University of Oslo

FYS-STK4155

Applied Data Analysis and Machine Learning

Project 3: Comparing Convolutional and Dense Neural Networks for Image Classification

Author:
Håkon Fossheim
December 24, 2022

In this project, we investigate how the performance of dense and convolutional neural networks (DNNs and CNNs) scale as the pixel count and in-class variability of images increase.

We tune a combination of our own, as well as Keras's models for classification on three data sets of varying resolution and discernibility, containing images of digits from 0 to 9.

It was found that for the 8×8 MNIST dataset consisting of grayscale images of handwritten digits, the DNN outperformed the CNN, giving a test accuracy and training time of (94.2%, 6.8 sec) and (88.9%, 1min and 12sec), respectively.

Increasing the resolution to the full 28×28 MNIST data set resulted in the convolutional network superceding the dense. The accuracy and training time of the DNN and CNN were (88.7%, 2min and 22sec) and (95.5%, 1min and 54sec), respectively.

The last and most challenging set, the SVHN (Street View House Numbers) consists of 32×32 resolution color images of house numbers taken by google street view. No DNNs exceeding an accuracy of 30% were found. A CNN inspired by LeNet-5 was able to achieve an accuracy of 82.6% with a training period of 5min, 16sec which was outperformed by a hand-picked convolutional model achieving 88.8% accuracy with a training period of 10min, 44sec.

I. Introduction

In performing image classification, we are often interested in discerning certain objects subject to various distortions such as variation in lighting, position, rotation, object shape etc.

What makes this feasible is that objects of a given category share high-level features that endure the distortions even though the pixel values change from image to image. For complex objects, these high-level features will be comprised of lower-level features[1] with a particular 2D shape that may undergo displacements, rotations and shifts from between in-class samples, but retaining its "form".

Feeding the image to a fully connected neural network requires transforming it to a one-dimensional array. Now the puzzle pieces are "shuffled" anew for each in-class variation, and the network presumably has to learn a

[1] Such as edges, corners and curves.

transformation (which depends on the dimensions of the image) in which this "form" is restored, necessitating more training data.
Furthermore, a fully connected network has to train separate sets of weights for detecting the same feature in different parts of the image. This is highly sub-optimal, as it necessitates a large number of weights, which are computationally expensive to train. Instead of storing the location of the features in the weights, CNNs store them contiguously to their original position in a 2D *feature map*. This is made possible through *weight sharing*. Each node of the map is connected by the same set of weights, a *kernel*, to the corresponding region in the image, called the node's receptive field. A matrix dot-product is performed between these weights and the activations contained in the receptive field to yields a measure of the presence of a feature defined by the weights. This architecture is inspired by the visual cortex, where certain neurons respond only to designated regions of the visual field[5], the sum of the receptive fields of these neurons often overlap and their sum covers the whole visual.
Building on the previous project[3], we implement our own CNN, but now with modular layers. First, we present the theory behind the convolutional neural network including a derivation of the backpropagation algorithm used used in our code. The Methods section give a short overview of how things were implemented before the results are presented and discussed.
The appendix contains derivations for the stride-and-padding enthusiasts.

II. Theory

At the heart of the convolutional layer is the process of "scanning" the input for local features by use of two-dimensional kernel(s). We start by presenting the 2D case, being the simplest possible convolutional layer, before extending to 3D. We found this conceptual decomposition useful, particularly in backpropagation.

A. Forward propagation, 2D convolutional layer

As mentioned, each node of feature map in layer l is connected by the same set of weights to its corresponding receptive field in the input, seen in Fig.(2). This is implemented by stepping a kernel, K^l, containing this set of weights across the input and taking the dot product of the weights with the overlapping nodes of the input.

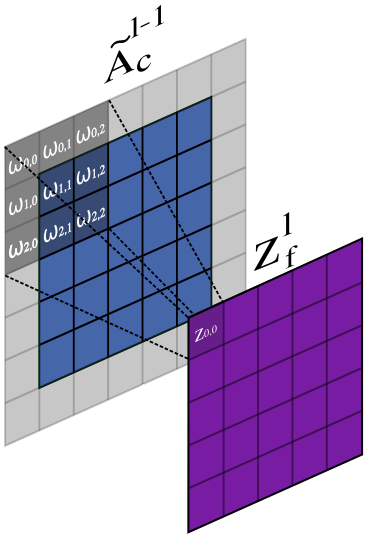

FIG. 1. Here we see a snapshot of the cross-correlation procedure in 2D. A kernel belonging to layer l steps across a padded feature map of the previous layer, measuring the presence of a kernel-specific feature in the overlapping activations and storing the result of these measurements as corresponding elements of Z^l. This is then passed through a non-linear activation function such as the ReLU or the Sigmoid to produce the feature map A^l.

For a filter of size $k \times k$, a single step is given by

$$z^l_{i,j} = \sum_{u,v=0}^{k-1} \omega^l_{u,v} \tilde{a}^{l-1}_{i+u,j+v} = \left(K^l \star \tilde{A}^{l-1}\right)_{i,j} \tag{1}$$

Which is the *cross-correlation*[2] of K^l and $\tilde{A}^{l-1}$.
Rotating the kernel 180° and performing the cross-correlation is equivalent to *convolution*:

$$\left(rot180°(K^l) \star \tilde{A}^{l-1}\right)_{i,j} = \sum_{u,v=0}^{k-1} \omega^l_{(w-1)-u,(w-1)-v} \tilde{a}^{l-1}_{i+u,j+v} \tag{2}$$

$$= \left(K^l * \tilde{A}^{l-1}\right)_{i,j} \tag{3}$$

Whether one uses cross-correlation or convolution for forward propagation is irrelevant, as the kernels learned by the respective network will be rotated 180° relative to each other, rendering them equivalent.
We try to abstain from calling Eq.(1) convolution, as is normally done, since this would require coming up with another name for the true convolution that we will use in backpropagation.

[2] Often just referred to as convolution.

B. Forward propagation, 3D convolutional layer

Typically, for CNN's to be useful, we need to give the network more building blocks from which it can assemble the high-level features. Which results in a separate feature map for each filter. To "combine" the features maps of the previous layer these filters also need to have a number of *channels*, corresponding to the depth of the input, i.e. number of channels/featuremaps in the input.

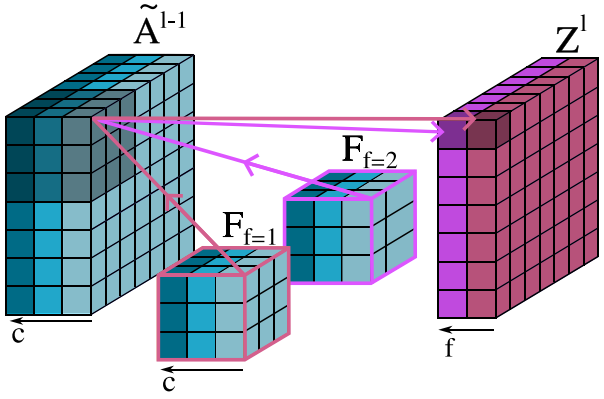

FIG. 2. Here we see a snapshot of the cross-correlation process in a layer with two filters containing three channels, matching the depth of the input. Different channels/filters are represented by different colors. Each filter steps across the input in the same manner as before, but now the input contains multiple feature maps, which are assigned a separate kernel in each filter. As the kernels of a filter cross-correlate across their respective feature maps, the sum is stored in a filter-specific feature map. The depth of the output will therefore be given by the number of filters used.

The results of the cross-correlations performed by a filter f are given by

$$Z_f^l = \sum_c \left(K_c^f \star \tilde{A}_c^{l-1} \right) + b_f \tag{4}$$

Which is passed through the layer-specific activation function to yield the activations of feature map f:

$$A_f^l = f(Z_f^l) \tag{5}$$

C. Backpropagation in 2D Convolutional Layers

In constructing $\frac{\partial C}{\partial \theta^l}$ we need to trace all paths from θ^l to C, applying the chain rule along the way to find how much a tiny nudge to θ^l changes the cost function through a particular path, i.e. its gradient. We find the total contribution by summing over all these paths.

$$\frac{\partial C}{\partial \theta^l} = \sum_{i,j \in Z^l} \frac{\partial z_{i,j}^l}{\partial \theta^l} \underbrace{\frac{\partial C}{\partial z_{i,j}^l}}_{\delta_{i,j}^l} \tag{6}$$

$$= \sum_{i,j \in Z^l} \frac{\partial z_{i,j}^l}{\partial \theta^l} \sum_{m,n \in Z^{l+1}} \frac{\partial z_{m,n}^{l+1}}{\partial z_{i,j}^l} \underbrace{\frac{\partial C}{\partial z_{m,n}^{l+1}}}_{\delta_{m,n}^{l+1}} \tag{7}$$

We see that only the first partial derivative depends on the parameter, whilst the rest of the gradient depends on the network architecture. We therefore wish to calculate the architecture-based gradients once for each layer and reuse it for all parameter updates of that layer.
The way we do this is by storing the pre-activation[3] gradients from all paths originating in the nodes of layer l contiguously in an error matrix δ^l with the same shape. The layer's parameters are then updated using this error matrix.
Because of the nested structure of a feedforward neural network, the errors of a layer are given by those of the layer before it and so on, i.e. $\delta^l = f_l(\delta^{l+1}(f_{l+1}(...))$. Errors should therefore be calculated from the last layer and backwards so as to avoid redundant calculations, thus the name *backpropagation*.
For this, we need only to find a formula for δ^l as a function of δ^{l+1}. In project 2 [reference proj], we found this relation when layer $l+1$ was a densely connected layer, and so we only need to derive an expression for when the upstream layer is a convolutional one[4].

