. They are conceptually easy to understand, and they exhibit good performance. Most gradient descent algorithms optimize the parameters of the model (the weights) by moving in the direction of the steepest negative gradient of the cost function. There are several different ways you can move in the direction of the negative gradient, and this is where the various choices of optimization algorithms come in. You can either take big steps, medium steps, small steps, or a mix of previous steps in the direction of the negative gradient. This step is known as the learning rate, and all the various methods have their own way of manipulating the learning rate. Lets list some of the most relevant and up to date[14].

The most prominent algorithm is known as the Adaptive Moment Estimation (Adam), followed by Root Mean Squared Propagation (RMSprop) and Adaptive Gradient (Adagrad). Following the notation of [14], the three variants are listed in eqs (8), (9), and (10)

$$\text{Adam:} \quad \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{8}$$

$$\text{RMSprop:} \quad \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E\left[g^2\right]_t + \epsilon}} g_t \tag{9}$$

$$\text{Adagrad:} \quad \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t. \tag{10}$$

Where θ are the model parameters (i.e. the weights), η is the learnings rate, $\hat{v}_t$ is the exponentially moving average of weights gradients, ϵ is a small constant for numerical stability, $\hat{m}_t$ is the bias corrected estimate of the first momentum, g_t is the gradient, G_t is the accumulated gradient, $E\left[g^2\right]_t$ is the same as $\hat{v}_t$, and $\odot$ is the Hadamard product[14].
These have in common, that they seek to solve and find the optimal learning rate. Adagrad calculates its learning rate based on which parameters of the model appear frequently. Setting a larger learning rate for infrequent parameters, and a smaller one for frequent parameters. This creates an issue of vanishing learning rates, which RMSprop aims to bring a solution to. The learning rate diminishes, as demonstrated by Eq(9) in article [14], when gradient accumulation occurs in the denominator of the update rule[14]. RMSprop slows down the issue of vanishing learning rates, by updating the parameters with an exponentially decaying average of gradients, instead of accumulating all past gradients. Adam solves the issues of vanishing learning rates by combining ideas from RMSprop and momentum optimization.

Having explored a range of cost and activation functions, equations (3), (4), (5), (6) and (7), and considered various learning rate options, equations (10), (8), and (9), it's now time to see how we can use these equations to optimize our model parameters.

2.3.1 Backpropagation

Before discussing the more complex aspects of backpropagation, a fundamental concept in machine learning, let's briefly introduce the subject to try and get a grasp of what it aims to accomplish. It is a method used to adjust the weights of connections between neurons in a neural network. By adjusting the weights, the network is able to improve its performance over time, by learning from its previous errors. It does this by making predictions or classifications based on the current network's weights, and then comparing them to the true values, the actual target values the network tries to replicate. The network's backpropagation algorithm then works backward through the network, updating the weights in the direction that minimizes the loss between the predicted values, and the true values.

One of the main challenges when constructing a CNN, is to create a backpropagation algorithm that works in all the network layers. In the following sections, we will cover how the backpropagation is implemented in a fully connected layer, the pooling layer, and lastly the convolutional layer.

2.3.2 Backpropagation in the Fully Connected Layer

Backpropagation, backprop for short, refers to the process of calculating gradients. Backprop along with the learning algorithms mentioned in Section 2.3, is what propagates backwards through the network after a forward pass, to update the parameters. The gradient backprop calculates is defined in Eq (11).

$$\frac{\partial \mathcal{C}\left(\hat{W}^L\right)}{\partial w_{jk}^L}. \tag{11}$$

$\mathcal{C}$ represents the cost function with respect to the weight matrix W for a given layer L, this gradient can be factorized using the chain rule, resulting in an expression dependent on three factors, as shown in Eq. (12),

$$\frac{\partial \mathcal{C}}{\partial w_{jk}^L} = -\frac{1}{n}\sum_{i=1}^{n}\left[\frac{\partial \mathcal{C}}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L}\right]. \tag{12}$$

Where n is the number of training samples. The factorized expression is easier to analytically calculate for various cost functions. Using log loss, and a sigmoid

activation function Eq (3), as an example, the three factors can be defined as eqs (13),(14), (15)

$$\frac{\partial \mathcal{C}}{\partial a_i^L} = -\left(\frac{t_i}{a_i^L} - \frac{1-t_i}{1-a_i^L}\right), \tag{13}$$

$$\frac{\partial a_j^L}{\partial z_j^L} = \sigma'\left(z_j^L\right) = a_j^L\left(1-a_j^L\right), \tag{14}$$

$$\frac{\partial z_j^L}{\partial w_{jk}^L} = a_k^{L-1}. \tag{15}$$

Where σ' is the derivative of the sigmoid/softmax activation function (6). Refer to the Appendix for the derivation (see 6). If we were to use the ReLU function, Eq(5), the expression would look different. These equations combine into a not-so-pretty analytical expression, given in Eq (16)

$$\frac{\partial \mathcal{C}}{\partial w_{jk}^L} = -\frac{1}{n}\sum_{i=1}^{n}\left[-\left(\frac{t_i}{a_i^L} - \frac{1-t_i}{1-a_i^L}\right)\cdot a_j^L\left(1-a_j^L\right)\cdot a_k^{L-1}\right]. \tag{16}$$

Eq (16) isn't typically implemented explicitly in code; instead, optimized libraries like JIT, JAX, PyTorch, or TensorFlow are used to calculate gradients. These libraries offer highly optimized tools for calculating gradients.

Using the gradient, the weights are updated in accordance with which optimization algorithms, outlined in eqs (10),(8), and (9), one has chosen.

2.3.3 Backpropagation in the Pooling Layers

The pooling layers do not have any weights that need optimization; therefore, no gradient calculations are typically done here. Pooling layers are there for dimensionality reduction and aim to capture the most salient feature data. During backpropagation, the gradients from the subsequent layer are distributed back to the pooling layer based on the positions of the maximum or average values. Our unique implementation of backpropagation in the pooling layer is further explained in the Method section 3

2.3.4 Backpropagation in the Convolutional Layer

There are a couple of different approaches one could take when it comes to explaining how the backpropagation algorithm works in the convolutional layer. This section will lay out a couple of illustrations showing computational graphs, and how that logic extends to obtaining the backpropagation gradient in the convolutional layer.

Figure 7 shows a convolution action between an input matrix X, and a filter or kernel K, producing an output. The dotted lines, are the gradients flowing as a backward pass through the network. In order to reduce clutter, the gradients are omitted from the figure, but the dotted lines from top left and down $\frac{\partial C}{\partial X}$, and $\frac{\partial C}{\partial K}$ respectively and the ingoing line from the right is the previous layers backward pass output, $\frac{\partial C}{\partial a}$.

