Introduction

...

Table of contents

1 Convolutional neural networks and their components

Let us start this paper with a **brief** recall of what a neural network *is*. A neural network is, intuitivly, a 'network' of 'neurons' which are connected together in a layerd structure. Each layer contains several neurons that are each connected to all the neurons of the previous layer. The neurons are supposed to 'look' for certain 'features' in its input data. If the neuron 'thinks' that these features are present it 'fires'. If it does not think that the input data has the features it is looking for, the neuron does not fire. In addition to layers of neurons, nerual networks can also contain special 'non-thinking' layers. Layers in which there are no neurons.

A neural network's job is to process some input data and produce a final 'ativation'. This final activation represents what the nerural network 'thinks' about the input it has processed. A very common application of nerual networks sees them used to classify large data sets. In this context, the output of the network is often interpreted as a probability which represents how confident the network is that one piece of data belongs to a particular class. The convolutional neural network that we are going to be using as an example in this paper, serves such a classification purpose.

In this section we are going to develop the equations which govern the operation of each layer in a convolutional neural network. In other words, the equations governing the operation of each component. We will start by looking at the forward operation of the network and then proccede with the backward operation.

Terminology and notation

Before we delve in to the components of a convolutional neural network we need to introduce some terms and notation. Scalars will be represented with a lowercase letter (e.g. x). Vectors will be represented with a bold lowercase letter (e.g. $\boldsymbol{x}$). Matricies will be represented with an uppercase letter (e.g. X). Tensors (three dimensional matricies) will be represented with a bold uppercase letter (e.g. $\boldsymbol{X}$).

Neruons in a nerual network consit, in princple, of only two parts: a set of weights and a bias. Weights will be represented with the letter 'w' while the bias will be represented with the letter 'b'. The preliminary output of a neruon (or layer of neurons), before it is passed through an activation, will be represented with the letter 'z'. The output after activation will be represented with the letter 'a'.

We need the ability to reference specific positions in the neural network and for that we need *a lot* of indexes. Since we want our mathematics to match our code to a certain extent, all the indices used in this paper will start at 0. The layer index which tells us which layer we are looking at will be written using a parenthetical superscript (e.g. $\boldsymbol{a}^{(l)}$ is the activation vector of layer l). Layers in a nerual network are divided up by activations. Everything after an activation up until the next activation is part of the next layer. Furthermore, the layers in a neural network all have one or more channels. The neurons within a layer are connected to all the channels in that layer. It is through these channels that input data is sent to a neuron. The channel index which tells us which channel we are looking at will be written using a parenthetical subscript (e.g. $a_{(c)}^{(l)}$ is the scalar activation of layer l in channel c). The output of each neuron in a layer becomes a channel in the next layer. To reference both the input and output channel, a double parenthetical subscript will be used (e.g. $w_{(c,c')}^{(l)}$ is the scalar weight in layer l connecting channel c' from layer l to channel c in layer $l+1$). To reference positions within a matrix we will use regular xy-coordinates (e.g. $a(x, y)$ is the scalar in row x at column y). Equations which relate a previous layer to the next one will generally use blue color for indicies belonging to the previous layer. Red color will be used for the indices beloning to the next layer.

We will often need to know the *dimensions* of layer. How many channels does it have and what are the sizes of the matrices in that layer. These dimensions will be written using the greek letter eta (η) with a layer index and a subscript denoting which dimension is in question. A 'c' subscript means it is the number of channels in the layer *minus one*. A 'x' and 'y' subscript means it is row and column length *minus one* respectivly. We use the letter η instead of n to remind the reader that all dimensions are substracted 1, because we want all our indices to start at zero. As an example, if layer 3 has eight individual channels then $\eta_c^{(3)} = 7$.

The inital input to a network can be though of as the first activation to the network. So we are going to denote the inital input using the appropriate 'a' letter with a zero in parenthetical subscript.

1.1 Forward operation

1.1.1 The activation function

All the 'thinking' layers of a neural network—those layers with neurons in them—make use of some activation function. A function that is non-linear which ultimately determines wheter a neuron fires or not. When designing a network one can chose between several different possible activation functions. In this paper we are only going to be looking at the rectified linear unit (ReLu) activation function. Since it is the one that is used in our working example. We are going to be using the ReLU function with both scalars, vectors, matrices and tensors. For the larger 'number structres' the ReLU function is applied to each entry individualy (i.e. each coordinate in a vector at a time). The ReLU function is defined as

$$\begin{aligned}
\mathrm{ReLU}(z) &= \max(0, z) \\
\mathrm{ReLU}(\boldsymbol{z}) &= \begin{pmatrix} \mathrm{ReLU}(z_0) \\ \mathrm{ReLU}(z_1) \\ \vdots \\ \mathrm{ReLU}(z_n) \end{pmatrix} \\
\mathrm{ReLU}(\mathrm{Z}) &= \begin{pmatrix} \mathrm{ReLU}(z_{0,0}) & \mathrm{ReLU}(z_{0,1}) & \dots & \mathrm{ReLU}(z_{0,n}) \\ \mathrm{ReLU}(z_{1,0}) & \mathrm{ReLU}(z_{1,1}) & \dots & \mathrm{ReLU}(z_{1,n}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{ReLU}(z_{m,0}) & \mathrm{ReLU}(z_{m,1}) & \dots & \mathrm{ReLU}(z_{m,n}) \end{pmatrix} \\
\mathrm{ReLU}(\boldsymbol{Z}) &= \dots
\end{aligned}$$

The index notation used here is only for illustration purposes with no semantic meaning.

1.1.2 Fully-connected layer

The fully-connected layer of a convolutional neural network is exactly the same as a regular layer of neurons from a classical multi-layer percepetron network. The layer takes a vector of scalar values as its input and returns a vector of scalar values as its output. Each neuron in a fully-connected layer contains a vector of weights and a bias. For the remainder of this section, let layer number l denote a fully-connected layer.

The equation governing the forward operation for this layer at the level of the individual neuron is

$$\begin{aligned} z_{(c)}^{(l)} &= \sum_{c'=0}^{\eta_c^{(l-1)}} \left(w_{(c,c')}^{(l)} \, a_{(c')}^{(l-1)} \right) + b_{(c)}^{(l)} \\ a_{(c)}^{(l)} &= \mathrm{ReLU}\left(z_{(c)}^{(l)} \right) \end{aligned} \tag{1}$$

We can scale this equation up to the layer level. But first we need the following definitions.