Recursive error relation

Coming back to Eq.(6) and expanding to include the activation yields

$$\frac{\partial C}{\partial \theta^l} = \sum_{i,j \in Z^l} \frac{\partial z_{i,j}^l}{\partial \theta^l} \overbrace{\frac{\partial a_{i,j}^l}{\partial z_{i,j}^l} \underbrace{\sum_{m,n \in Z^{l+1}} \underbrace{\frac{\partial z_{m,n}^{l+1}}{\partial a_{i,j}^l}}_{(i)} \underbrace{\frac{\partial C}{\partial z_{m,n}^{l+1}}}_{\delta_{m,n}^l}}_{(I)}}^{\delta_{i,j}^l} \tag{8}$$

After cross-correlation has been performed on layer l, only a specific set of nodes in layer $l+1$ will depend on $a_{i,j}^l$, i.e. those whose visual fields include $a_{i,j}^l$. We therefore only need to sum over the "field of influence" of $a_{i,j}^l$

[3] The gradients passed on from above is then multiplied by the $f'(Z^l)$ post-transfer, which allows for cleaner and more modular code.
[4] As well as for a max pool layer, but this is a simpler task.

in calculating (I) rather than the the entire layer. Furthermore, we need to calculate (i) for each path, which will just be the kernel weight that overlapped $a^l_{i,j}$ during the cross-correlation step that produced $z^{l+1}_{m,n}$.
Fig.(3) tries to depict this, where one can imagine the filter in $\tilde{A}^l$ (gray) sliding over the stationary blue and green boxes to produce their "field of influence" in layer $l+1$.

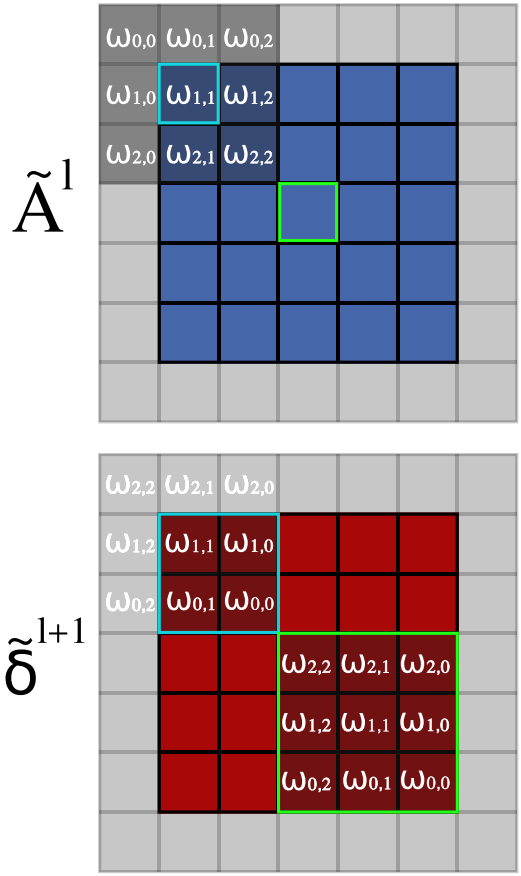

FIG. 3. Figure visualizing which nodes in the next layer depend on those of the previous layer. We see that the nodes in layer $l+1$ through which node (i,j) of layer l can affect the cost function is given by placing the filter on node (i,j) of layer $l+1$, which needs to be appropriately padded. By rotating the filter 180 degrees, the weights that contribute overlap the nodes of layer $l+1$.

We see that the summed contribution of paths originating in $a^l_{i,j}$, i.e. (I), is given by a cross-correlation step on $\tilde{\delta}^{l+1}$, but with the kernel rotated 180°, which is just convolution.
We thus have

$$\underline{(I) = \left(K^{l+1} * \tilde{\delta}^{l+1}\right)_{i,j}} \tag{9}$$

We note that the errors of layer $l+1$ may need to be padded to ensure correct placement of the rotated kernel. The appropriate padding can be found by realizing that the shape of δ^l produced by the convolution process must be the same as A^l. From repeated application of Eq.(...) the padding size is found to be

$$p_\delta = (w-1) - p \tag{10}$$

A more rigorous derivation of Eq.(9) is given in the appendix, from which same padding formula pops out.
Now, returning to Eq.(8), we have

$$\delta^l_{i,j} = \frac{\partial a^l_{i,j}}{\partial z^l_{i,j}}(I) \tag{11}$$

Which inserted for Eq.(9) yields

$$\delta^l_{i,j} = f'(z^l_{i,j})\left(K^{l+1} * \tilde{\delta}^{l+1}\right)_{i,j} \tag{12}$$

Or

$$\boxed{\delta^l = f'(Z^l) \circ \left(K^{l+1} * \tilde{\delta}^{l+1}\right)} \tag{13}$$

Parameter updating With an expression for $\delta^l_{i,j}$ at hand, Eq.(8) can be simplified greatly;

$$\frac{\partial C}{\partial \theta^l} = \sum_{i,j \in Z^l} \frac{\partial z^l_{i,j}}{\partial \theta^l}\delta^l_{i,j} \tag{14}$$

$$z^l_{i,j} = \sum_{u,v=0}^{w-1} \omega^l_{u,v}\tilde{a}^{l-1}_{i+u,j+v} + b_l \tag{15}$$

$$\frac{\partial z^l_{i,j}}{\partial \theta^l} = \frac{\partial}{\partial \theta^l}\left(b_l + \sum_{u,v=0}^{w-1}\omega^l_{u,v}\tilde{a}^{l-1}_{i+u,j+v}\right) \tag{16}$$

$$\frac{\partial z^l_{i,j}}{\partial \theta^l} = \begin{cases} \tilde{a}^l_{i+u',j+v'}, & \text{if } \theta^l = w^l_{u',v'} \\ 1, & \text{if } \theta^l = b_l \end{cases} \tag{17}$$

For $\theta^l = w^l_{u',v'}$ we then get

$$\frac{\partial C}{\partial w^l_{u',v'}} = \sum_{i,j \in Z^l}\delta^l_{i,j}\tilde{a}^l_{u'+i,v'+j} = \left(\delta^l \star \tilde{A}^{l-1}\right)_{u',v'} \tag{18}$$

Which we can rewrite as the following matrix equation:

$$\boxed{\frac{\partial C}{\partial K^l} = \delta^l \star \tilde{A}^{l-1}}$$

For $\theta^l = b_l$, we have

$$\boxed{\frac{\partial C}{\partial b_l} = \sum_{i,j \in Z^l}\delta^l_{i,j} = I^l \star \delta^l} \tag{20}$$

Where I^l is a matrix of same shape as δ^l and filled with ones. It is worth noting that here we use one bias per feature map

D. Backpropagation in 3D Convolutional Layers

We define i, j, f as the index variables over the height, width and depth of the activations of layer l, respectively. The corresponding variables of layer $l+1$ are $m, n, \tilde{f}$. The channel number of a kernel in a given filter is given by c. Instances of these variables are bold-faced. Filter f of layer l is denoted as $\mathbf{F}^{l,f}$. The kernel K_c^f is the 2D slice of filter f at channel c, while $\omega_{u,v,c}^f$ represents a single weight/element in the filter. The planes of $\mathbf{Z}, \mathbf{A}$ and $\boldsymbol{\delta}$ produced by separate cross-correlations are denoted by non-boldface uppercase letters (Z_f^l, A_f^l, δ_f^l) and individual elements of these by the corresponding lowercase letters.

Because $\omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}$ is only used in the production of $A_{f=\mathbf{f}}^l$, we only need to sum the paths going through the nodes of this one feature map, as was done in the 2D case.

$$\frac{\partial C}{\partial \omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}} = \sum_{i,j} \frac{\partial z_{i,j,\mathbf{f}}^l}{\partial \omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}} \underbrace{\frac{\partial a_{i,j,\mathbf{f}}^l}{\partial z_{i,j,\mathbf{f}}^l} \overbrace{\frac{\partial C}{\partial a_{i,j,\mathbf{f}}^l}}^{\textbf{(I)}}}_{\delta_{i,j,\mathbf{f}}^l} \tag{21}$$

$$\textbf{(I)} = \sum_{\tilde{f}} \sum_{m,n} \underbrace{\frac{\partial z_{m,n,\tilde{f}}^{l+1}}{\partial a_{i,j,\mathbf{f}}^l}}_{\textbf{(i)}} \underbrace{\frac{\partial C}{\partial z_{m,n,\tilde{f}}^{l+1}}}_{\delta_{m,n}^{l+1}} \tag{22}$$

As seen, calculating **(I)** now requires summing over all feature maps of layer $l+1$, since each filter adds a contribution to its feature map from cross-correlating $A_{\mathbf{f}}^l$ with its corresponding kernel.

$$\textbf{(i)} = \frac{\partial z_{m,n,\tilde{f}}^{l+1}}{\partial a_{i,j,\mathbf{f}}^l} = \frac{\partial}{\partial a_{i,j,\mathbf{f}}^l} \sum_{\tilde{c}} \left(K_{\tilde{c}}^{\tilde{f}} \star \tilde{A}_{\tilde{c}}^l \right)_{m,n} \tag{23}$$

Only channel $\tilde{c} = \mathbf{f}$ contributes, and so we get

$$(i) = \frac{\partial}{\partial a_{i,j,\mathbf{f}}^l} \left(K_{\mathbf{f}}^{\tilde{f}} \star \tilde{A}_{\mathbf{f}}^l \right)_{m,n} \tag{24}$$