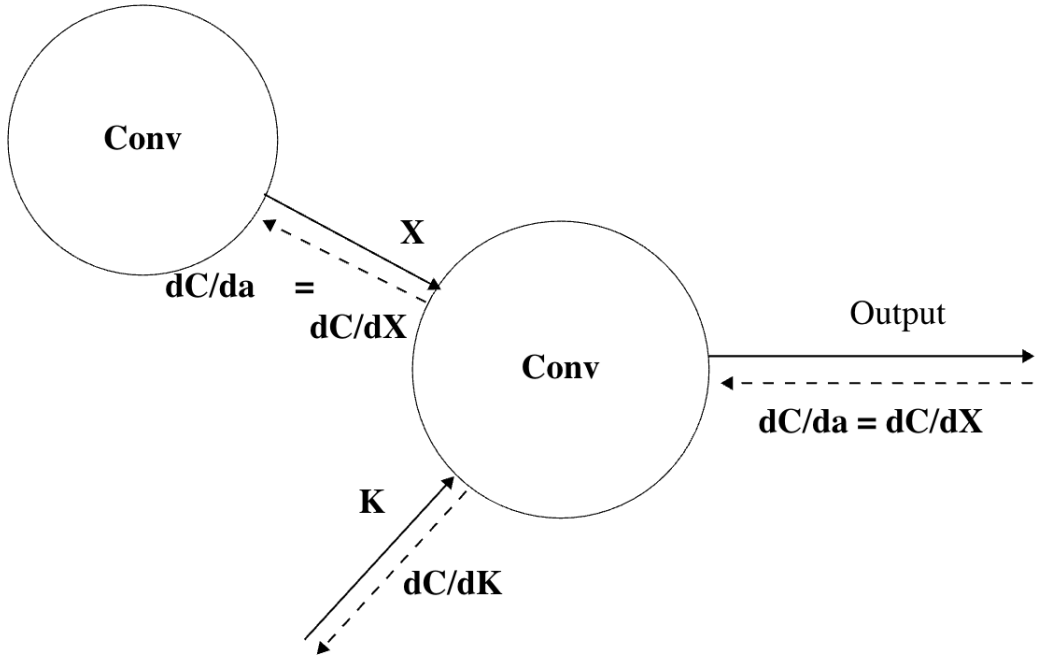


Figure 7: Computational graph showing the gradients to be calculated in the backpropagation.

The goal of the backpropagation algorithm is to calculate the gradients $\frac{\partial C}{\partial X}$, and $\frac{\partial C}{\partial K}$, which are then used to update the kernel K. The gradient $\frac{\partial C}{\partial X}$ will be the next layers backward pass output $\frac{\partial C}{\partial a}$.

There are several ways to show the calculation of the backprop gradients, and the following approach will adopt the neat way of doing it as described in the recommended article on convolutional backprop[2]

The backward pass in the convolutional layer mirrors the forward pass, as both involve convolutions. The gradients, $\frac{\partial C}{\partial X}$ and $\frac{\partial C}{\partial K}$, can be described as convolutions between the input matrix X and the loss gradient $\frac{\partial C}{\partial a}$, and a full convolution between

[2]The article is found in the notebook for week 6 in the lecture slides, https://github.com/CompPhysics/AdvancedMachineLearning/blob/main/doc/pub/week6/ipynb/week6.ipynb

a 180° rotated filter F and the loss gradient, respectively. These two convolutions are illustrated in Eqs. (17) and (18).

$$\frac{\partial C}{\partial K} = \text{ Convolution } \left(\text{ Input } X, \text{ Loss gradient } \frac{\partial C}{\partial a} \right), \tag{17}$$

$$\frac{\partial C}{\partial X} = \underset{\text{Convolution}}{\text{Full}} \left(\begin{array}{c} 180° \text{ rotated} \\ \text{Kernel } K \end{array} , \text{ Loss gradient } \frac{\partial C}{\partial a} \right). \tag{18}$$

The gradient we need for updating the biases is just

$$\frac{\partial C}{\partial B} = \frac{\partial C}{\partial a}.$$

In the very last layer of the network (which corresponds to the first layer during the backward pass), the loss gradient with respect to the output is already known. This is because the output of this layer is the result of the last forward pass in the network.

The input matrix X, loss gradient, and the kernel K, are all matrices. The below equations are Eqs. (17), and (18) expanded to matrix form,

$$\begin{array}{|c|c|}\hline \frac{\partial C}{\partial K_{11}} & \frac{\partial C}{\partial K_{12}} \\ \hline \frac{\partial C}{\partial K_{21}} & \frac{\partial C}{\partial K_{22}} \\ \hline \end{array} = \text{Conv} \left(\begin{array}{|c|c|c|}\hline x_{11} & x_{12} & x_{13} \\ \hline x_{21} & x_{22} & x_{23} \\ \hline x_{31} & x_{32} & x_{33} \\ \hline \end{array}, \begin{array}{|c|c|}\hline \frac{\partial C}{\partial a_{11}} & \frac{\partial C}{\partial a_{12}} \\ \hline \frac{\partial C}{\partial a_{21}} & \frac{\partial C}{\partial a_{22}} \\ \hline \end{array} \right),$$

$$\begin{array}{|c|c|c|}\hline \frac{\partial C}{\partial x_{11}} & \frac{\partial C}{\partial x}\,12 & \frac{\partial C}{\partial x_{13}} \\ \hline \frac{\partial C}{\partial x_{21}} & \frac{\partial C}{\partial x}\,22 & \frac{\partial C}{\partial x_{23}} \\ \hline \frac{\partial C}{\partial x_{31}} & \frac{\partial C}{\partial x}\,32 & \frac{\partial C}{\partial x_{33}} \\ \hline \end{array} = \text{Full Conv} \left(\begin{array}{|c|c|}\hline K_{22} & K_{21} \\ \hline K_{12} & K_{11} \\ \hline \end{array}, \begin{array}{|c|c|}\hline \frac{\partial C}{\partial a_{11}} & \frac{\partial C}{\partial a_{12}} \\ \hline \frac{\partial C}{\partial a_{21}} & \frac{\partial C}{\partial a_{22}} \\ \hline \end{array} \right).$$

With the gradient $\frac{\partial C}{\partial K}$ found, the kernel weights can be updated using one of the previously mentioned optimization algorithms in Sec 2. The backward pass flows through the layers of the network, until it reaches the input layer, updating the kernel weights based on the gradient and the chosen optimization algorithm. For more details on the mathematical operation of a convolution between matrices, see Appendix Sec 6.2.

3 Method

This section is included to give an overview of how the general theory of CNNs, as discussed in Section 2, is adapted and refined in our specific CNN architecture. There are a wide variety of ways one can construct a CNN architecture, each with its unique characteristics. Notable examples include LeNet-5[3], and AlexNet[4]. Firstly, we will cover what layers we decided to use, how we used them, and how the various integral functions like feed-forward, and backpropagation work for each layer.