$$\boldsymbol{z}^{(l)} = \begin{pmatrix} z_{(1)}^{(l)} \\ z_{(2)}^{(l)} \\ \vdots \\ z_{(\eta_c^{(l+1)})}^{(l)} \end{pmatrix}$$

$$W^{(l)} = \begin{pmatrix} w_{(0,0)}^{(l)} & w_{(0,1)}^{(l)} & \cdots & w_{(0,\eta_c^{(l)})}^{(l)} \\ w_{(1,0)}^{(l)} & w_{(1,1)}^{(l)} & \cdots & w_{(1,\eta_c^{(l)})}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{(\eta_c^{(l+1)},0)}^{(l)} & w_{(\eta_c^{(l+1)},1)}^{(l)} & \cdots & w_{(\eta_c^{(l+1)},\eta_c^{(l)})}^{(l)} \end{pmatrix}$$

$$\boldsymbol{a}^{(l-1)} = \begin{pmatrix} a_{(1)}^{(l-1)} \\ a_{(2)}^{(l-1)} \\ \vdots \\ a_{(\eta_c^{(l)})}^{(l-1)} \end{pmatrix}$$

$$\boldsymbol{b}^{(l)} = \begin{pmatrix} b_{(c)}^{(l)} \\ b_{(c)}^{(l)} \\ \vdots \\ b_{(c)}^{(l)} \end{pmatrix}$$

Using these definitions we can write equation (1) at the layer level using matrix multiplication.

$$\begin{aligned} \boldsymbol{z}^{(l)} &= W^{(l)} \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)} \\ \boldsymbol{a}^{(l)} &= \mathrm{ReLU}\left(\boldsymbol{z}^{(l)} \right) \end{aligned} \tag{2}$$

1.1.3 Convolutional layer

Descriptive text ...

The equation governing the forward operation for this layer at the level of the individual neuron is

$$\begin{aligned} z_{(c)}^{(l)}(x,y) &= \sum_{c'}^{\eta_c^{(l-1)}} \left(\sum_{y'}^{\eta_y^{(l-1)}} \sum_{x'}^{\eta_x^{(l-1)}} \left(w_{(c,c')}^{(l)}(x',y') \, a_{(c)}^{(l-1)}(x+x', x+y') \right) \right) + b_{(c)}^{(l)} \\ a_{(c)}^{(l)}(x,y) &= \mathrm{ReLU}\left(z_{(c)}^{(l)}(x,y) \right) \end{aligned} \tag{3}$$

This equation is rather complex and ‘scaling’ it up to the layer level is not trivial at all. First we have to introduce the intermediary Hadamard product variable h.

$$
\begin{aligned}
h^{(l)}_{(c,c')}(x,y) &= \sum_{y'}^{\eta_y^{(l-1)}} \sum_{x'}^{\eta_x^{(l-1)}} \left(w^{(l)}_{(c,c')}(x',y')\, a^{(l-1)}_{(c)}(x+x', x+y') \right) \\
h^{(l)}_{(c)}(x,y) &= \sum_{c'}^{\eta_c^{(l-1)}} h^{(l)}_{(c,c')}(x,y)
\end{aligned}
$$

We call it the Hadamard product because the mathematical operation that correspond to the sum over y' and x' is the Hadamard product. From here we can scale up the variables in a straight forward way.

$$
Z^{(l)}_{(c)} = \begin{pmatrix}
z^{(l)}_{c}(0,0) & z^{(l)}_{c}(0,1) & \dots & z^{(l)}_{c}(0,\eta_y^{(l)}) \\
z^{(l)}_{c}(1,0) & z^{(l)}_{c}(1,1) & \dots & z^{(l)}_{c}(1,\eta_y^{(l)}) \\
\vdots & \vdots & \ddots & \vdots \\
z^{(l)}_{c}(\eta_x^{(l)},0) & z^{(l)}_{c}(\eta_x^{(l)},1) & \dots & z^{(l)}_{c}(\eta_x^{(l)},\eta_y^{(l)})
\end{pmatrix}
$$

$$
H^{(l)}_{(c)} = \begin{pmatrix}
h^{(l)}_{c}(0,0) & h^{(l)}_{c}(0,1) & \dots & h^{(l)}_{c}(0,\eta_y^{(l)}) \\
h^{(l)}_{c}(1,0) & h^{(l)}_{c}(1,1) & \dots & h^{(l)}_{c}(1,\eta_y^{(l)}) \\
\vdots & \vdots & \ddots & \vdots \\
h^{(l)}_{c}(\eta_x^{(l)},0) & h^{(l)}_{c}(\eta_x^{(l)},1) & \dots & h^{(l)}_{c}(\eta_x^{(l)},\eta_y^{(l)})
\end{pmatrix}
$$

$$
B^{(l)}_{(c)} = \begin{pmatrix}
b^{(l)}_{c} & b^{(l)}_{c} & \dots & b^{(l)}_{c} \\
b^{(l)}_{c} & b^{(l)}_{c} & \dots & b^{(l)}_{c} \\
\vdots & \vdots & \ddots & \vdots \\
b^{(l)}_{c} & b^{(l)}_{c} & \dots & b^{(l)}_{c}
\end{pmatrix}
$$

Before we can write down the equation which governs the operation at the layer level, we need to remove one more index.

$$
\begin{aligned}
\boldsymbol{Z}^{(l)} &= \begin{pmatrix} Z^{(l)}_{(0)} & Z^{(l)}_{(1)} & \dots & Z^{(l)}_{(\eta_c^{(l)})} \end{pmatrix} \\
\boldsymbol{H}^{(l)} &= \begin{pmatrix} H^{(l)}_{(0)} & H^{(l)}_{(1)} & \dots & H^{(l)}_{(\eta_c^{(l)})} \end{pmatrix} \\
\boldsymbol{B}^{(l)} &= \begin{pmatrix} B^{(l)}_{(0)} & B^{(l)}_{(1)} & \dots & B^{(l)}_{(\eta_c^{(l)})} \end{pmatrix}
\end{aligned}
$$

We can now write down the equation which governs the forward operation at the layer level.

$$
\begin{aligned}
\boldsymbol{Z}^{(l)} &= \boldsymbol{H}^{(l)} + \boldsymbol{B}^{(l)} \\
\boldsymbol{A}^{(l)} &= \mathrm{ReLU}\left(\boldsymbol{Z}^{(l)}\right)
\end{aligned} \tag{4}
$$

1.1.4 Pooling layer

Descriptive text ...

Equations which give the dimensional relations.

The equation which governs the forward operation of the layer is

$$\begin{aligned} a_{(c)}^{(l)}(x, y) &= \operatorname{maxpool}\left(a_{(c)}^{(l-1)}(x, y)\right) \\ &= \max_{\substack{r_x \in [0 \mathinner{..} \delta_x] \\ r_y \in [0 \mathinner{..} \delta_y]}} \left(a_{(c)}^{(l-1)}(x s_x + r_x, y s_y + r_y)\right) \end{aligned} \tag{5}$$

Going to the layer level is straightforward.

$$A_{(c)}^{(l)} = \begin{pmatrix} a_{(c)}^{(l)}(0, 0) & a_{(c)}^{(l)}(0, 1) & \dots & a_{(c)}^{(l)}(0, \eta_y^{(l)}) \\ a_{(c)}^{(l)}(1, 0) & a_{(c)}^{(l)}(1, 1) & \dots & a_{(c)}^{(l)}(1, \eta_y^{(l)}) \\ \vdots & \vdots & \ddots & \vdots \\ a_{(c)}^{(l)}(\eta_x^{(l)}, 0) & a_{(c)}^{(l)}(\eta_x^{(l)}, 1) & \dots & a_{(c)}^{(l)}(\eta_x^{(l)}, \eta_y^{(l)}) \end{pmatrix}$$

Which yields the rather boring layer equation

$$\boldsymbol{A}^{(l)} = \begin{pmatrix} A_{(0)}^{(l)} & A_{(1)}^{(l)} & \dots & A_{(\eta_c^{(l)})}^{(l)} \end{pmatrix}$$

$$\boldsymbol{A}^{(l)} = \operatorname{maxpool}\left(\boldsymbol{A}^{(l-1)}\right) \tag{6}$$

1.1.5 Output layer

Descriptive text ...