Which is the same expression we had in the 2D case. Inserting this gives into (I) and using the result found in the 2D case gives

$$(I) = \sum_{\tilde{f}} \sum_{m,n} \frac{\partial}{\partial a_{i,j,\mathbf{f}}^l} \left(K_{\mathbf{f}}^{\tilde{f}} \star A_{\mathbf{f}}^l \right)_{m,n} \delta_{m,n,\tilde{f}}^{l+1} \tag{25}$$

$$(I) = \sum_{\tilde{f}} \left(K_{\mathbf{f}}^{\tilde{f}} * \tilde{\delta}_{\tilde{f}}^{l+1} \right)_{i,j} \tag{26}$$

From Eq.(21) we have that

$$\delta_{i,j,\mathbf{f}}^l = \frac{\partial a_{i,j,\mathbf{f}}^l}{\partial z_{i,j,\mathbf{f}}^l} \cdot (I) \tag{27}$$

$$= f'\left(z_{i,j,\mathbf{f}}^l\right) \sum_{\tilde{f}} \left(K_{\mathbf{f}}^{\tilde{f}} * \tilde{\delta}_{\tilde{f}}^{l+1} \right)_{i,j} \tag{28}$$

$$\delta_{i,j,\mathbf{f}}^l = f'\left(z_{i,j,\mathbf{f}}^l\right) \left(\sum_{\tilde{f}} K_{\mathbf{f}}^{\tilde{f}} * \tilde{\delta}_{\tilde{f}}^{l+1} \right)_{i,j} \tag{29}$$

And so

$$\boxed{\delta_{\mathbf{f}}^l = f'(Z_{\mathbf{f}}^l) \circ \sum_{\tilde{f}} K_{\mathbf{f}}^{\tilde{f}} * \tilde{\delta}_{\tilde{f}}^{l+1}} \tag{30}$$

Where we ignored the layer index $(l+1)$ for the kernel, as the filter index $\tilde{f}$ tells us that it belongs to the layer succeeding l.

Now, returning to Eq.(21), we had that

$$\frac{\partial C}{\partial \omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}} = \sum_{i,j} \frac{\partial z_{i,j,\mathbf{f}}^l}{\partial \omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}} \delta_{i,j,\mathbf{f}}^l \tag{31}$$

Where

$$\frac{\partial z_{i,j,\mathbf{f}}^l}{\partial \omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}} = \frac{\partial}{\partial \omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}} \left(\sum_c K_c^f \star \tilde{A}_c^{l-1} \right)_{i,j} \tag{32}$$

,

Only the cross-correletion where $c = \mathbf{c}$ involve $w_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}$ and so we get

$$\frac{\partial z_{i,j,f}^l}{\partial \omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}} = \frac{\partial}{\partial \omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}} \left(K_{\mathbf{c}}^f \star \tilde{A}_{\mathbf{c}}^{l-1} \right)_{i,j} \tag{33}$$

$$= \frac{\partial}{\partial \omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}} \sum_{u,v=0}^{-1} \omega_{u,v,\mathbf{c}}^{\mathbf{f}} \tilde{a}_{i+u,j+v,\mathbf{c}}^{l-1} + b_{\mathbf{f}} \tag{34}$$

$$= \tilde{a}_{i+\mathbf{u},j+\mathbf{v},\mathbf{c}}^{l-1} \tag{35}$$

Inserting this back into the expression for $\frac{\partial C}{\partial \omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}}$ yields

$$\frac{\partial C}{\partial \omega_{\mathbf{u},\mathbf{v},\mathbf{c}}^{\mathbf{f}}} = \sum_{i,j} \delta_{i,j,\mathbf{f}}^l \tilde{a}_{\mathbf{u}+i,\mathbf{v}+j,\mathbf{c}}^{l-1} = \left(\delta_{\mathbf{f}}^l \star \tilde{A}_{\mathbf{c}}^{l-1} \right)_{\mathbf{u},\mathbf{v}} \tag{36}$$

$$\boxed{\frac{\partial C}{\partial K_{\mathbf{c}}^{\mathbf{f}}} = \delta_{\mathbf{f}}^l \star \tilde{A}_{\mathbf{c}}^{l-1}} \tag{37}$$

Where $\delta_{\mathbf{f}}^l$ taking a similar form to the 2D case, but now containing a sum over all filters in layer $l+1$.

I ncluding biases

Since the first term in Eq.(21) is the only one that depends on the parameter to be differentiated, this is all we need to recalculate. In this project, we use so-called *tied* biases, meaning that every element of $Z^l_{\mathbf{f}}$ contains the same bias term. With *untied* biases, every element of $Z^l_{\mathbf{f}}$ would be assigned a distinct bias. This increases the model complexity, however the same could be done by e.g. increase the number of filters or convolutional layers.

In our case

$$z^l_{\mathbf{i},\mathbf{j},\mathbf{f}} = \left(\sum_c K^{\mathbf{f}}_c \star \tilde{A}^{l-1}_c \right)_{\mathbf{i},\mathbf{j}} + b^l_{\mathbf{f}} \tag{38}$$

And so

$$\frac{\partial z^l_{\mathbf{i},\mathbf{j},\mathbf{f}}}{\partial b^l_{\mathbf{f}}} = 1 \tag{39}$$

From Eq.(31), we get

$$\frac{\partial C}{\partial b^l_{\mathbf{f}}} = \sum_{i,j} \frac{\partial z^l_{i,j,\mathbf{f}}}{\partial b^l_{\mathbf{f}}} \delta^l_{i,j,\mathbf{f}} = \sum_{i,j} \delta^l_{i,j,\mathbf{f}} \tag{40}$$

$$\boxed{\frac{\partial C}{\partial b^l_{\mathbf{f}}} = \mathbf{I}^l \star \delta^l_{\mathbf{f}}} \tag{41}$$

In summary, for a 3D convolutional layer we have

$$A^l_f = \sum_c \left(K^f_c \star \tilde{A}^{l-1}_c \right) + b_f \tag{42}$$

$$\boxed{\delta^l_{\mathbf{f}} = f'(Z^l_{\mathbf{f}}) \circ \sum_{\tilde{f}} K^{\tilde{f}}_{\mathbf{f}} * \delta^{l+1}_{\tilde{f}}} \tag{43}$$

$$\boxed{\frac{\partial C}{\partial K^{\mathbf{f}}_{\mathbf{c}}} = \delta^l_{\mathbf{f}} \star \tilde{A}^{l-1}_{\mathbf{c}}} \tag{44}$$

$$\boxed{\frac{\partial C}{\partial b^l_{\mathbf{f}}} = \mathbf{I}^l \star \delta^l_{\mathbf{f}}} \tag{45}$$

Note that cross-correlation and convolution are commutative operators[4] and so we could changing the order of operation is permitted. Doing so reveals the similarities with the backpropagation equations for the dense layer.

In our code, the CNN class allows for custom layer architectures. Layers therefore need to be able to reconstruct their error without access to the member variables of the layer upstream, since we don't know if it is a max pool/flattening layer etc. All layer-dependent calculations needed for reconstructing $\delta^l_{\mathbf{f}}$ are thus performed before the pass, meaning we pass $\sum_{\tilde{f}} K^{\tilde{f}}_{\mathbf{f}} * \delta^{l+1}_{\tilde{f}}$ rather than $\delta^{l+1}_{\tilde{f}}$ to the downstream layer. To allow feeding multiple samples at once, our code assumes the first axis/dimension of the data to be the sample index. Feeding a single sample is also possible, but a new axis/dimension must be added. Samples are then stored as a separate elements of A^l and δ^l along the first axis, making their shapes $(\mathrm{n_{samples}}, \mathrm{Height}, \mathrm{Width}, \mathrm{n_{channels}})$. Eq.(43) and Eq.(42) are applied separately to each sample in during propagation. Only at the last step, when updating the parameters, are their contributions summed together.

For the dense layers which will be used at the end of the network, we found in project 2[3] that for data of shape $(\mathrm{n_{samples}}, \mathrm{n_{features}})$, the backpropagation equations were given by

$$\underline{\boldsymbol{\delta}^L = f'(\mathbf{Z}^L) \circ \frac{\partial C}{\partial \mathbf{A}^L}} \tag{46}$$

$$\underline{\boldsymbol{\delta}^l = f'(\mathbf{Z}^l) \circ \boldsymbol{\delta}^{l+1} \left(\mathbf{W}^{l+1} \right)^T} \tag{47}$$

$$\underline{\frac{\partial C}{\partial \mathbf{W}^l} = \left(\mathbf{A}^{l-1} \right)^T \boldsymbol{\delta}^l} \tag{48}$$

$$\underline{\frac{\partial C}{\partial \vec{b}^l} = \boldsymbol{\delta}^l \cdot I_{n_l,1}} \tag{49}$$

Downsampling

Downsampling is a widely used method for reducing the pixel count of feature maps, which forces the network to learn high-level feature representations through information compression.

Furthermore, it reduces the number of cross-correlation and convolution operations required in forward and backward propagation respectively, thus reducing computational cost.

The most common implementation of downsampling is called max pooling. As in the convolutional layer, we step a filter/pool across the input, but now we simply extract the largest pixel value in the pool. Fig.5 depicts the pooling process using a stride of 2 and a 2×2 filter. In this example, the third column can be included in the pooling procedure, but only if zero-padding is added to the input. In max pooling, this is called *same* padding, whilst ignoring the non-fitting edge is referred to as *valid* padding.