3.1 The CNN Architecture

For our CNN, we decided we would try building our architecture first with only one of each of the previously mentioned layers in Section 2.2.1. We also tried extra pooling layers, and convolutional layers. Our network for the MNIST data is structured as the illustration in Figure 8 shows. As mentioned and as seen in the figure, we mainly use a single convolution layer with kernel size 3×3, and an average pool or maxpool layer, and vary the number of filters for the output feature map. Then we flatten the output and use a single hidden fully connected layer with x hidden nodes (sometimes we also dropped this layer). Lastly, we use a softmax layer to get probabilities of each class and classify the output. Figure 9 shows the structure of the model we used on the Cifar10 data. We only used a single convolution layer with a kernel size 4×4 and 6 filters, with a maxpool layer, and no hidden fully connected layer, just an output layer. We will discuss this more in Section 4.

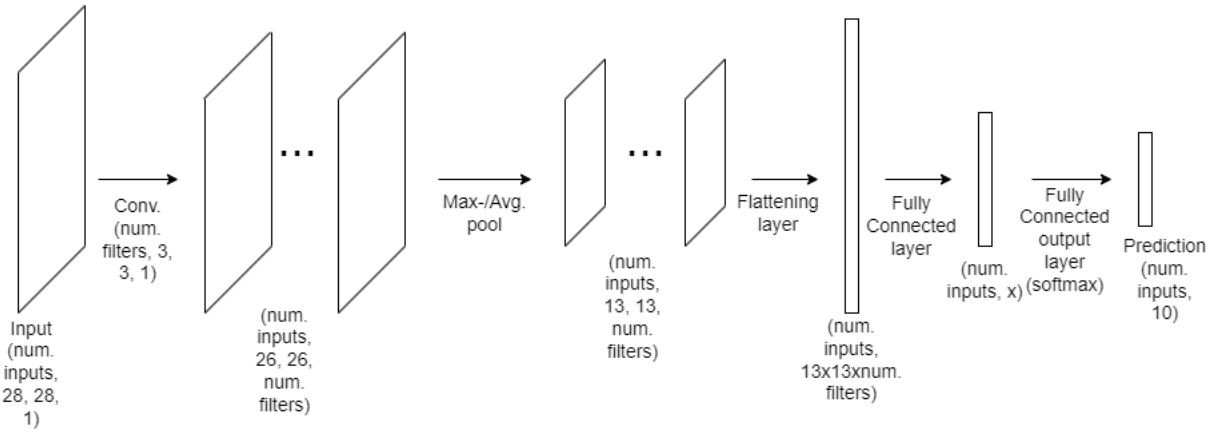


Figure 8: Structure of our model used on the MNIST data.

[3]Proposed by Yann LeCun and widely recognized for its application in handwritten digit recognition tasks [4]

[4]first network to achieve sub 25% error rate for ImageNet Large Scale Visual Recognition Competition in 2012.

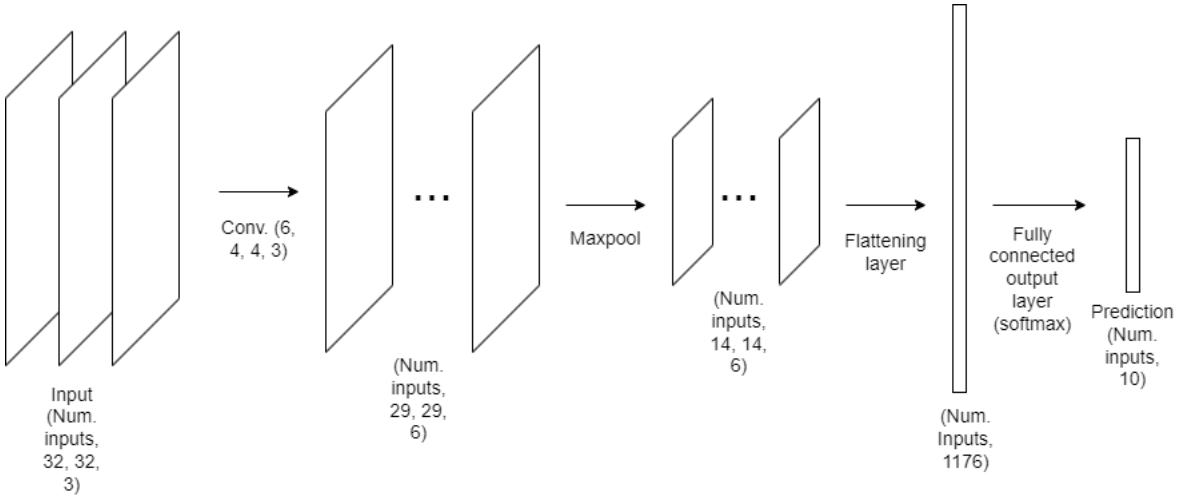


Figure 9: Structure of our model used on the Cifar10 data.

In our CNN implementation, each layer is represented as a class, each containing specific class methods such as a feed-forward method and a backpropagate method. These methods handle the matrix operations detailed in sections 2.2.2, 2.2.4, and 2.2.3, as well as the gradient calculations discussed in Section 2.3 for the backpropagate method.

The CNN itself serves as the overarching class, featuring an add_layer method. This method enables users to add as many layers to the architecture as wanted.

3.2 Feed Forward

The feed-forward method is a common method for all layers, from the first convolutional layer, to the last fully-connected layer. The single goal of the feed-forward method is to make sure that the input and subsequent output propagate throughout the network. The CNN class itself initializes the feed_forward method in all the added layers, and in the correct order. The mathematical implementation of the convolution operation as implemented in code, can be found in Appendix Sec 6.2. Notably, we decided to vectorise the convolution operation to be computationally effecient.

3.3 Backpropagation

The backpropagate method is a shared class method among all layers, as well as in the overarching CNN class. Similar to the feed-forward method, backpropagate initializes all subsequent layers' backpropagation methods within the network. As the backward pass traverses backward through all layers, each layer's weights are

updated according to the principles outlined in Sec 2.3.1. The implementation of the convolution operation in the backpropagate method is described in the Appendix, Sec 6.4,

3.4 Flatten

The goal of our implementation of the flattening layer is to take the 3D (4D if you include different inputs as a dimension) output of the pooling layer and flatten it to a 1D array (2D if you include different inputs as a dimension).

3.5 Optimization

To avoid extensive use of time expensive for-loops in our implementation, we have used an algorithm for the forward and backward propagation which only needs matrix multiplications. This involves manipulating the kernel and the input so they will fit such an operation. The implementation is explained in Section 6.3.