Note that the last layer before the output layer does not need an activation function.

The equations are

$$\tilde{y}_{(c)} = \frac{\exp\left(a_{(c)}^{(l)}\right)}{\sum_{c^*=0}^{\eta_c^{(l)}} \exp\left(a_{(c^*)}^{(l)}\right)} \tag{7}$$

$$\ell = -\sum_{c=0}^{\eta_c^{(l)}} y_{(c)} \log\left(\tilde{y}_{(c)}\right) \tag{8}$$

1.2 Backward propagation

1.2.1 Output layer

Descriptive text ...

$$\frac{\partial L}{\partial \tilde{y}_{(c)}} = \frac{\partial}{\partial \tilde{y}_{(c)}} \left(-\sum_{c^*=0}^{\eta_c^{(l)}} y_{(c^*)} \log\left(\tilde{y}_{(c^*)}\right)\right) = -\frac{y_{(c)}}{\tilde{y}_{(c)}}$$

$$\begin{aligned} \frac{\partial \tilde{y}_{(c)}}{\partial a_{(c)}^{(l)}} &= \frac{\partial}{\partial a_{(c)}^{(l)}} \left(\frac{\exp\left(a_{(c)}^{(l)}\right)}{\sum_{c^*=0}^{\eta_c^{(l)}} \exp\left(a_{(c^*)}^{(l)}\right)}\right) \\ &= \frac{\exp\left(a_{(c)}^{(l)}\right)}{\sum_{c^*=0}^{\eta_c^{(l)}} \exp\left(a_{(c^*)}^{(l)}\right)} - \left(\frac{\exp\left(a_{(c)}^{(l)}\right)}{\sum_{c^*=0}^{\eta_c^{(l)}} \exp\left(a_{(c^*)}^{(l)}\right)}\right)^2 \\ &= \tilde{y}_{(c)} - \left(\tilde{y}_{(c)}\right)^2 \\ &= \tilde{y}_{(c)}(1 - \tilde{y}_{(c)}) \end{aligned}$$

$$\frac{\partial L}{\partial a^{(l)}_{(c)}} = \frac{\partial L}{\partial \tilde{y}_{(c)}} \frac{\partial \tilde{y}_{(c)}}{\partial a^{(l)}_{(c)}} = -\frac{y_{(c)}}{\tilde{y}_{(c)}} \tilde{y}_{(c)}(1-\tilde{y}_{(c)}) = \tilde{y}_{(c)} - y_{(c)} \tag{9}$$

1.2.2 Fully-connected layer

Descriptive text ...

$$\frac{\partial z^{(l)}_{(c)}}{\partial a^{(l-1)}_{(c^*)}} = \frac{\partial}{\partial a^{(l-1)}_{(c^*)}} \left(\sum_{c'=0}^{\eta_c^{(l-1)}} \left(w^{(l)}_{(c,c')} a^{(l-1)}_{(c')} \right) + b^{(l)}_{(c)} \right) = w^{(l)}_{(c,c^*)}$$

1.2.3 Pooling

Descriptive text ...

1.2.4 Convolutional layer

Descriptive text ...

1.3 Fully-connected layer

Admit that the choice of index is a bit odd but that it will make sense later. Regular formula.

$$Z_c^{(l+1)} = \sum_{c'=0}^{\eta_c^{(l)}} \left(W_{c'}^{(l,c)} Z_{c'}^{(l)} \right) + b^{(l,c)} \tag{10}$$

Partial derivative with respect to a $Z^{(l)}$.

$$\begin{aligned} \frac{\partial Z_c^{(l+1)}}{\partial Z_{c^*}^{(l)}} &= \frac{\partial}{\partial Z_{c^*}^{(l)}} \left(\sum_{c'=0}^{\eta_c^{(l)}} \left(W_{c'}^{(l,c)} Z_{c'}^{(l)} \right) + b^{(l,c)} \right) \\ &= \sum_{c'=0}^{\eta_c^{(l)}} \left(\frac{\partial}{\partial Z_{c^*}^{(l)}} \left(W_{c'}^{(l,c)} Z_{c'}^{(l)} \right) \right) + 0 \\ &= W_{c^*}^{(l,c)} \end{aligned} \tag{11}$$

Partial derivative with respect to a $W^{(l,c)}$.

$$\begin{aligned} \frac{\partial Z_c^{(l+1)}}{\partial W_{c^*}^{(l,c)}} &= \frac{\partial}{\partial W_{c^*}^{(l,c)}} \left(\sum_{c'=0}^{\eta_c^{(l)}} \left(W_{c'}^{(l,c)} Z_{c'}^{(l)} \right) + b^{(l,c)} \right) \\ &= \sum_{c'=0}^{\eta_c^{(l)}} \left(\frac{\partial}{\partial W_{c^*}^{(l,c)}} \left(W_{c'}^{(l,c)} Z_{c'}^{(l)} \right) \right) + 0 \\ &= Z_{c^*}^{(l)} \end{aligned} \tag{12}$$

Partial derivative with respect to the bias $b^{(l,c)}$.

$$\begin{aligned} \frac{\partial Z_{c}^{(l+1)}}{\partial b^{(l,c)}} &= \frac{\partial}{\partial b^{(l,c)}} \left(\sum_{c'=0}^{\eta_c^{(l)}} \left(W_{c'}^{(l,c)} Z_{c'}^{(l)} \right) + b^{(l,c)} \right) \\ &= \sum_{c'=0}^{\eta_c^{(l)}} \left(\frac{\partial}{\partial b^{(l,c)}} \left(W_{c'}^{(l,c)} Z_{c'}^{(l)} \right) \right) + 1 \\ &= 1 \end{aligned} \tag{13}$$

1.3.1 Layer level

Now let us move on up to the layer level. Z without a subscript represents a column vector of all the entries.

$$Z^{(l)} = \begin{pmatrix} Z_1^{(l)} \\ Z_2^{(l)} \\ \vdots \\ Z_{\eta_c^{(l)}}^{(l)} \end{pmatrix}$$

W without a neuron index and without a subscript represents a matrix. The neruon index selects row and the channel index selects column.

$$W^{(l)} = \begin{pmatrix} W_1^{(l,1)} & W_2^{(l,1)} & \dots & W_{\eta_c^{(l)}}^{(l,1)} \\ W_1^{(l,2)} & W_2^{(l,2)} & \dots & W_{\eta_c^{(l)}}^{(l,2)} \\ \vdots & \vdots & \ddots & \vdots \\ W_1^{(l,\eta_c^{(l+1)})} & W_2^{(l,\eta_c^{(l+1)})} & \dots & W_{\eta_c^{(l)}}^{(l,\eta_c^{(l+1)})} \end{pmatrix}$$

b without a neuron index is a column vector of biases.