Implementing this was not as straightforward as expected, due to all possible combinations of stride, filter

and image size that need to be considered. The derivation of our max pool padding procedure can be found in the appendix.

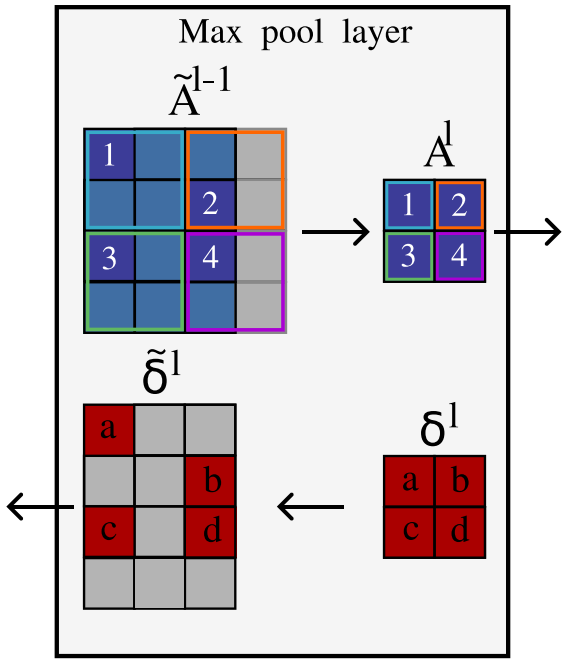

FIG. 4. Depiction of the feedforward and backpropagation methods in a max pooling layer. The position of the filter as it steps across the input is indicated by the colored squares. The maximum values contained in the pools are darkened and numbered according to the step number. Note that if the step size is smaller than the filter in a given dimension, the pools will overlap. This may result in the same pixel being selected multiple times. Elements of $\boldsymbol{\delta}^l$ originating from the same pixel of $\mathbf{A}^{l-1}$ must be summed together before being placed in their pre-pooling location.

Only the maximum values selected will contribute to the cost function from this layer onward. Therefore all errors not belonging to these values will be zero. Furthermore, calculating the error of layer $l-2$ requires positional information to be conserved[5]. We therefore store a mapping from A^l to the coordinates of the originating pixels that can be used in constructing $\boldsymbol{\delta}^l$ from $\boldsymbol{\delta}^{l+1}$. This is illustrated in Fig.5 where the grey pixels represent zeros. We also need to consider the case where the pool size is greater than the step size. This can result in the same pixel in $\mathbf{A}^{l-1}$ being selected in multiple steps. All errors in $\boldsymbol{\delta}^{l+1}$ originating from the same element of $\mathbf{A}^{l-1}$ must therefore be summed together and stored in the corresponding location in $\boldsymbol{\delta}^l$.
As seen, pooling results in loss of positional information about the features. However, the features which the filters have learned to be of discriminatory importance will still be largely preserved in max pooling.
Reducing/removing the positional dependence, i.e. making the feature maps invariant with respect to translation of features in the original image, is beneficial for the dense network since in-class translations of the same feature maps are not presented as separate instances that the dense network needs to learn to correlate with the same class, which as mentioned it is not optimized for due to the one dimensional layers.

[5] Otherwise the convolution given in Eq.?? picks up the wrong weights.

III. Methods

The class CNN allows for the creation of custom convolutional (or dense) neural networks by use of modular layers. Each layer type has a class associated with it, which can be added to an instance of the CNN through its *addLayer* member function:

```
model = CNN(name=my model)
model.addLayer( Conv((3,3),2,reluL,"same") )
model.addLayer( MaxPool((2,2),2,"same") )
model.addLayer( Flatten() )
model.addLayer( Dense(50,sigmoidL) )
model.addLayer( Dense(10,softmaxL) )
```

Layers are indexed from 0, which would correspond to the convolutional layer in this example. The input is therefore not considered a layer. Next, before initializing, we can provide the desired initialization scheme as follows:

```
model.scheme = "Xavier"
```

We include 3 different schemes, viz. *Xavier*, *He* and a default scheme. This default scheme initializes the weights according to a Gaussian with mean 0 and a variance of 1, if not provided with either of the two aforementioned schemes. The *Xavier* initialization sets this variance to $var = 1/n_{\text{prev}}$ where n_{prev} is the number of nodes of the previous layer. The *He* initialization sets it to $var = 2/n_{\text{prev}}$. Biases are initialized to 0 for all schemes. The weights can then be initialized by doing

```
model.initialize_weights(X_train.shape)
```

As seen, the model needs to be provided the shape of the input, which must be in the form of $(n_{inputs}, Height, Width, n_{channels})$. The number of inputs is superfluous as this is provided as arguments when predicting/training, but it makes life a little easier since this is the typical shape of our data.
Before training our model, we need to set the hyperparameters:

```
model.set_hyperparams(hyperparams)
```

Where

```
hyperparams = [epochs, batch_size,eta,lmbd]
```

```
model.train(X_train,y_train)
```

We can then feed (test) data through the trained model by use of *predict()*:

```
y_pred = model.predict(X_test)
```

Model selection

If starting from scratch, finding good hyperparameters for your model can be a daunting task. Choices need to be made on layer architectures, of which there are infinite possible combinations. When an architecture has been chosen, we must also specify the number of filters for each convolutional layer, kernel size, stride etc. as well the attributes for the max-pool and dense layers.
Picking an arbitrary model and performing a gridsearch of e.g. learning rate and regularization parameter is immensely time consuming, and this must be done for each viable model architecture to find the best one.
Instead, we rely on the existing research in the field and construct our model based on what has already been found to work.
LeNet is a CNN originally proposed in 1989 by Yann LeCun et al. [6] for handwritten zip code recognition. In the 1990s they introduced LeNet-5, marking a landmark for CNNs.
Our DNN was able to handle being fed the 10k test-set, but this was not the case for our CNN. The convolutional layers contain far more activations and thus take up more memory, resulting in the propagation process being killed. We therefore split the set and feed it through part by part when this is necessary. This is controlled in the evaluate_fit() function in benchmarking.py

Validation set

If one is performing hyperparameter tuning to maximize a model's performance on the test set, the resulting model will be optimized for the test data and the benchmarks may not be representative of what one would get for completely unseen data.
For this reason, one splits the data into a training, validation and test set. The validation set is used when tuning the model, and the final benchmarks are performed using the tuned model on the test set.

Testing our implementation

To test if the derived backpropagation equation and our implementation of them are correct, we use the method of finite differences.
A data sample is fed through the network and the cost function is evaluated. We then alter one of the weights of a kernel in the first layer and feed the same sample through again, evaluating how much the cost function differs from the first time. A numerical measure for the gradient of $\omega^f_{u,v,c}$, is then given by

$$\frac{\partial C}{\partial \omega^f_{u,v,f}} \approx \frac{\Delta C}{\Delta \omega^f_{u,v,c}} = \text{grad}_{\text{num}} \tag{50}$$

Which we can compare with what we found in our derivation. In Eq.(II D) we had the following expression for the gradient of a single weight

$$\frac{\partial C}{\partial \omega^f_{u,v,c}} = \left(\delta^l_f \star \tilde{A}^{l-1}_c\right)_{u,v} = \text{grad}_{\text{analytical}} \tag{51}$$

Performing this cross-correlation, we can then check whether $\text{grad}_{\text{num}} = \text{grad}_{\text{analytical}}$. To be sure, we do this for a whole kernel. If the gradients match, we can be pretty certain the equations for backpropagation are correct and have been implemented properly.

IV. Results

Typically in machine-learning one has to strike a balance between model complexity and predictive power. The predictive power of a model initially increases until it is complex enough to model the relevant features of the data, after which a decrease occurs due to incorporating irrelevant training-set specific features.
We will investigate how well the dense and convolutional neural network strike this balance by comparing their accuracy and measuring their training time.
We found initialization to matter greatly for the training of the DNN. With Sigmoid as activation function, no training occurred when initializing using the Xavier scheme. Changing to initializing with a variance of 1 solved this. Surprisingly, we had little success with the ReLU as activation function. Regardless of initialization scheme, the learning often stagnated and gridsearches came back empty-handed. We therefore relied mostly on the sigmoid, but sometimes also the hyperbolic tangent (tanh) as our activation function.
Due to not being optimized for speed, our python implemented CNN models often end up being too slow for tuning, especially to the SVHN dataset. Gridsearches were generally very time consuming and so our models are found mostly through trial and error. For the SVHN dataset, we exclusively utilize Keras with tensorflow backend in tuning the CNN model(s).
The models follow a loose[6] naming scheme to aid the tuning process. They are named according to their type, i.e. dense or convolutional as well as whether they rely on our implementation or Keras. We also specify the number of

[6] LeNet2 being the exception.

layers of the model and the number of nodes/filters in the name.
E.g. denseNet2_20 is a dense network from our implementation with 2 layers, with 20 nodes in the first layer while convKeras4_20 is a 4 layered CNN with 20 filters in the first layer implemented by use of Keras. The final layer of each model is a softmax layer with 10 nodes. Categorical cross-entropy is used as cost function and stochastig gradient descent is implemented as the optimization algorithm.

A. MNIST(8×8)

Our first dataset consists of 1797 samples which we split ourselves using a test size of 0.2. The images are very low resolution, with only 8×8 pixels and one color channel, i.e. they are in grayscale. Due to the low feature count of the images, we don not expect too much difficulty in applying the dense network. Furthermore, because of this low feature/pixel count, any kernel size we choose will be relatively large compared to the image, thus limiting the resolution of the features they are able to detect. This is likely to hamper the CNN performance at very low resolution.