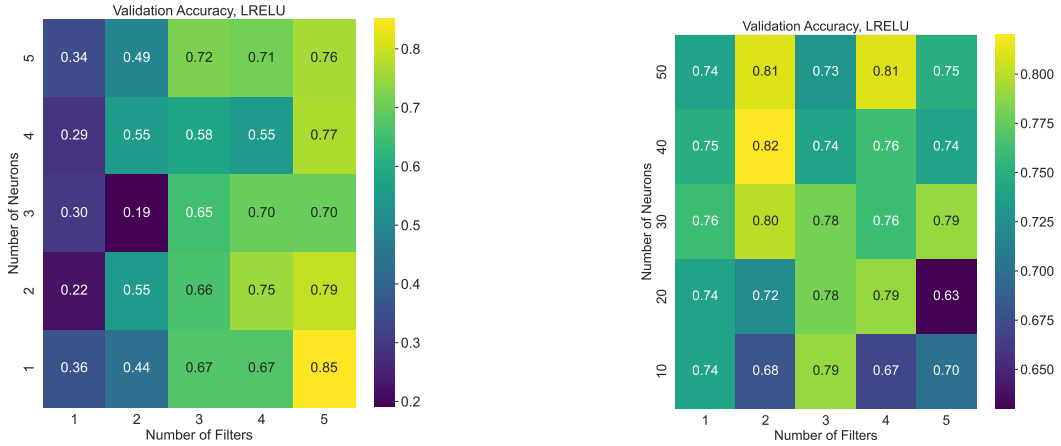
4 Results and Discussion

This section introduces the results of our finalized CNN. We will report and discuss the accuracy of the network, and the optimized parameters such as η, λ. Additionally, we include several heatmap plots illustrating the optimal parameter combinations. Our network will also be compared to the performance of a TensorFlow network with the same structure as our CNN.

4.1 The MNIST Dataset

For all the plots in this section we used we used Adam as optimizer with exponential decay rates $\rho 1 = 0.9$ and $\rho 2 = 0.999$.
In Fig. 10 we show the validation accuracies for the LRELU activation function when varying the number of neurons in the hidden fully connected layer, and the number of filters in the convolution layer. Here we have looked at the MNIST dataset, using 600 images for training. The best result is achieved when we use only one neuron in the hidden layer, and five filters in the convolution layer, which gives an accuracy of 0.85. The fact that we get the highest accuracy when using so few neurons might indicate that an increase in the number of neurons leads to overfitting. If we use too many neurons in the hidden layer, we allow the model to fine-tune in such a way that it could risk fitting the training data very well without considering how to give good predictions for unseen data. Thus the model would not be able to predict well, and the validation accuracy would decrease. This is likely what we see when increasing the number of neurons. The fact that our best result comes from using 5 filters might be because this allows the model to pick up on more patterns in the images. Remember that the main goal of the convolution layer in the network is to use filters

that can find different patterns in an image. For instance, one filter might show where there are vertical lines in the image, and another might look for horizontal lines. If we use only one filter, the network cannot pick up on much structure. This may be why the first column in Fig. 10a gives very low accuracies ranging from 0.22 to 0.36. When using five filters, however, we allow the model to pick up on more patterns in the images, which improves the overall performance. This is likely why we get the best result with five filters.
The models were trained using $\eta = 0.01$ and $\lambda = 0.001$. For the sparser models in Figure 10a we used 100 batches and 100 epochs. For the more complex models in Figure 10b we used 300 batches and 200 epochs, as we thought that a more complex model should consider more observations before adjusting its weights, to avoid overfitting. It is worth noting that this also could affect the results in favour of the sparser models. Ideally, we would have tried more combinations, but as the program ran very slowly on our computers, we chose to adjust these parameters simultaneously.



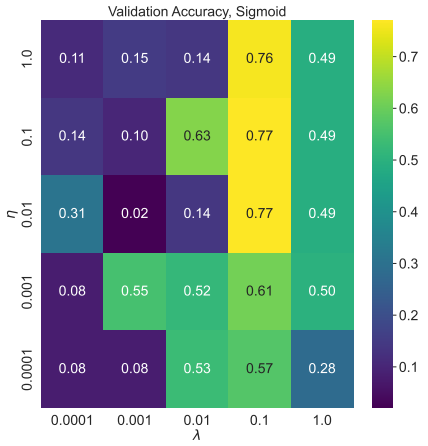
(a) Accuracies for a small hidden layer with one to five neurons.

(b) Accuracies for a larger hidden layer with ten to fifty neurons.

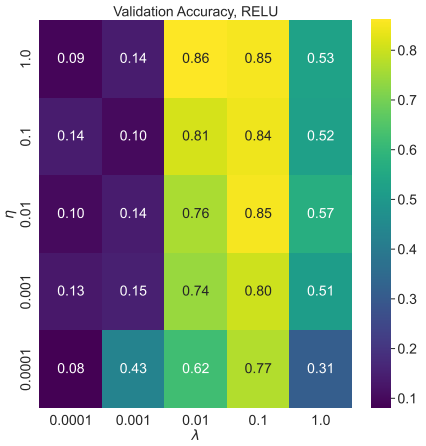
Figure 10: Validation accuracies for different numbers of neurons in the hidden layer, and different number of filters in the convolution layer, using the LRELU activation function. These accuracies were obtained from training our model on 600 images from the MNIST dataset.

In Fig. 11 we show the validation accuracies for the sigmoid, RELU and LRELU activation functions when varying the learning rate η and the regularization parameter λ. For these accuracies we trained our model on 600 images from the MNIST dataset. The best result here is achieved with the LRELU activation function, which gave a maximum accuracy of 0.87 for $\eta = 0.1$ and $\lambda = 0.1$. We got similarly good results using the RELU activation function, which gave a maximum accuracy of 0.86 for

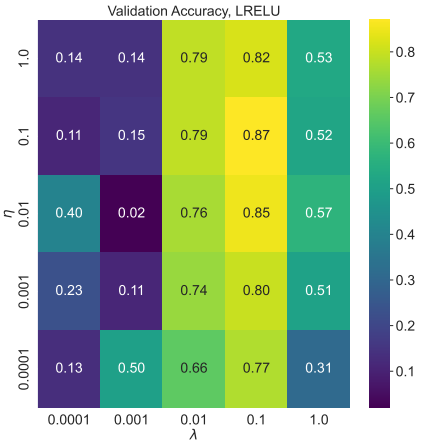
$\eta = 1$ and $\lambda = 0.01$. The sigmoid activation function, however, did not give as good results. Its maximum was 0.77 at $\eta = 0.1$ and $\lambda = 0.1$ or $\eta = 0.01$ and $\lambda = 0.1$. The fact that sigmoid performs poorly might be surprising, since the way it scales any neuron output to be between 0 and 1 would seem perfect for classification problems. However, the derivative of the sigmoid activation function is not well-behaved, giving values close to zero when the input is very small or very large. This is known as the vanishing gradient problem, and might explain why RELU and LRELU outperforms sigmoid so greatly. Note that the RELU also has a problem with the gradient vanishing for input-values below zero, but since we are working with classification most of the parameters we deal with are positive, so this is not as big of an issue as for sigmoid. This gradient problem could, however, explain why the LRELU activation function performs slightly better than RELU.