$$b^{(l)} = \begin{pmatrix} b^{(l,1)} \\ b^{(l,2)} \\ \vdots \\ b^{(l,\eta_c^{(l)})} \end{pmatrix}$$

Regular formula.

$$Z^{(l+1)} = W^{(l)} \cdot Z^{(l)} + b^{(l)} \tag{14}$$

Gradient with respect to a $Z^{(l)}$. Hvordan skrive gradient med hensyn på noe?

$$\frac{\partial Z^{(l+1)}}{\partial Z^{(l)}} = W^{(l)} \tag{15}$$

Gradient with respect to a $W^{(l)}$.

$$\frac{\partial Z^{(l+1)}}{\partial W^{(l)}} = Z^{(l)} \tag{16}$$

Gradient with respect to a $b^{(l,c)}$.

$$\frac{\partial Z^{(l+1)}}{\partial b^{(l)}} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \tag{17}$$

1.4 Convolutional layer

Introduce the indexes. y selects row while x selects column. Regular formula.

$$Z^{(l+1)}_{c,y,x} = \sum_{c'=0}^{\eta_c^{(l)}} \sum_{y'=0}^{\eta_y^{(l)}} \sum_{x'=0}^{\eta_x^{(l)}} \left(W^{(l,c)}_{c',y',x'} Z^{(l)}_{c',y+y',x+x'} \right) + b^{(l,c)} \tag{18}$$

Partial derivative with respect to a $Z_{(l)}$. $c^* \in [c \mathinner{\ldotp\ldotp} c + \eta_c^{(l)}]$. $y^* \in [y \mathinner{\ldotp\ldotp} y + \eta_y^{(l)}]$. $x^* \in [x \mathinner{\ldotp\ldotp} x + \eta_x^{(l)}]$.

$$\begin{aligned} \frac{\partial Z^{(l+1)}_{c,y,x}}{\partial Z^{(l)}_{c^*,y^*,x^*}} &= \frac{\partial}{\partial Z^{(l)}_{c^*,y^*,x^*}} \left(\sum_{c'=0}^{\eta_c^{(l)}} \sum_{y'=0}^{\eta_y^{(l)}} \sum_{x'=0}^{\eta_x^{(l)}} \left(W^{(l,c)}_{c',y',x'} Z^{(l)}_{c',y+y',x+x'} \right) + b^{(l,c)} \right) \\ &= \sum_{c'=0}^{\eta_c^{(l)}} \sum_{y'=0}^{\eta_y^{(l)}} \sum_{x'=0}^{\eta_x^{(l)}} \left(\frac{\partial}{\partial Z^{(l)}_{c^*,y^*,x^*}} \left(W^{(l,c)}_{x',y',c'} Z^{(l)}_{x+x',y+y',c'} \right) \right) + 0 \\ &= W^{(l,c)}_{c^*,y',x'} \end{aligned} \tag{19}$$

Partial derivative with respecto to a $W^{(l,c)}$. $c^* \in [0 \mathinner{\ldotp\ldotp} \eta_c^{(l)}]$. $y^* \in [0 \mathinner{\ldotp\ldotp} \eta_y^{(l)}]$. $x^* \in [0 \mathinner{\ldotp\ldotp} \eta_x^{(l)}]$.

$$\begin{aligned} \frac{\partial Z^{(l+1)}_{c,y,x}}{\partial W^{(l,c)}_{c^*,y^*,x^*}} &= \frac{\partial}{\partial W^{(l,c)}_{c^*,y^*,x^*}} \left(\sum_{c'=0}^{\eta_c^{(l)}} \sum_{y'=0}^{\eta_y^{(l)}} \sum_{x'=0}^{\eta_x^{(l)}} \left(W^{(l,c)}_{c',y',x'} Z^{(l)}_{c,y+y',x+x'} \right) + b^{(l,c)} \right) \\ &= \sum_{c'=0}^{\eta_c^{(l)}} \sum_{y'=0}^{\eta_y^{(l)}} \sum_{x'=0}^{\eta_x^{(l)}} \left(\frac{\partial}{\partial W^{(l,c)}_{c^*,y^*,x^*}} \left(W^{(l,c)}_{c',y',x'} Z^{(l)}_{c,y+y',x+x'} \right) \right) + 0 \\ &= Z^{(l)}_{c^*,y+y^*,x+x^*} \end{aligned} \tag{20}$$

Partial derivative with respect to $b^{l,c}$.

$$\begin{aligned} \frac{\partial Z^{(l+1)}_{c,y,x}}{\partial b^{(l,c)}} &= \frac{\partial}{\partial b^{(l,c)}} \left(\sum_{c'=0}^{\eta_c^{(l)}} \sum_{y'=0}^{\eta_y^{(l)}} \sum_{x'=0}^{\eta_x^{(l)}} \left(W^{(l,c)}_{c',y',x'} Z^{(l)}_{c,y+y',x+x'} \right) + b^{(l,c)} \right) \\ &= 1 \end{aligned} \tag{21}$$

1.4.1 Layer level

Now let us move up to the layer level. To progress up to the layer level we only need to deal with the hadamard product. After that everything is exactly the same as for the fully-connected layer. $Z^{(l)}_c$ represents a two dimensional matrix.

$$Z^{(l)}_{c} = \begin{pmatrix} Z^{(l)}_{c,1,1} & Z^{(l)}_{c,1,2} & \cdots & Z^{(l)}_{c,1,\eta_x^{(l)}} \\ Z^{(l)}_{c,2,1} & Z^{(l)}_{c,2,2} & \cdots & Z^{(l)}_{c,2,\eta_x^{(l)}} \\ \vdots & \vdots & \ddots & \vdots \\ Z^{(l)}_{c,\eta_y^{(l)},1} & Z^{(l)}_{c,\eta_y^{(l)},2} & \cdots & Z^{(l)}_{c,\eta_y^{(l)},\eta_x^{(l)}} \end{pmatrix}$$

$W^{l,c}_{c'}$ also represents a two dimensional matrix.

$$W^{l,c}_{c'} = \begin{pmatrix} W^{l,c}_{c',1,1} & W^{l,c}_{c',1,2} & \cdots & W^{l,c}_{c',1,\eta_x^{(l)}} \\ W^{l,c}_{c',2,1} & W^{l,c}_{c',2,2} & \cdots & W^{l,c}_{c',2,\eta_x^{(l)}} \\ \vdots & \vdots & \ddots & \vdots \\ W^{l,c}_{c',\eta_y^{(l)},1} & W^{l,c}_{c',\eta_y^{(l)},2} & \cdots & W^{l,c}_{c',\eta_y^{(l)},\eta_x^{(l)}} \end{pmatrix}$$

I will denote the hadamard product by $\odot$. Here we introduce the intermediary variable $H^{(l,c)}_{c'}$. It is the hadamard product of $Z^{(l)}_{c}$ with $W^{l,c}_{c'}$.