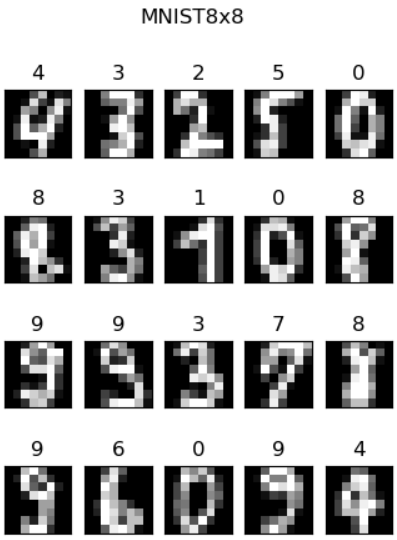

FIG. 5. Random selection of 20 samples from the MNIST(8x8) dataset[2] with labels. Aquired through Scikit-Learn. Resolution: 8x8x1, 1797 data samples in total.

The first network we apply is denseNet2_20. It is an instance of the class CNN with 2 fully connected layers with (20,20) nodes. We used sigmoidal activation with std. normal initialization.

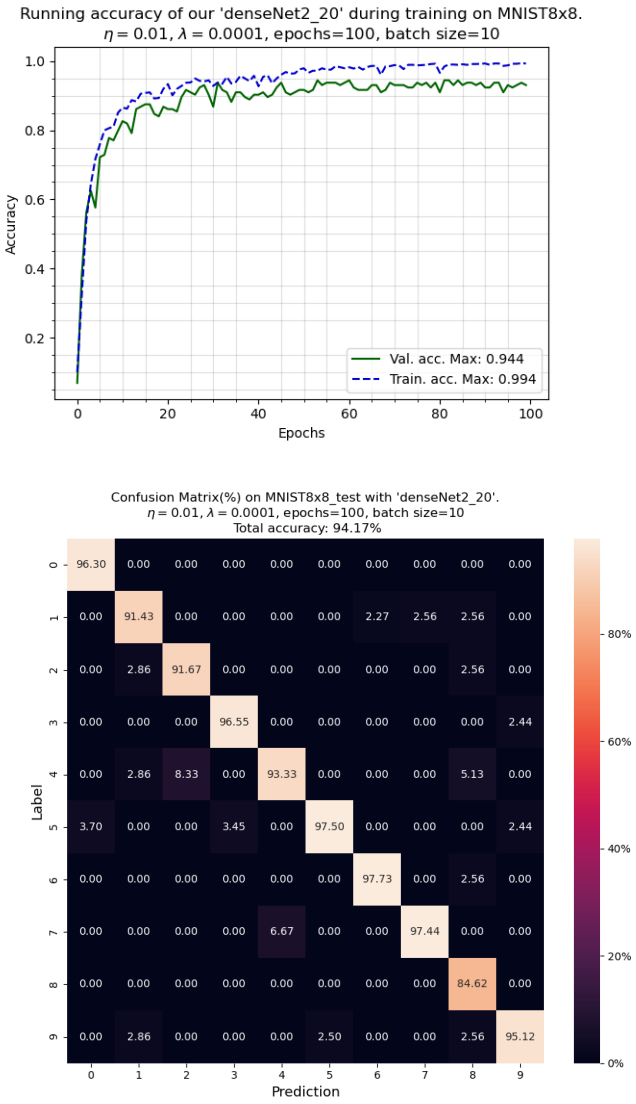

FIG. 6. Accuracy scores of our denseNet2_20. The model consists of 2 dense layers with (20,20) nodes using the sigmoid activation function. Initialized with std. normal distribution. Final test accuracy = 94.2% with 6.8 sec. training time.

As seen from the upper plot of Fig.(6), the accuracies quickly saturate and the training accuracy lowly diverges upwards from that of the validation as the epochs increases. However the validation accuracy keeps increasing (at a decreasing rate) and so we are not overfitting. The lower plot shows the confusion matrix of denseNet2_20 as well with the total test accuracy in the title. As seen, the accuracy is quite high for all categories, with the exception of 8. A total accuracy test of 94.2% with a training time of 6.8 s is achieved.

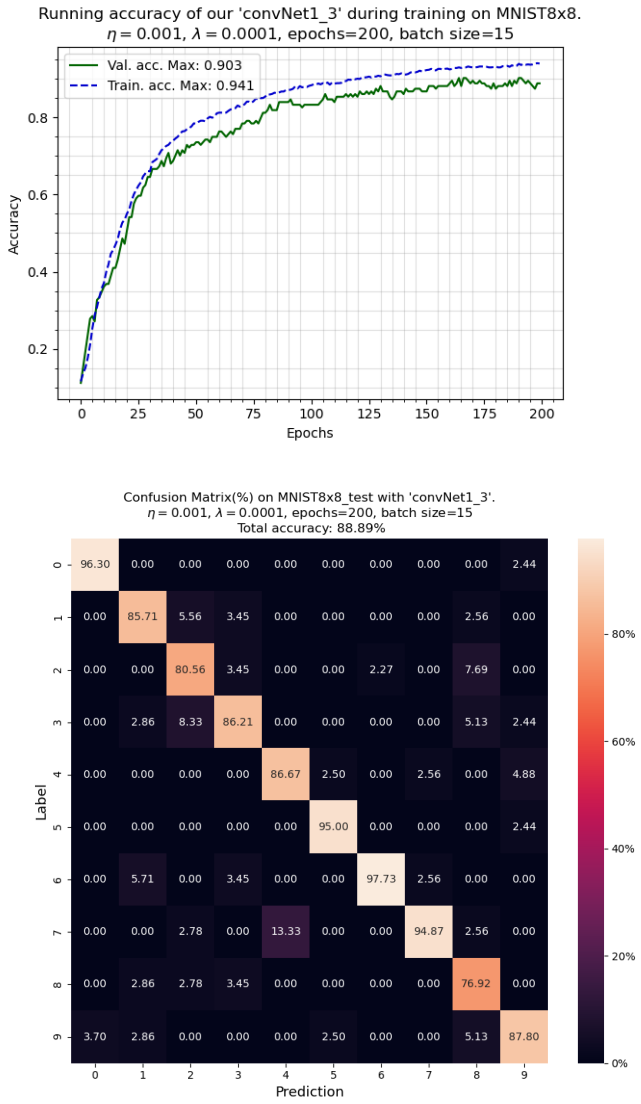

FIG. 7. Accuracy scores of our convNet1_3. The model consists of 1 convolutional layer with three 3×3 filters, followed by a dense layer of 20 nodes. All layers use sigmoid as activation. Initialized with std. normal distribution. Final test accuracy = 88.9% with 72.2s $\approx$ 1min,12sec training time.

We also use our own implementation for the CNN model. The best model found, convNet1_3, performed rather poorly in comparison to the previous DNN.
As seen in the upper plot of Fig.(7), the validation accuracy takes a lot longer to "saturate", not completely flattening out even after 200 epochs, indicating increase this number further could be beneficial.
The lower plot shows the best accuracy to be 97.7% for 6's, the same as with denseNet2_20. Both models show roughly the same relative distribution in accuracy over the classes.
The CNN having a test accuracy of only 88.9% and rougly 10 times the training time of the DNN, the dense network seems best suited for classifying the MNIST(8×8) dataset.

B. MNIST

We now look at the same data, but with higher resolution (28×28) and more samples (50 000 training samples, 10 000 test samples). This is expected to cause greater strain on the dense network in particular, which has to connect every node of the input layer to 784 inputs rather than 64 as was previously the case. Furthermore, the CNN should be able to pick up lower resolution features with its kernels.

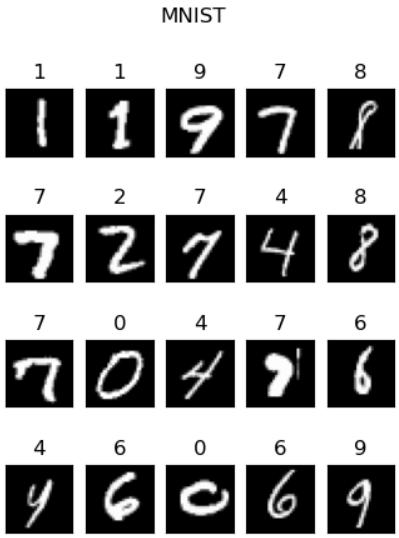

FIG. 8. Random selection of 20 samples from the MNIST dataset[7], with labels. Downloaded through keras. Resolution: 28x28x1, 50k training samples, 10k test samples.

In the upper plot of Fig.9 we see a sharp initial increase in both training and validation accuracy for denseNet4_300, before suddenly flatting out. Why this occurs, we are not sure. Additionally, the two curves overlap each other throughout the training process, indicating that the model is able to extract to learn the features relevant to the problem while ignoring the non-relevant features that may be specific to the training set.