(a) Accuracies when using the sigmoid activation function.



(b) Accuracies when using the RELU activation function.



(c) Accuracies when using the LRELU activation function.

Figure 11: Validation accuracies for the sigmoid, RELU and LRELU activation functions when varying the learning rate η and the regularization parameter λ. These accuracies were obtained from training our model on 600 images from the MNIST dataset. When training this model we did not include a hidden fully-connected layer and we used 20 filters of size 3×3. We used 100 batches and trained the models for 100 epochs.

In Fig. 12 we show the validation accuracies for the LRELU activation function when varying the learning rate η and the regularization parameter λ. In this figure we have used 6 000 images from the MNIST dataset, opposed to the 600 we used previously. Since our code is quite slow compared to optimized packages such as Tensorflow, training on 6000 images requires too much computational effort, which is why we

have not done so for the previous results. It is, however, interesting to see how our model would improve with an increased number of images, which is what we want to look at in Fig. 12. What we see is that we get better results when using more images, as we reach a maximum accuracy of 0.91 for $\eta = 0.1$ and $\lambda = 0.1$. This is a clear improvement to our previous results, being the first result to give an accuracy above 90%. The reason we get better results now is likely that the model has more data to train on, meaning it has more datapoints to fit the different parameters to. In particular, this may allow us to increase the complexity of our model without overfitting to the data, which is why increased the number of hidden neurons to 20. If we had more time, and access to better computers to run our models on, it would definitely have been interesting to generate heatmap plots as seen in Fig. 10 and 11 with 6 000 images, or even using all 60 000 images provided by the MNIST dataset, and see how this would affect, and perhaps improve, our results.

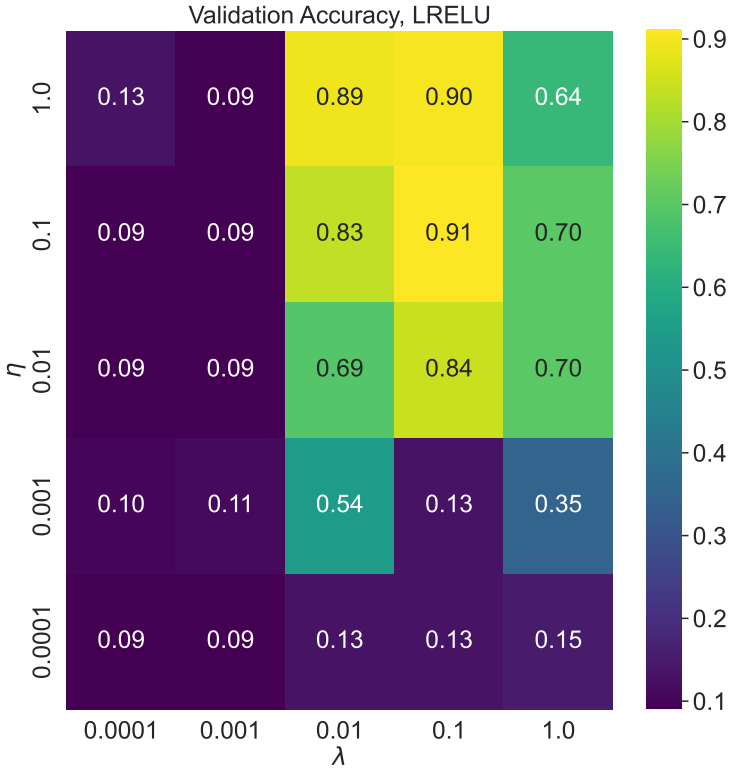


Figure 12: Validation accuracies for the LRELU activation function when varying the learning rate η and the regularization parameter λ. These accuracies were obtained from our model on 6000 images from the MNIST dataset. In this model we have used a hidden fully-connected layer with 20 neurons and 20 filters of size 3×3. We used 200 batches and trained the models for 100 epochs.

Lastly, in Figure 13 we changed the pooling layer to averagepool instead of maxpool, which is what we used in the previous models. We trained the model on 600 images, with 20 filters of size 3×3 using the LRELU activation function. We see that the best accuracy is 0.85 found at $\lambda = 0.0001$ and $\eta = 0.01$. The best accuracy is slightly worse than what we found for the corresponding model differing only in the pooling layer, see Figure 11c.

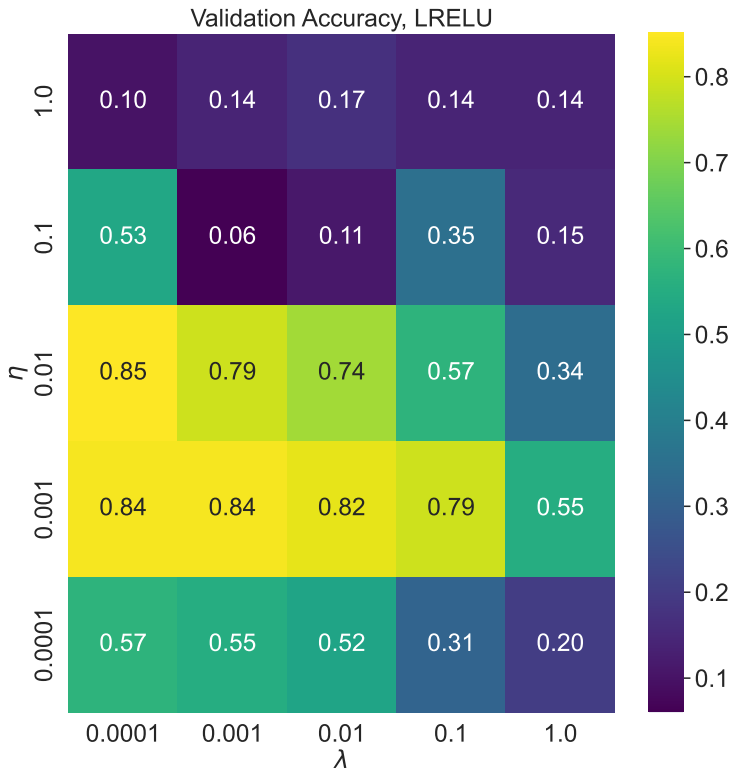


Figure 13: Validation accuracies for the LRELU activation function, when varying the learning rate η and the regularization parameter λ. These accuracies were obtained from our model on 600 images from the MNIST dataset. In this model we have used an averagepool layer, 20 filters of size 3×3, 100 batches and 100 epochs.