$$H^{(l,c)}_{c'} = W^{(l,c)}_{c'} \odot Z^{(l)}_{c} = \sum_{y'=0}^{\eta_y^{(l)}} \sum_{x'=0}^{\eta_x^{(l)}} \left(W^{(l,c)}_{c',y',x'} Z^{(l)}_{c,y+y',x+x'} \right)$$

Substituting for it in Formula 18 we get an expression early similar to Formula 10.

$$Z^{(l+1)}_{c} = \sum_{c'=0}^{\eta_c^{(l)}} \left(H^{(l,c)}_{c'} \right) + b^{(l,c)} \tag{22}$$

The hadamard product is nothing more than a long finite sum of different weights and products. Taking the partial derivative of it–with respect ot any one of them—yields the cooresponding weight or product entry. Thus we have that.

$$\frac{\partial H^{(l,c)}_{c'}}{\partial Z^{(l)}_{c}} = W^{(l,c)}_{c'} \tag{23}$$

$$\frac{\partial H^{(l,c)}_{c'}}{\partial W^{(l,c)}_{c'}} = Z^{(l)}_{c} \tag{24}$$

If we scale up even further we get the formula which relates layer $l+1$ with layer l.

$$H^{(l)} = \begin{pmatrix} H^{(l,1)} \\ H^{(l,2)} \\ \vdots \\ H^{(l,\eta_c^{(l)})} \end{pmatrix} = \begin{pmatrix} \sum_{c'=0}^{\eta_c^{(l)}} \left(H^{(l,1)}_{c'} \right) \\ \sum_{c'=0}^{\eta_c^{(l)}} \left(H^{(l,2)}_{c'} \right) \\ \vdots \\ \sum_{c'=0}^{\eta_c^{(l)}} \left(H^{(l,\eta_c^{(l)})}_{c'} \right) \end{pmatrix}$$

Let us now calculate the partial derivatives.

$$\frac{\partial Z^{(l+1)}_{c}}{Z^{(l)}_{c'}} = \tag{25}$$

1.5 Pooling layer

A non thinking layer, good for adjusting the networks performance.

1.6 Final layer

$$w^{(l)}_{(c',c)}(y,x)$$

2 Neural network concepts

A nerual netwok can be used for solving a wide array of problems in computer science, arguably the most common of which lies in computer vision. The purpose of this section is to explain and demonstrate the motivation behind the structure of a typical neural netowork: the multi-layered perceptron (MLP). These neural networks can be applied to a wide range of problems, but for this section we are going to focus on computer vision.

The MLP performs well in image classification. The network can be considered a function with a number of inputs equal to the number of pixels in the image times three in the case of an RGB. The number of outputs is in this case the number of classes. This is obviously an extremely complicated function and finding the exact function is near impossible. A more intelligent approach is to approximate it with composition of several easily computed functions, such as linear functions. Figure 1 on page 12 presents a visualization of this concept.

The structured is loosely based on neuroscience; the nodes are often called neurons, which in reality is a number describing whether or not the neuron is active. The connections between neurons are called weights, denoted W, and translates how much the activation of one neuron should impact the activation of a following neuron. Lastly, we usually add constant values called

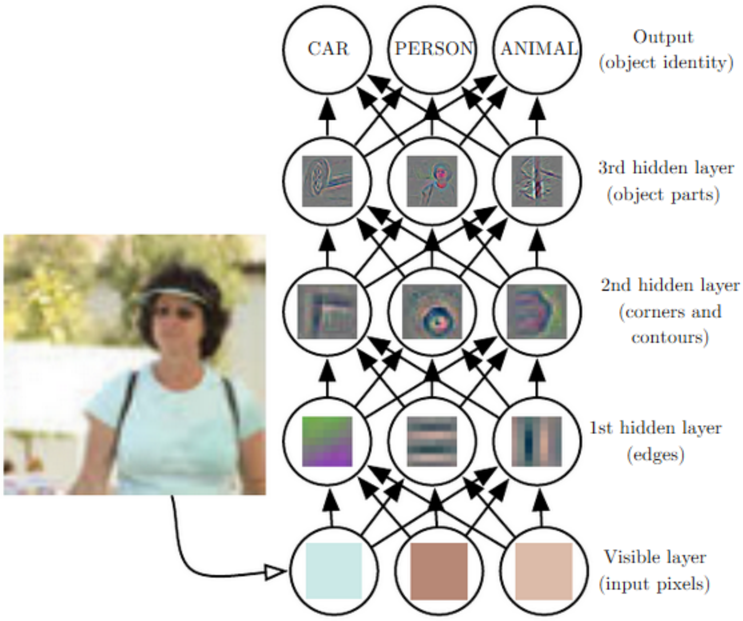

Figure 1 A visual

biases noted b. These values can be seen as thresholds in order for a neuron to activate. Parallel neurons are joined in layers denoted by $X^{(l)}$ where l is the layer index.

An MLP is a so-called fully connected neural network. This means that each neuron in layer $X^{(l)}$, impacts the activation of each neuron in the following layer $X^{(l+1)}$. With these notations, the passage from layer to the next can be written

$$X^{(l+1)} = WX^{(l)} + b \tag{26}$$

A simple network might consist of two such layers with Y number of neurons each. This means (Y) number of adjustable parameters! With the structure defined, we have yet to determine the parameters. This is where the training phase enters.

2.1 Training phase

The idea is simple. We start out with a large dataset with n classified examples, called a training set and we adjust our network to fit these data. The hope is then that the network will be able to classify new images with these same parameters. In order to quantify the performance of the MLP during the training phase, we introduce a cost function that, in short, is a non-negative function that is closer to zero the better the network performs. Knowing the desired output with the corresponding input, we look to minimize this cost function by adjusting the parameters of the MLP i.e. its weights and biases. This can be seen as an optimization problem with thousands of inputs (in our case, Y weights and biases). For demonstration purposes, we will explain the concept of gradient descent, an intuitive and common optimization algorithm.

2.2 Optimization using gradient descent

We have a function that we want to minimize without calculating all its possible values. The gradient of the function is an operation that tells us which inputs to adjust and by how much, in order to increase the function by as much as possible in a given point. Visually, it can be considered the direction in the input space that increases the output the most. In our case, we want to decrease the cost by as much as possible so we take the negative value of the gradient. As such, the gradient descent algorithm consists of, at each iteration, calculating the cost and its gradient, adjusting the inputs by “stepping in the direction of the gradient" (Figure 2) and repeating until we have found a local minimum or until the cost is sufficiently low. A limitation to this algorithm, is that there can be several local minima and there is no guarantee of finding the lowest one. In practice, though, the local minima in the context of neural networks are usually sufficiently low.