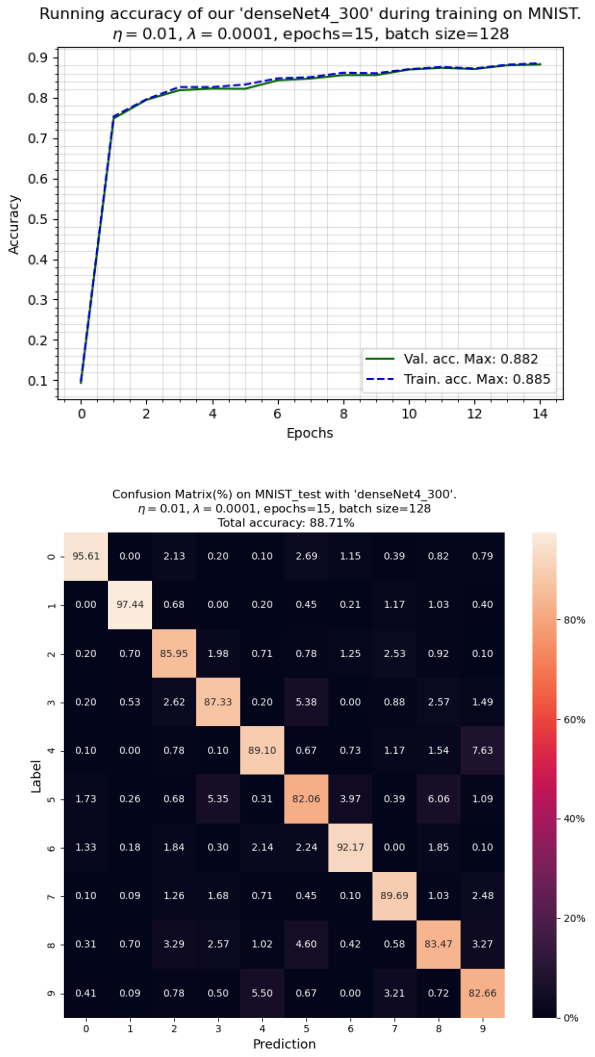

FIG. 9. Accuracy scores of our denseNet4_300. The model consists of 4 dense layers with (300,128,128,64) nodes using tanh as activation. Initialized with std. normal distribution. Final test accuracy = 88.7% with 142s ≈ 2min, 22sec training time.

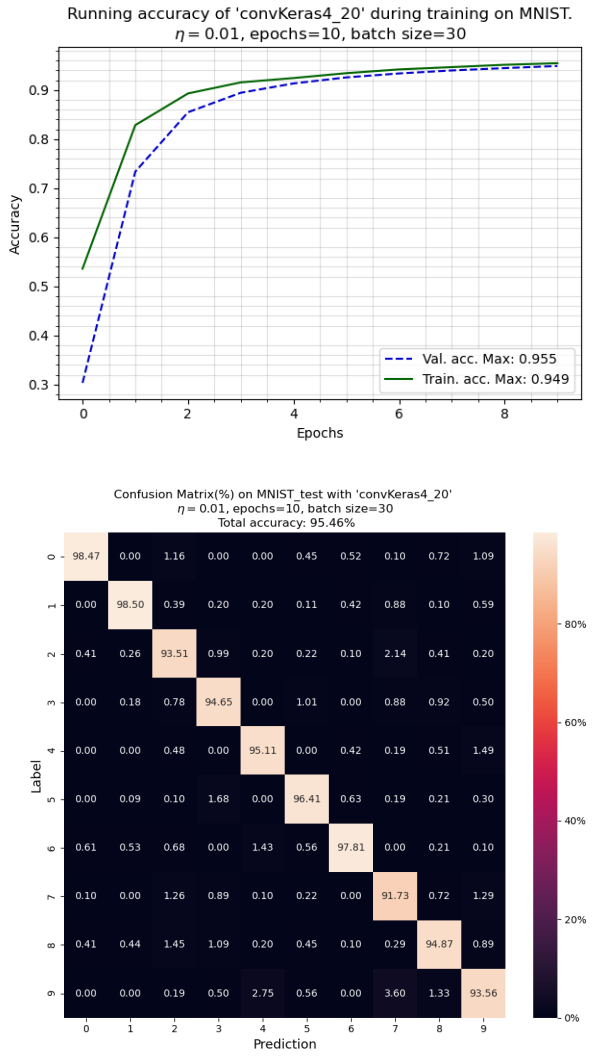

FIG. 10. Accuracy scores of Keras's convKeras4_20. The model consists of 2 convolutional layers of with 20 filters of size 3×3, each followed by a max pool layer. This is followed by a dense layer of 128 nodes. All layers use sigmoid activation. Final test accuracy = 95.5% with 113.9 ≈ 1min, 54sec training time.

The lower panel shows the minimum test accuracy to be in classifying 5s while the maximum is for 1. The final test accuracy is seen to be 88.7% for denseNet4_300 with a modestly high training time of about 2min, 22sec.

The CNN model applied to the MNIST problem Keras's convKeras4_20. The upper plot of Fig.10 shows a similiar initial sharp initial increase in accuracy as for the DNN (the scale is "stretched" due to fewer epochs), with validation lagging below the training accuracy before catching up to yield a vlaue of $\text{acc}_{\text{val}} = 0.956$ after 10 epochs. From the lower plot, we see good scores across the classes and in particular for 0, which was classified with 99% accuracy. The total test for convKeras4_20 was found to be 95.5% with a training time of about 1min, 54sec.
The CNN therefore seems more suitable for classifying the MNIST(28×28) data set.

C. Street View Housing Numbers (SVHN)

As seen in Fig.(11), this data set is a lot more challenging. Digits are presented with varying backgrounds and greater variance in skew and rotation. Additionally, the images may contain other digits than the one to be classified i.e. the centered one.

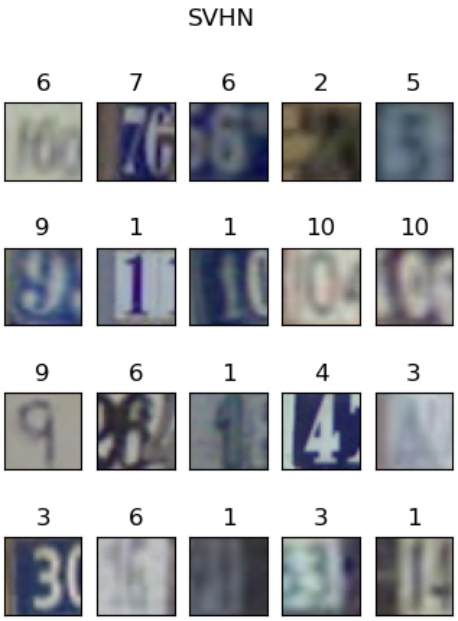

FIG. 11. Random selection of 20 samples from the Street View Housing Numbers (SVHN)[1] training set with corresponding labels. Resolution: 32x32x3.

Applying denseNet4_300 from MNIST required training and testing on subsets/partitions of the data due to memory constraints. The results were poor, with validation accuracy leveling out at about 0.2. We explored a variety of DNNs through keras, thereby granting us more freedom in use of activation functions and increased speed, but none proved much better.
Through keras, we found two convolutional networks with promising results. The first is a modification of LeNet-5, proposed by LeCunn et al. in 1998 for classification of handwritten zip code recognition.

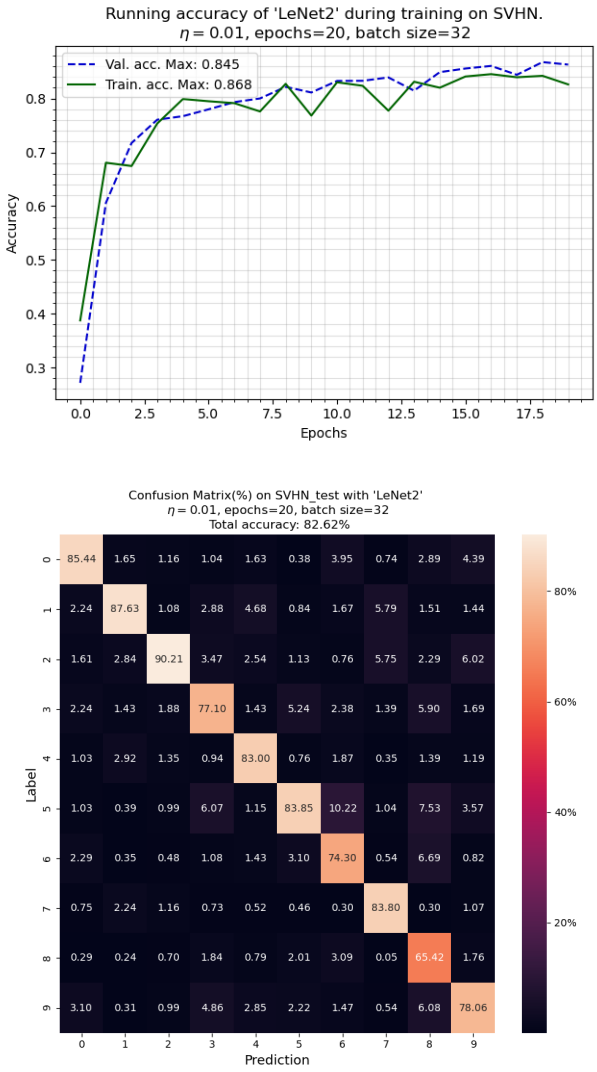

FIG. 12. Performance of leNet2 on SVHN. It consists of 6 convolutional layers with (32,32,64,120) filters with kernel size (3×3). A max pool layer follows each. This is again followed by a dense layer of 128 nodes. All layers use the tanh activation function. Final test accuracy = 82.6% with 351.5s ≈ 5min, 16sec training time.

leNet2 consists of 4 convolutional layers with $(32, 32, 64, 120)$ 3×3 filters, each followed by a max pool layer. This is followed by a single fully connected layer with 128 nodes. Notably, we use the tanh function, as was done in LeNet-5.
Fig.(12) shows how the Keras implemented leNet2 performed on SVHN. As seen in the upper plot, there is little sign of overfitting. When a model is overfitting to the training data, it will start scoring higher on the training data while validation accuracy remains low or decreases. This is clearly not the case, indicating that we could benefit from a more complex model. This is not surprising, as this dataset is more complex than what LeNet-5 was tailored for.