4.2 CIFAR-10 dataset

A heatmap plot for different learning rates η and L2 regularization parameters λ for the validaton set is shown in Figure 14 using our program and Tensorflow to compare the results. Figure 15 shows the training accuracies for the same parameters. The structure of our model is illustrated in Figure 9. We use Adam as optimizer with exponential decay rates $\rho_1 = 0.9$ and $\rho_2 = 0.999$. For all the runs we used batch size 50 and 50 epochs. We use LRELU as activation function in the convolution layer, and softmax on the output layer, with Log Loss as cost function.

To find the best structure and model parameters, we tested different models using the TensorFlow code, because it has some optimizations which makes it a lot faster than ours to run. The problem was that we had to limit the amount of input images to 500 training images and 100 validation images, or else our code would take a very long time to run (which we tested on bigger datasets), so quite a simple structure with not too many parameters ended up performing the best. From the validation and training accuracies achieved with Tensorflow, we can see that such a small dataset will lead to overfitting, since the training accuracy is much higher than the validation accuracy. The highest validation accuracy score for Tensorflow was 0.38 for $\eta = 0.001$ and $\lambda = 0.001$, which is better than randomly guessing categories, but far from a good accuracy for any problem. For the training set the best accuracy was 1.00 for $\eta = 0.01$ and $\lambda = 0.00001, 0.0001, 0.001$ and 0.01, with validation accuracies for the same parameters being respectively $0.30, 0.29, 0.30$ and 0.34, which is a clear sign of overfitting.

When it comes to our own model, it did not really train on the dataset at all. The best training accuracy was 0.17 for $\eta = 0.001$ and $\lambda = 0.0001$ or 0.001, while the best validation accuracy was 0.14 for $\eta = 0.001$ and $\lambda = 0.001$, which is basically the same as arbitrarily guessing the label at an accuracy around 0.10. Potentially if we had a better optimized program, we could run larger models with a larger portion of the available data, and also use more epochs and batches. This way we could confirm if there is something wrong with the code, or if it is just the lack of structural depth and generalization that causes the problem. One advantage TensorFlow has over our code is that it executes in C++ [15], which is a low-level coding language with typically much better performance than python. This could be looked into for our code, but would take a long time to implement. Another more simple upgrade would be to make a grad function for the activation when we initialize a new class, this way we don't need to calculate the gradient for every epoch and every batch. Also, more different functionalities, like dropout which improves generalization, batch normalization, residual connections or gradient clipping, which improve gradient flow, could make the problem more easily solvable, but it should be noted that for example residual connections are used for larger model architectures. Our optimized convolution does not include stride or padding either, which is important to look

into as an improvement. The reason why we included the CIFAR-10 dataset was mainly to test our program on input with multiple input channels, but our program clearly needs additions to handle this.

Figure 14: Validation accuracies for different learning rates and regularization parameters on the CIFAR-10 dataset using TensorFlow on the left and our program in the righthand plot.

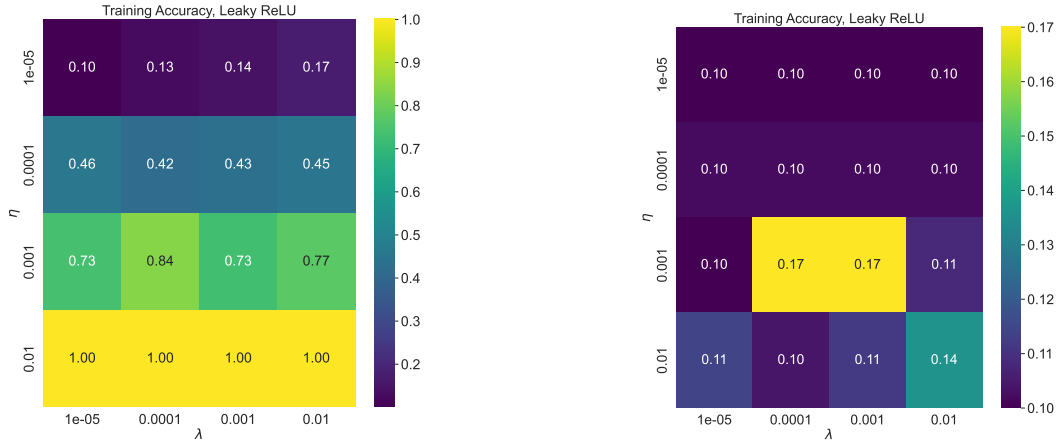


Figure 15: Training accuracies for different learning rates and regularization parameters on the CIFAR-10 dataset using TensorFlow on the left and our program on the right.

5 Conclusion

In this project we have written a convolutional neural network from scratch, and used it to classify images of handwritten digits in the MNIST dataset, and colored images of animals and vehicles in the CIFAR-10 dataset. When training our network, we varied different parameters of our network such as the learning rate η, the regularization parameter λ, layer architecture and activation functions.
For the MNIST dataset we obtained the best results when training the network on 6 000 images using the LRELU activation function, with $\eta = \lambda = 0.1$ using 20 convolution filters of size 3×3 and a hidden fully-connected layer of 20 neurons. These parameters gave an accuracy of 91% on the validation data, which is relatively good.
We did not get any good results when applying our code to the CIFAR-10 dataset, with a maximum validation accuracy of 14%. The fact that our network performed poorly on CIFAR-10 is likely because the addition of color channels makes the training process so computationally heavy that we could only afford to train on a very limited number of images, namely 500. Although the results were not good, the CIFAR-10 dataset made it possible for us to test whether our network can be used on colored images, and we found that, although the code runs properly, the network clearly needs some upgrades to be able to handle such data.
Being able to account for stride and padding in our optimized convolution should be added. Translating our code to a low-level coding language like C++ would most likely improve running time, and could therefore be looked into. Also, we could go in detail and make sure that we avoid unnecessary work, like calculating gradients more times than what is necessary. Further improvements include more advanced functionality, like dropout and batch normalization layers, residual connections or gradient clipping.