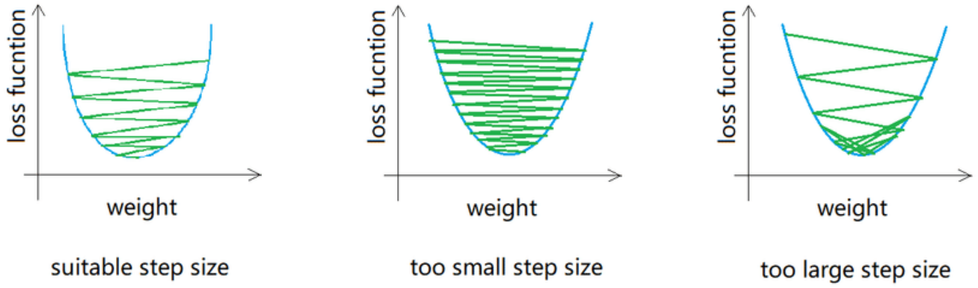

Figure 2 A visual

In the context of training a neural network, we have yet to introduce a way to calculate this very important gradient of the cost function. To do so, we first need to determine a mathematical expression of the neural network itself. In fact, the network typically consists of more than just linear functions.

2.3 Activation function

Up until now, we have introduced a transition from one layer to the next as a linear transformation. The problems we want the network to solve are usually far from linear, which is why we introduce a non-linear activation function phi. We now have

$$X^{(l+1)} = \phi(WX^{(l)} + b) \tag{27}$$

There are different variants of activation functions. In some cases, we want to avoid too large values. For optimization purposes, we also want the activation function to be derivable. Both the sigmoid function defined as

$$\phi(x) = \frac{1}{1 + e^{-x}} \tag{28}$$

and the arctan function have these desired properties, though the simple ReLU defined as

$$\phi(x) = \max(0, x)$$

is shown to be more efficient for optimization (source Goodfellow) and thus more common in practice.

2.4 Softmax

Before evaluating a cost function, we transform the output layer in a way that their values are positive and their sum is equal to 1. In this way, each value in the output layer can be interpreted as the probability that a given input is classified as such. This is done using the softmax function defined as

$$\sigma(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)} \tag{29}$$

where k is equal to the number of outputs.

While in the training phase, we obviously know the desired output to each input, thus we want that specific neuron to have a value close to 1, and all the others to be close to 0. The loss function,

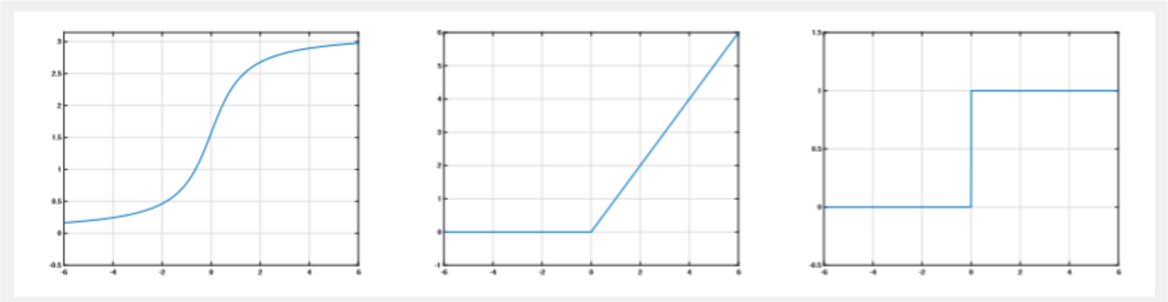

Figure 3 A visual

which quantifies “how close” the network’s guess is to a given output, can have different definitions depending on the problem. For instance, a common loss function that pairs well with the softmax function, is the negative log-likelihood defined as

$$L_i = -\log(y_i) \tag{30}$$

In order to quantify how well the network performs on the entire training set, we want to evaluate the loss function on each data-input. The cost function can then be defined as $\sum_{i=1}^{n} L_i$. Or the sum of losses.

2.5 Forward propagation

We now have an expression for each component in the neural network. Let’s see how they are assembled in the forward propagation (Figure 4).

2.6 Backward propogation

Backward propagation, finding the gradient. As previously mentioned, the NN can be seen as a composition of functions of several functions of which we can easily find the derivate. Intuitively, in order to find the gradient of the cost function with respect to the weights and biases, we can just apply the chain rule (Figure 5). As we can see, the output layer is dependent on its preceding layer, weights and biases, which, in turn, depends on the previous parameters. This creates a cascade of gradients calculated from the output- to the input layer, hence the name backward propagation.

After iterating over the training set, we now have the gradients needed for the optimization algorithm, which in turn allows us to adjust the weights and biases in order to, iteration by iteration, reach a local minimum. In practice, this approach performs relatively well, though there are improvements to be made. In the following chapters, we will examine how convolutional neural networks differs from the classic MLP.

3 *Convolutional* Neural Networks

A convolutional neural network is an evolution of a classical multilayer percepetron network. Recall the basic principle underpinning how a normal neuron in a neural network is supposed to work. The neuron is supposed to ‘look’ for features in its input data. If the neuron ‘thinks’ that those features are present in the input data it ‘fires’. Otherwise the neuron does not fire.

In a classical multilayer percepetron network this is implemented in the following way. Each neuron contains a vector of weights, a bias and an activation function. The input to the neuron—which must be a vector of equal length to the neuron’s own weight vector—is combined with the neuron’s weight vector using the dot product. The neuron’s bias is added onto the result which in turn is passed to the activation function which finally determines if the neuron ‘fires’ or not.

Using several layers of neurons one can achive quite remarkable results using this implementation of a neural network. However, a multilayer percepetron is inherently limited. The major problem is that neurons in these kinds of networks only accept input that is in the form of a vector. This means that for applications where it is not natural for the input to be in a vector format, say image recognition, the input first has to be translated to a vector format. Usually this results in a loss of information contained in the input. In the typical case of image recongition, the input is in the form of one or more arrays of two dimentions. For a multilayer percepetron to treat this input, the images has to be ‘flattend’ into a vector of one dimention before it can be passed on to the network. This procedure eliminates some of the pixel relations in the image. To deduce this, consider the process of reconstructing a flattend image. If the image's dimentions prior to being flattend is not known, it is impossible, without the aid of pattern recognition, to reconstruct the image and be sure the reconstruction is equal to the original image.

To fix this problem, we can use a 'simple' solution. Instead of having the neuron contain a vector of weights, let it have an array of weights. If we change the neuron's vector of weights into an array, we also need to change the operation that is used to combine the weights with the input (which in a neuron of a multilayer percepetron is the dot product). There are two things to consider here. The purpose of the weights is to look for features or *patterns* in the input—by emphasizing or deemphasizing certain aspects—and the operation must reflect this purpose of the weights. Furthermore, the result of the operation should be a single number which, in a sense, representes the neurons 'initial' confidence that the feature it is looking for is present in the input. The operation which does both of these things is the *Hadamard product*. The Hadamard product can be viewed as an extension of the dot product to two dimentional arrays. It combines two arrays—of the same dimentions—by multiplying corresponding entries together and summing the results. Which is precisly what the dot product does with two vectors.