From the confusion matrix in the lower plot of Fig.(12), we see the best performance in prediction 2s and the worst in 8s. This is likely due to the similarity in shape between 8,6 and 3. We also see a final test accuracy is 82.6% and training time of 5min, 16sec for leNet2.

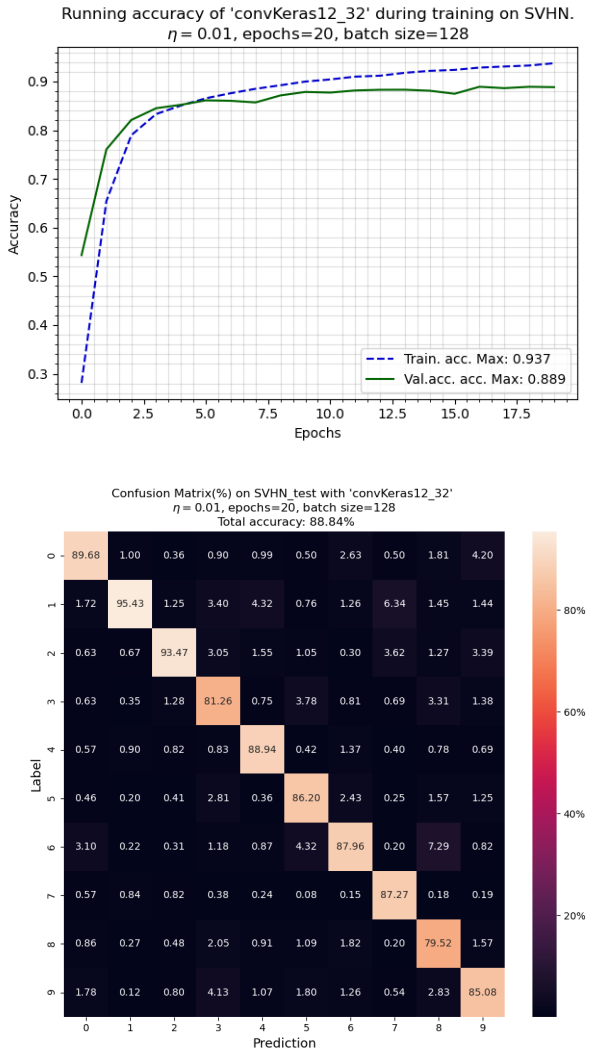

FIG. 13. Performance of convKeras12_32 on SVHN. This model consists of 6 convolutional layers, each with a kernel size of (3x3). The first two convolutional layers contain 32 filters (each), followed by two layers with 64 filters and lastly two layers of 128 filters. A max pool layer is applied after each convolutional layer. For this model, we used the ReLU function. As seen, we got a test accuracy of 88.84% for a training time of 644.4s ≈ 10min, 44sec

We managed to improve on the results of leNet2 by including two additional convolutional layers. The convKeras12_32 model thus consists of 12 layers before the dense network when including max pooling. As seen in the upper plot of Fig.(13), the accuracy plot is a lot smoother than for leNet2 which might be due to employing the ReLU function whose usefulness had not yet been discovered when leNet was introduced. Additionally, we see hints of overfitting since the training accuracy keeps increasing with minimal increase in validation accuracy. The model might therefore benefit from regularization using the l1/l2 norm or drop-out.
As seen in the lower plot, this model performed best in classifying 1's, and worst on 8's, which has been typical. The final test accuracy was 88.8% with a training time of about 10min, 44sec. A summary of the performance of all the models is shown in Tab.I.

TABLE I. Test accuracy and training time for our selected models

Data set	Model	acc_{test}	$t_{\text{train}}(s)$	$t_{\text{train}}(m,s)$
MNIST(8x8)	denseNet2_20	0.942	6.8s	6.8s
	convNet1_3	0.889	72.2s	1m,12s
MNIST	denseNet4_300	0.887	142.0s	2m,22s
	convKeras4_20	0.955	113.9s	1m,54s
SVHN	leNet2	0.826	351.5s	5m,16s
	convKeras12_32	0.888	644.4s	10m,44s

V. Discussion

We created our own cross-correlate and convolution function, but they proved ≈ 130 times slower than that of scipy's, which we ended up using, since it makes up a conceptually small part of the code.
From our theory section one might expect the max pooling layer to reduce computational cost, but our implementation of it drastically increase feedforward and backpropagation times to the point where it could not included. Unlike the convolutional layer, we are not aided by scipy and so the python for-loops take their toll. Improvements to this can likely be made by use of e.g. Numba's JIT.
Further improvements would include a more thorough hyperparameter search, in particular of model architecture, learning rate and initialization schemes. The use of more modern techniques such as ADAM for optimization, drop-out for regularization, adaptive learning rate and batch normalization would likely also improve our results.
Due to time constraints, we ended up comparing the models we managed to find. This resulted in us comparing a Keras model to our own for the MNIST(8×8) data, which is not ideal, since some of the difference is likely due to the superior implementation of Keras.

VI. Conclusion

We applied an assortment of convolutional and dense neural networks from both Keras and our own code to classify three datasets, all consisting of digits ranging

from 0-9. These datasets placed successively higher demands for efficient feature extraction from the networks. We found that for MNIST(8×8), the dense network provided high predictive power at low computational cost with a test accuracy of 94.2% and 6.8s training time. This outperformed the best CNN model we found, which gave a test accuracy of 88.9% with 1min, 12sec training time.
For the MNIST(8×8) data set, we saw the performance of the dense network starting to fall off compared to the CNN. The accuracy scores were 88.7% and 95.5%, with training times of 2min, 22sec and 1min, 54sec respectively.
The large number of features and increased geometric distortions of the $32 \times 32 \times 3$ Street View House Number images resulted in our search providing no dense models with final test accuracy above 30%. Two convolutional models gave promising results on this data set. A modified version of LeNet-5 gave an accuracy of 82.6% and training time of about 5min, 16sec and the model convKeras12_32 containing 6 convolutional layers achieved an accuracy of 88.8% with a training time of 10m, 44sec.
These results indicate that the general purpose DNN is better suited for very low resolution images where they can utilize the dense connectivity without being computationally prohibitive, whereas the specialized architecture of the CNN becomes necessary for more demanding data sets due to the high computational cost they incur during training.

References

[1] Yuval Netzer et al. *Reading Digits in Natural Images with Unsupervised Feature Learning*. 2011. URL: http://ufldl.stanford.edu/housenumbers.

[2] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[3] H. Fossheim. *Project 2: Classification and Regression, from linear and logistic regression to neural networks*. Nov. 2022. URL: https://github.com/fosheimdet/FYS-STK4155/tree/main/Project2.

[4] M. Hjorth-Jensen. *Applied Data Analysis and Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html.

[5] D. H. Hubel and T. N. Wiesel. "Receptive fields of single neurones in the cat's striate cortex". In: *The Journal of Physiology* 148.3 (1959), pp. 574–591. DOI: https://doi.org/10.1113/jphysiol.1959.sp006308. eprint: https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.1959.sp006308. URL: https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1959.sp006308.

[6] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541. eprint: https://direct.mit.edu/neco/article-pdf/1/4/541/811941/neco.1989.1.4.541.pdf. URL: https://doi.org/10.1162/neco.1989.1.4.541.

[7] Y. LeCun et al. *Gradient-based learning applied to document recognition*. Nov. 1998. DOI: 86(11):2278-2324. URL: http://yann.lecun.com/exdb/mnist/.

Appendices

A. Deriving the expression for $\delta^l_{i,j}$

$$\frac{\partial C}{\partial \theta^l} = \sum_{i,j \in Z^l} \frac{\partial z^l_{i,j}}{\partial \theta^l} \overbrace{\frac{\partial a^l_{i,j}}{\partial z^l_{i,j}} \underbrace{\sum_{m,n \in Z^{l+1}} \underbrace{\frac{\partial z^{l+1}_{m,n}}{\partial a^l_{i,j}}}_{\textbf{(i)}} \underbrace{\frac{\partial C}{\partial z^{l+1}_{m,n}}}_{\delta^l_{m,n}}}_{\textbf{(I)}}}^{\delta^l_{i,j}} \tag{52}$$

$$z^{l+1}_{m,n} = \sum_{u,v=0}^{w-1} \omega^{l+1}_{u,v} \tilde{a}^l_{m+u,n+v} + b_l \tag{53}$$

$$\textbf{(i)} = \frac{\partial z^{l+1}_{m,n}}{\partial a^l_{i,j}} = \frac{\partial}{\partial a^l_{i,j}} \sum_{u,v=0}^{w-1} \omega^{l+1}_{u,v} \tilde{a}^l_{m+u,n+v} + b_l \tag{54}$$

Due to padding, the valid region of $\tilde{A}^l$ is indexed differently than A^l. Indices start at (0,0) in the upper left corner of both maps, and so the first valid node of in $\tilde{A}^l$ is $\tilde{a}^l_{p,p}$. Letting negative values of (i,j) represent zero-padding, we then have

$$\tilde{a}^l_{\tilde{i},\tilde{j}} = a^l_{\tilde{i}-p,\tilde{j}-p} = a^l_{m+u-p,n+v-p} \tag{55}$$

Where we inserted for $\tilde{i} = m+u,\ \tilde{j} = n+v$.