6 Appendix

6.1 Gradient Calculation

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{Softmax}$$

$$\begin{aligned}\frac{\partial a_k^L}{\partial z_k^L} = \sigma'\left(z_k^L\right) = \frac{\partial}{\partial z_k}\left(\frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}\right) = \frac{e^{z_k}\left(\sum_j^K e^{z_j} - \delta_{kj} e^{z_j} e^{z_k}\right)}{\left(\sum_j^K e^{z_j}\right)^2} \\ = \frac{e^{z_k}}{\sum_j^K e^{z_j}}\left(1 - \frac{e^{z_k}}{\sum_j^K e^{z_j}}\right) = a(1-a)\end{aligned}$$

6.2 Convolution

Let X be a $m \times n$ matrix, and K a $k \times l$ matrix. The convolution $X * K$ is given by

$$
\begin{aligned}
X * K &= \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} * \begin{bmatrix} k_{11} & \cdots & k_{1l} \\ \vdots & \ddots & \vdots \\ k_{k1} & \cdots & k_{kl} \end{bmatrix} \\
&= \begin{bmatrix} \sum_{i=1}^{k}\sum_{j=1}^{l} x_{ij}k_{ij} & \cdots & \sum_{i=1}^{k}\sum_{j=1}^{l} x_{1,n-j+1}k_{ij} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{k}\sum_{j=1}^{l} x_{m-i+1,j}k_{ij} & \cdots & \sum_{i=1}^{k}\sum_{j=1}^{l} x_{m-i+1,n-j+1}k_{ij} \end{bmatrix} \qquad (19) \\
(X * K)_{ab} &= \sum_{i=1}^{k}\sum_{j=1}^{l} x_{a-1+i,b-1+j}k_{ij},
\end{aligned}
$$

where $(X*K)$ is a $(m-k+1)\times(n-l+1)$ matrix. The way we have defined convolution here is known as a *valid* convolution. It should be noted that a convolution requires the kernel to be rotated 180 degrees, so what we here define as a convolution is really a *correlation* in mathematical terms.

6.3 Optimized convolution

Here we will show in detail how we turn the convolution operation into a matrix multiplication to optimize our numerical implementation. We start by flattening the expression in Eq (19) to get

$$
(X * K)_{\text{flat}} = \left[\sum_{i=1}^{k}\sum_{j=1}^{l} x_{ij}k_{ij} \cdots \sum_{i=1}^{k}\sum_{j=1}^{l} x_{1,n-j+1}k_{ij} \cdots \sum_{i=1}^{k}\sum_{j=1}^{l} x_{m-i+1,j}k_{ij} \cdots \sum_{i=1}^{k}\sum_{j=1}^{l} x_{m-i+1,n-j+1}k_{ij} \right]
$$

$$
= \begin{bmatrix} k_{11} & \cdots & k_{1l} & \cdots & k_{kl} & \cdots & k_{kl} \end{bmatrix} \begin{bmatrix} x_{11} & \cdots & x_{ab} & \cdots & x_{m-k+1,n-l+1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{1l} & \cdots & x_{a,b+l-1} & \cdots & x_{m-k+1,n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{k1} & \cdots & x_{a+k-1,b} & \cdots & x_{m,n-l+1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{kl} & \cdots & x_{a+k-1,b+l-1} & \cdots & x_{mn} \end{bmatrix},
$$

$$
= K_{\text{flat}} X'
$$

where K_{flat} is the flattened kernel K, and the c^{th} column in X' contains the x-elements that are summed over in the c^{th} element of $(X * K)_{\text{flat}}$. Obtaining K_{flat} only requires a reshaping of K, so it is very simple. The more difficult part is obtaining the correct elements in X'. To do this numerically, we will create arrays using Numpy[16], that will store the indices of x at each element in X'. If we look at the first column of X', we see that the row-indices follow a pattern where we start with l ones, then l twos, up until we have l k's. Numerically we represent this with the vector i_0 given by

$$i_0 = \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 2 \\ \vdots \\ 2 \\ \vdots \\ k \\ \vdots \\ k \end{bmatrix},$$

with $k \cdot l$ elements. Looking at the first row of X', we see that the row-indices should follow a similar pattern, but they should range from 1 to $m - k + 1$, and each integer should be repeated $n - l + 1$ times. This gives the vector i_1 given by

$$i_1 = \begin{bmatrix} 1 & \dots & 1 & 2 & \dots & 2 & \dots & k & \dots & k \end{bmatrix},$$

with $(m - k + 1) \cdot (n - l + 1)$ elements. If we add $i_0 + i_1$ in Numpy, we get a matrix where each element is a sum of the corresponding row of i_0 and column of i_1. We name the resulting matrix i. Note that in Python, indices start at 0, not 1, so the numbers given here are a bit misleading. For instance, the first element (first row and first column) of i looks like it would be $1 + 1 = 2$, but in Python this becomes $0 + 0 = 0$, so it gives the index of the first row in X', not the second. Keeping this in mind, we have that any element of the matrix i contains the row-index of the x-value in the corresponding element in X'.

We do a similar thing for the column-indices of X'. If we look at the first column, we see that the column-indices follow a pattern of going through 1 to l, before starting over, and repeats this k times. Thus we get k repetitions of the range 1 to l, and

represent this with the vector j_0 given by

$$j_0 = \begin{bmatrix} 1 \\ \vdots \\ l \\ 2 \\ \vdots \\ l \\ \vdots \\ k \\ \vdots \\ l \end{bmatrix},$$

with $k \cdot l$ elements. Looking at the first row of X', we find a similar pattern where the column-indices go from 1 to $n - l + 1$, and then repeats this $m - k + 1$ times. We represent this with the vector j_1 given by

$$j_1 = \begin{bmatrix} 1 & \dots & n-l+1 & 1 & \dots & n-l+1 & \dots & 1 & \dots & n-l+1 \end{bmatrix},$$

with $(m - k + 1) \cdot (n - l + 1)$ elements. We can add $j_0 + j_1$ as we did for i_0 and i_1 in Numpy, and get a matrix j consisting of the column indices of x for each element in X'. We can then find X' in Numpy by obtaining the correct indices of X, which is done using X[i,j] in Numpy. The recipe for finding the convolution $X * K$ is then the following:

1. Find the index-matrices i and j. This can be done effectively using Numpy's methods *repeat* and *tile*.
2. Find X' from X, i and j.
3. Compute $(X * K)_{\text{flat}} = K_{\text{flat}} X'$.
4. Reshape the flattened $(X * K)_{\text{flat}}$ to get $X * K$.

And in this way we have an optimized way to compute the convolution of two matrices by reducing the operation to a matrix multiplication. When implementing this in our code we need to consider extra dimensions, since we store inputs and kernels as four-dimensional arrays, and because of this we need to consider each use case a bit differently to make sure all dimensions behave as we want them to. The method outlined in this section is, however, the general idea of how we compute the convolutions, and is used consistently in all cases met in our code.

6.4 CNN backpropagation derivation

The output of the convolutional layer is

$$z = X * K = \begin{bmatrix} \sum_{i=1}^{k} \sum_{j=1}^{1} x_{ij} k_{ij} & \cdots & \sum_{i=1}^{k} \sum_{j=1}^{1} x_{m-i+1,j} k_{ij} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{k} \sum_{j=1}^{1} x_{m-i+1,j} k_{ij} & \cdots & \sum_{i=1}^{k} \sum_{j=1}^{l} x_{m-i+1,n-j+1} k_{ij} \end{bmatrix}$$

When backpropagating through the network we need to compute $\frac{\partial C}{\partial x_{ij}}$, $\frac{\partial C}{\partial k_{ij}}$, $\frac{\partial C}{\partial b_{ij}}$. If we assume that we have already backpropagated through the other layers, such that we have $\frac{\partial C}{\partial a}$, where $a = \sigma(z)$ and σ is the activation function. We define the delta matrix $\delta = \frac{\partial C}{\partial a} \frac{\partial a}{\partial z} = \frac{\partial C}{\partial a} \sigma'(z)$ to simplify calculations.