The Hadamard product (Hp) of two matricies A, B with entries $a_{i,j}, b_{i,j}$ of equal dimentions is

$$\text{Hp}(A, B) = \sum_j \sum_i a_{i,j} \cdot b_{i,j} \tag{31}$$

Another thing we need to take into account is the dimentions of our new weight array. If we were to proceed analogically to how a multilayer percepetron works, the array should have the same size as the input to the neuron. 'Connecting' each value in the input to an individual weight in the neuron. But this approach means that each neuron, in principle, looks for a single feature in the entire input at once. A more refined approach, is to let the neuron's weight array be smaller in size than its input. Instead of applying the weight array (using the Hadamard product) to the entire input at once, we apply it (still using the Hadamard product) to portions of the input seperately. Intuitivly, this means that the weight arrays are 'scanned' across the entire input image. This allows the neuron to be trained to look for a single feature, such as a 'sharp edge' or a 'round corner', in multiple areas of the input. The result of this method will no longer be a single number representing how 'initialy' confident the neuron is that a particular feature is present in the input ***as a whole***. Rather, the result becomes a *feature map*. Another two dimentional array which representes how 'initialy' confident the neuron is that a particular feature is present in ***specific locations*** of the input.

Going one step further, we can allow the neuron to treat inputs of not only a single two dimentional array, but several two dimentional arrays. A typical example where the input would consists of several interlinked two dimentional arrays is RGB images. An RGB image consists of three arrays of pixel values (numbers) that desribe how red, green and blue an image is in each pixel. The number of interlinked two dimentional arrays present in the input, is known as the number of *channels* that the input has. In order for our neuron to treat inputs with more than one channel we let the neuron have as many channels as the input. That is to say, we equip the neuron with a weight array for each channel in the input. For each channel the weight arrays are applied to the input (using Hadamard) and the result in each channel is combined to form a final single feature map.

A neural network which makes use of layers of neurons of this kind, is a convolutional neural network. The multi-channel two dimentional arrays of weights inside each such neuron is called the neuron's *kernel* or *filter*. Why are these neural networks called convolutional neural networks? That will be explained in the next section.

3.1 The convolutional neuron

Let us start this section with a simple convolutional neuron. The neuron's kernel consists of a single weight matrix (one channel), some bias and some activation function. Consequently, the input that this neuron accepts is any one channel two dimentional array of size greater than its kernel.

The process of calcuating the neurons 'initial' confidence that the feature it is looking for is present in the input, is illustrated by Figure 4 on page 19. Figure 4 shows how the kernel 'scans' the entire input array to compute the feature map. Let us construct the general formula for this feature map.

We will denote the feature map, kernel and input as F, K and M respectivly. They are two dimentional arrays and their individual entries will be denoted as $F(x, y)$ where x is the column index and y the row index. The indexes will all start at zero (e.g. $M(1, 2)$ is the entry in the second column, third row of M). There are many lengths involved in working with three seperate arrays, various lengths will here be denoted by n. A subscript, either x or y, will indicate if it is a horisontal or vertical length and a subsequent argument will indicate which array the length belongs to. So for example, $n_x(M)$ is the horisontal length of the input array—the number of columns or x's in M.

If the input represents a typical MNIST image with dimentions 28x28, then $n_x(M) = 28$ and $n_y(M) = 28$. In this scenario, the indices of M range in $[0 \mathinner{..} n_x(M) - 1] \times [0 \mathinner{..} n_y(M) - 1]$.

Let us say that the kernel scans its input one column and one row at a time. In this scenario, the output feature map will have dimentions

3.2 The convolutional layer

Okay this is where I will start developing the mathematics. First we need to define the various variables used and some terminology. A network has several layers and each layer has its own specific attributes. A supersricpt encolsed in parantheses will be used as the layer index.

$$\begin{aligned} \delta x_Z^{(l+1)} &= \delta x_W^{(l)} - \delta x_Z^{(l)} + 1 \\ y_Z^{(l+1)} &= \delta y_W^{(l)} - \delta y_Z^{(l)} + 1 \end{aligned} \tag{32}$$

$$\begin{aligned} x &\in [0 \mathinner{..} n_x(F) - 1] \qquad & y &\in [0 \mathinner{..} n_y(F) - 1] \\ x &\in [0 \mathinner{..} n_x(K) - 1] \qquad & y &\in [0 \mathinner{..} n_y(K) - 1] \end{aligned} \tag{33}$$

The product in layer l before activation, here denoted Z is given by the following formula Regular formula

$$Z_{x,y,c}^{(l+1)} = \sum_{c'=0}^{\delta c_Z^{(l)}} \sum_{y'=0}^{\delta y_Z^{(l)}} \sum_{x'=0}^{\delta x_Z^{(l)}} \left(W_{x',y',c'}^{(l,c)} Z_{x+x',y+y',c'}^{(l)} \right) + b^{(l,c)} \tag{34}$$

Derivative with respect to Z. $c^* \in [c \mathinner{..} c + \delta c_W^{(l)}]$. $y^* \in [y \mathinner{..} y + \delta y_W^{(l)}]$. $x^* \in [x \mathinner{..} x + \delta x_W^{(l)}]$.

$$\begin{aligned}\frac{\partial Z^{(l+1)}_{x,y,c}}{\partial Z^{(l)}_{x^*,y^*,c^*}} &= \frac{\partial}{\partial Z^{(l)}_{x^*,y^*,c^*}}\left(\sum_{c'=0}^{\delta c_W^{(l)}}\sum_{y'=0}^{\delta y_W^{(l)}}\sum_{x'=0}^{\delta x_W^{(l)}}\left(W^{(l,c)}_{x',y',c'}\,Z^{(l)}_{x+x',y+y',c'}\right)+b^{(l,c)}\right)\\ &= \sum_{c'=0}^{\delta c_W^{(l)}}\sum_{y'=0}^{\delta y_W^{(l)}}\sum_{x'=0}^{\delta x_W^{(l)}}\frac{\partial}{\partial Z^{(l)}_{x^*,y^*,c^*}}\left(\left(W^{(l,c)}_{x',y',c'}\,Z^{(l)}_{x+x',y+y',c'}\right)+b^{(l,c)}\right)\\ &= W^{(l,c)}_{x',y',c'}\end{aligned}\tag{35}$$

Derivative with respecto to W. $c^* \in [0\,..\,\delta c_W^{(l)}]$. $y^* \in [0\,..\,\delta y_W^{(l)}]$. $x^* \in [0\,..\,\delta x_W^{(l)}]$.

$$\begin{aligned}\frac{\partial Z^{(l+1)}_{x,y,c}}{\partial W^{(l,c)}_{x^*,y^*,c^*}} &= \frac{\partial}{\partial W^{(l,c)}_{x^*,y^*,c^*}}\left(\sum_{c'=0}^{\delta c_W^{(l)}}\sum_{y'=0}^{\delta y_W^{(l)}}\sum_{x'=0}^{\delta x_W^{(l)}}\left(W^{(l,c)}_{x',y',c'}\,Z^{(l)}_{x+x',y+y',c'}\right)+b^{(l,c)}\right)\\ &= \sum_{c'=0}^{\delta c_W^{(l)}}\sum_{y'=0}^{\delta y_W^{(l)}}\sum_{x'=0}^{\delta x_W^{(l)}}\frac{\partial}{\partial W^{(l,c)}_{x^*,y^*,c^*}}\left(\left(W^{(l,c)}_{x',y',c'}\,Z^{(l)}_{x+x',y+y',c'}\right)+b^{(l,c)}\right)\\ &= Z^{(l)}_{x^*+x',y^*+y',c^*}\end{aligned}\tag{36}$$