$$\textbf{(i)} = \frac{\partial}{\partial a^l_{i,j}} \sum_{u,v=0}^{w-1} \omega^{l+1}_{u,v} a^l_{m+u-p,n+v-p} \tag{56}$$

Furthermore, **(i)** will only be non-zero when (i,j) is in the receptive field of (m,n).
This is the case when

$$m - p \leq i \leq m - p + (w-1) \tag{57}$$

$$\Rightarrow \underbrace{i + p - (w-1)}_{m_{\min}} \leq m \leq \underbrace{i+p}_{m_{\max}} \tag{58}$$

And likewise for j. Going forth, we omit the second index until the final expression, since the calculations are equivalent. The condition (57) can be enforced by multiplying the reduced version of **(i)** by $\{H[m = m_{\min}] - H[m = m_{\max}]\}$ where H is the Heaviside step function:

$$\textbf{(i)}_r = \frac{\partial}{\partial a^l_i} \sum_{u=0}^{w-1} \omega^{l+1}_u a^l_{m+u-p} \left\{H[m_{\min}] - H[m_{\max}]\right\} \tag{59}$$

Only the term where $m+u-p = i \rightarrow u = i-m+p$ will be non-zero, giving

$$\underline{\textbf{(i)}_r = \omega^{l+1}_{i-m+p} \left\{H[m_{\min}] - H[m_{\max}]\right\}} \tag{60}$$

We can now insert this into **(I)**:

$$\textbf{(I)}_r = \sum_{m=0}^{H^{l+1}-1} \omega^{l+1}_{i-m+p} \delta^{l+1}_m \left\{H[m_{\min}] - H[_{\max}]\right\} \tag{61}$$

$$= \sum_{m=i-[(w-1)-p]}^{i+p} \omega^{l+1}_{i-m+p} \delta^{l+1}_m \tag{62}$$

Where we allowed for m to take on negative values to remove the lower Heaviside function. We therefore let $\delta^{l+1}_m = 0$ for $m < 0$.
Changing variable to $u' = m - i + [(w-1)-p]$ yields

$$\textbf{(I)}_r = \sum_{u'=0}^{w-1} \omega^{l+1}_{(w-1)-u'} \delta^{l+1}_{\{i-[(w-1)-p]\}+u'} = \sum_{u'=0}^{w-1} \omega^{l+1}_{(w-1)-u'} \delta^{l+1}_a \tag{63}$$

Where we defined the index $a = i - [(w-1)-p] + u'$. It has a minimum value of

$$a_{\min} = i_{\min} - [(w-1)-p] = \underline{m_{\min} - [(w-1)-p]} \tag{64}$$

And maximum value of

$$a_{\max} = (i_{\max}) - p = (m_{\max} + (w-1)) - p \tag{65}$$

$$= \underline{m_{\max} + [(w-1)-p]} \tag{66}$$

δ^{l+1} therefore needs to be zero-padded on all sides with a padding of size $p_\delta = (w-1) - p$, where we recall that p was the size of the padding used on A^l before cross-correlation. As is done when padding A^l, we re-map the indices so that they are all positive and start at (0,0) in the upper left corner. The index values corresponding to the same element of both matrices will then be shifted by an amount equal to the padding:

$$\delta^{l+1}_a = \tilde{\delta}^{l+1}_{a+[(w-1)-p]} \tag{67}$$

Expanding out a on the RHS yields

$$\delta^{l+1}_a = \tilde{\delta}^{l+1}_{i+u'} \tag{68}$$

Which can finally be inserted into Eq.(63) to give

$$\textbf{(I)}_r = \sum_{u'=0}^{w-1} \omega_{(w-1)-u'} \tilde{\delta}^{l+1}_{i+u'} \tag{69}$$

The calculations for the second index are equivalent, and so the full expression is

$$\mathbf{(I)} = \sum_{u',v'=0}^{w-1} \omega^{l+1}_{(w-1)-u',(w-1)-v'} \tilde{\delta}^{l+1}_{i+u,j+v'} = \left(K^{l+1} * \tilde{\delta}^{l+1}\right)_{i,j} \tag{70}$$

Lastly, as seen in Eq.(52), the error is given by

$$\delta^l_{i,j} = \frac{\partial a^l_{i,j}}{\partial z^l_{i,j}}(\mathbf{I}) \tag{71}$$

$$\Rightarrow \underline{\delta^l_{i,j} = f'(z^l_{i,j}) \left(K^{l+1} * \tilde{\delta}^{l+1}\right)_{i,j}} \tag{72}$$

B. Padding for backpropagation, heuristic version

In cross-correlating a matrix A^l of shape (H^l, W^l) with a kernel K^{l+1} of shape (K_h, K_w), using a padding of p^l and a stride of S, the resulting matrix will have the following shape

$$H^{l+1} = 1 + (H^l - K_h + 2p^l)/S^{l+1} \tag{73}$$
$$W^{l+1} = 1 + (W^l - K_w + 2p^l)/S^{l+1} \tag{74}$$

We want to pad our $\boldsymbol{\delta}^{l+1}$ such that

$$\left[\tilde{\boldsymbol{\delta}}^{l+1} * K^{l+1}\right] = \left[A^l\right] = (H^l, W^l) \tag{75}$$

$$W^l = W^{l+1} - 2r + 2p^{l+1} \tag{76}$$
$$W^l = (W^l - 2r + 2p^l) - 2r + 2p^{l+1} \tag{77}$$
$$4r = 2(p^l + p^{l+1}) \tag{78}$$
$$\Rightarrow \underline{p^{l+1} = 2r - p^l} \tag{79}$$

Or:

$$W^l = W^{l+1} - 2r + 2p^{l+1} \tag{80}$$
$$\Rightarrow \underline{p^{l+1} = (W^l - W^{l+1} + 2r)/2} \tag{81}$$

Each layer instance therefore need the shape of the previous layer in order to perform backpropagation. This is provided during from the initialization procedure.

A. Padding in Max Pooling

Assume we are using a filter of size $\omega \times \omega$ to down-sample the feature map(s) produced by a convolutional layer. This is done by iterating a filter across the feature maps and storing only the maximum value contained in the overlap between the feature map and the filter.
The stride (S) of the pooling process determines the step size, i.e. the distance the filter moves from one iteration to the next.
Depending on the dimensions of the feature map, the stride as well as the filter size, the filter may not fit completely within the valid region of the feature map. One can either discard this last step, thereby discarding a portion of the feature map ("valid" mode) or add appropriate zero-padding to allow all values to be included ("same" mode).
The math is the same for both dimensions, so we will only calculate appropriate the relevant expression for the horizontal case.

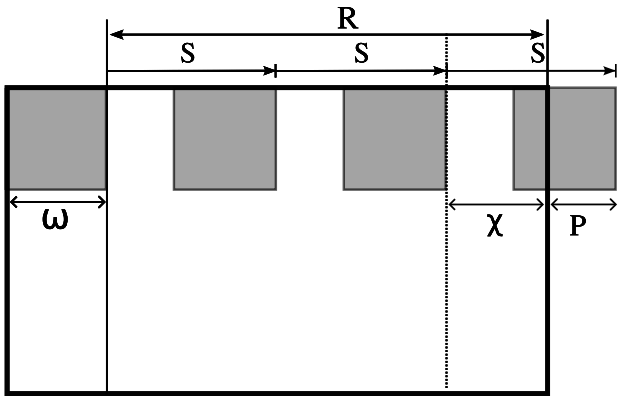

FIG. 14. Example of the pooling in the horizontal direction when the width (W) is greater than the filter size (ω).

Assuming $W > \omega$, we find it handy to introduce

$$R = W - \omega \tag{82}$$

Since if this quantity is divisible by the stride S, no padding will be needed as the right edge of the filter lines up with the border of the feature map. The number of times the stride fits in R is given by

$$\frac{R}{S} = \frac{n_s S + \chi}{S} = \left\lfloor \frac{R}{S} \right\rfloor + \frac{\chi}{S} \tag{83}$$

Where $n_s = \left\lfloor \frac{R}{S} \right\rfloor$ is the number of times the stride fits fully in R and χ is the remainder. We thus have

$$\underline{\chi = R - \left\lfloor \frac{R}{S} \right\rfloor} \tag{84}$$

"same" pooling If we choose "same" pooling, we see from Fig.14 that we need to add the following padding:

$$\boxed{P = \left\lceil \frac{\chi}{S} \right\rceil S - \chi} \tag{85}$$

The reason we don't set $P = S - \chi$ is that this results in $P_{\text{whole}} = S$ when the stride fits a whole number of times in R, which we do not want. The ceiling function remedies this: $P_{\text{whole}} = \left\lceil \frac{0}{S} \right\rceil S - 0 = 0$.
There are however two caveats to this equation which need to be accounted for:

- Firstly, if the calculated padding is greater than or equal to the filter size ($P \geq \omega$), the last step will place the filter completely outside of the valid region of the feature map[7]. We therefore add the following demand

```
if(P>=w):
    P=0
```

- Secondly, if the size of the filter exceeds the size of the feature map in the dimension under consideration ($\omega > W$), then the equation does not hold (though modifications to include for this may be possible). In this case the padding will just be given by how much the filter exceeds the height/width of the feature map, i.e. $P = W - \omega$:

```
if(w>=W):
    P=W-w
```

"valid" mode From Fig.14 we see that we simply need to add subtract χ from W in the calculation of the width of the pooled layer

$$W_{\text{pooled}} = 1 + \frac{(W - \omega - \chi)}{S} \tag{86}$$

- If $\omega > W$, we throw an error since it is impossible to perform the pooling on valid pixels only.

[7] Despite the stride starting from inside of the feature map.