6.4.1 Gradient with respect to the bias

We compute the gradient of the cost function with respect to the bias $\frac{\partial C}{\partial b}$ component-wise.

$$\begin{aligned} \frac{\partial C}{\partial b_{ij}} &= \sum_{n,m} \frac{\partial C}{\partial a_{nm}} \frac{\partial a_{nm}}{\partial z_{nm}} \frac{\partial z_{nm}}{\partial b_{ij}} \\ &= \sum_{n,m} \frac{\partial C}{\partial a_{nm}} \frac{\partial a_{nm}}{\partial z_{nm}} \delta_{in} \delta_{jm} \\ &= \frac{\partial C}{\partial a_{ij}} \frac{\partial a_{ij}}{\partial z_{ij}} \\ \frac{\partial C}{\partial b} &= \delta, \end{aligned}$$

where δ_{in} and δ_{jm} denote the Kronecker delta, not the delta matrix defined above.

6.4.2 Gradient with respect to the kernel

We compute the gradient of the cost function with respect to the kernel $\frac{\partial C}{\partial K}$ element-wise.

$$\begin{aligned}
\frac{\partial C}{\partial k_{ij}} &= \sum_{n,m} \frac{\partial C}{\partial a_{nm}} \frac{\partial a_{nm}}{\partial z_{nm}} \frac{\partial z_{nm}}{\partial k_{ij}} \\
&= \sum_{n,m} \frac{\partial C}{\partial a_{nm}} \frac{\partial a_{nm}}{\partial z_{nm}} \frac{\partial}{\partial k_{ij}} \left[\sum_{u,v} x_{n-1+u,m-1+v} k_{uv} \right] \\
&= \sum_{n,m} \frac{\partial C}{\partial a_{nm}} \frac{\partial a_{nm}}{\partial z_{nm}} x_{n-1+u,m-1+v} \\
&= \sum_{n,m} \delta_{nm} x_{n-1+u,m-1+v} \\
&= (X * \delta)_{ij} \\
\frac{\partial C}{\partial K} &= K * \delta,
\end{aligned}$$

where this time, δ_{nm} refers to the delta-matrix, not the Kronecker delta (confusing, I know).

6.4.3 Gradient with respect to the input

We compute the gradient of the cost function with respect to the input $\frac{\partial C}{\partial X}$ element-wise.

$$\begin{aligned}
\frac{\partial C}{\partial x_{ij}} &= \sum_{n,m} \frac{\partial C}{\partial a_{nm}} \frac{\partial a_{nm}}{\partial z_{nm}} \frac{\partial z_{nm}}{\partial x_{ij}} \\
&= \sum_{n,m} \delta_{nm} \frac{\partial}{\partial x_{ij}} \left[\sum_{u,v} x_{n-1+u,m-1+v} k_{uv} \right] \\
&= \sum_{n,m} \delta_{nm} \begin{cases} k_{i+1-n,j+1-m} & \text{if } i \in [a, a+k-1] \text{ and } j \in [b, b+l-1] \\ 0 & \text{else} \end{cases} \\
&= (\delta \circledast K)_{ij} \\
\frac{\partial C}{\partial x} &= \delta \circledast K,
\end{aligned}$$

where $\circledast$ denotes a full convolution. Remember that the $*$ operation we have called a convolution is really (in the mathematical sense) a correlation, but when we now introduce $\circledast$ as a full convolution, we actually mean convolution (that is, a correlation where the kernel is rotated 180 degrees). Thus, if we define δ' to be the matrix δ with a padding such that $\delta' * K$ gives a *full* correlation between δ and K, and we define K^R to be the 180° rotated kernel K, we can express the full convolution used above as $\delta \circledast K = \delta' * K^R$.

References

[1] M. Bansal, *Image processing in autonomous vehicles: Seeing the road ahead*, Medium, 2023. [Online]. Available: https://medium.com/@mohanjeetbansal777/image-processing-in-autonomous-vehicles-seeing-the-road-ahead-b400d176f877 (visited on 12/14/2023).

[2] R. C. Toppdahl and M. E. Mullis, "Snart vil kunstig intelligens analysere kroppen din," *NRK*, [Online]. Available: https://www.nrk.no/rogaland/xl/snart-vil-kunstig-intelligens-analysere-kroppen-din_-_-vi-er-for-darlig-forberedt-1.16553955.

[3] A. Krizhevsky, "Learning multiple layers of features from tiny images," *Computer Science University of Toronto. Available: https://www.cs.toronto.edu/~kriz/cifar.html*, 2009, Article with info on the dataset: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[4] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010, The dataset is obtained using tensorflow, see https://www.tensorflow.org/datasets/catalog/mnist.

[5] M. Abadi, A. Agarwal, P. Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[6] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning* (Springer Series in Statistics), Second. Springer, New York, 2009, pp. xxii+745, Data mining, inference, and prediction, ISBN: 978-0-387-84857-0. DOI: 10.1007/978-0-387-84858-7. [Online]. Available: https://doi.org/10.1007/978-0-387-84858-7.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning* (Adaptive Computation and Machine Learning). MIT Press, Cambridge, MA, 2016, pp. xxii+775, ISBN: 978-0-262-03561-3.

[8] K. O'Shea and R. Nash, *An introduction to convolutional neural networks*, 2015. arXiv: 1511.08458 [cs.NE].

[9] W. Commons, *File:comparison convolution correlation.svg — wikimedia commons, the free media repository*, [Online; accessed 26-May-2024], 2024. [Online]. Available: https://commons.wikimedia.org/w/index.php?title=File:Comparison_convolution_correlation.svg&oldid=866704127.

[10] J. Zheng, L. Ma, Y. Wu, L. Ye, and F. Shen, "Nonlinear dynamic soft sensor development with a supervised hybrid cnn-lstm network for industrial processes," *ACS Omega*, vol. 7, p. 16 655, May 2022. DOI: 10.1021/acsomega.2c01108.

[11] M. Yani, S Irawan, and C. Setianingsih, "Application of transfer learning using convolutional neural network method for early detection of terry's nail," *Journal of Physics: Conference Series*, vol. 1201, p. 3, May 2019. DOI: 10.1088/1742-6596/1201/1/012052.

[12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, 1998, pp. 2278–2324. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," vol. 25, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[14] S. Ruder, *An overview of gradient descent optimization algorithms*, 2017. arXiv: 1609.04747 [cs.LG].

[15] S. Yegulalp, *What is tensorflow? the machine learning library explained*, [Online; accessed 05-June-2024], 2024. [Online]. Available: https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html.

[16] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.