Derivative with respect to b

$$\begin{aligned}\frac{\partial Z^{(l+1)}_{x,y,c}}{\partial b^{(l,c)}} &= \frac{\partial}{\partial b^{(l,c)}}\left(\sum_{c'=0}^{\delta c_W^{(l)}}\sum_{y'=0}^{\delta y_W^{(l)}}\sum_{x'=0}^{\delta x_W^{(l)}}\left(W^{(l,c)}_{x',y',c'}\,Z^{(l)}_{x+x',y+y',c'}\right)+b^{(l,c)}\right)\\ &= 1\end{aligned}\tag{37}$$

If we add a bias b to this filter, the formula becomes

$$F_*(x,y) = \sum_y\sum_x\Big(K(x,y)\cdot M(x+x,y+y)+b\Big)\tag{38}$$

Sending this formula throug an activation function such as ReLU or the sigmoid function, we obtain the final output feature.

$$F_*(x,y) = \text{activation}\left(\sum_y\sum_x\Big(K(x,y)\cdot M(x+x,y+y)+b\Big)\right)\tag{39}$$

3.2.1 Where is the convolution?

In mathematics, the convolution operation is denoted by $*$ and the convolution s (itself a function) of two functions k and m is defined as

$$s(t) = (k*m)(t) = \int_{-\infty}^{+\infty} k(x)\cdot m(t-x)\,\mathrm{d}x\tag{40}$$

$$s(x) = (k*m)(x) = \sum_{x=-\infty}^{+\infty} k(x)\cdot m(x-x)\tag{41}$$

For continuous (Eq. (40)) and discrete (Eq. (41)) functions respectivly. On a computer, we are in practice always working with discrete convolutions so for us the convolution of interest is Eq. (41).

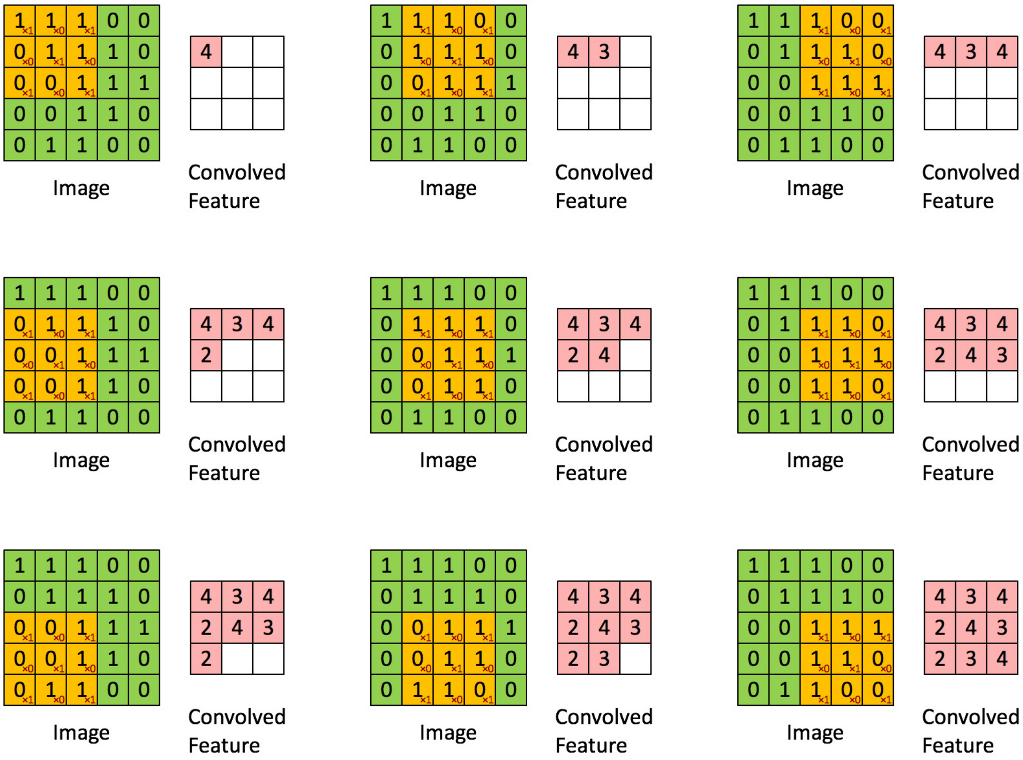

Figure 4 The basic forward operation of a convolutional layer

If we make the functions k and m two dimentional—they take two arguments as their input—their convolution also becomes two dimentional.

$$s(x, y) = (k * m)(x, y) = \sum_{y=-\infty}^{+\infty} \sum_{x=-\infty}^{+\infty} k(x, y) \cdot m(x - x, y - y) \tag{42}$$

Let us known say that k and m are two functions which each index an array of two dimentions. Meaning that the functions take two indexes as their input arguments. Let us further assume that k and m's arrays are zero at any index but for those contained in a small area. For k that area is $[0 \mathinner{..} n_x(k) - 1] \times [0 \mathinner{..} n_y(k) - 1]$ and for m it is $[0 \mathinner{..} n_x(m) - 1] \times [0 \mathinner{..} n_y(m) - 1]$. In this case, the convolution of k and m reduces to

$$s(x, y) = (k * m)(x, y) = \sum_{y=0}^{n_y(k)-1} \sum_{x=0}^{n_x(k)-1} k(x, y) \cdot m(x - x, y - y) \tag{43}$$

Where $x \in [0 \mathinner{..} n_x(m) - 1]$ and $y \in [0 \mathinner{..} n_y(m) - 1]$.

The resemblance to Eq. (34) is already apparent but there is a subtle difference. Eq. (43) subtracts the blue indices from the red ones while Eq. (34) adds them together. The effect of subtracting instead of adding renders the convolution operation $(*)$ commutative.

$$\begin{gathered}(k * m)(x, y) = (m * k)(x, y) \\ \sum_{x} \sum_{y} k(x, y) \cdot m(x - x, y - y) = \sum_{x} \sum_{y} m(x, y) \cdot k(x - x, y - y)\end{gathered} \tag{44}$$

So what is the effect of replacing the pluses in Eq. (34) with minuses? Well at first glance it seems that you cannot do it as you would end up with indexes that are out of bounds for the input image M. But let us assume that M behaves as m meaning that it is zero everywhere where the indices are out of bounds. What you find if you visualize the computation of F using Eq. (43), is that it is effectivly the same as using Eq. (34) with the kernel flipped about its horisontal and vertical axes. Which in turn means that flipping the kernel relative to the input, renders the computation of the feature map commutative. But this is a property that is usefull in mathematical contexts and less so for computing neural networks. It is Eq. (34) that is commonly used in macine learning applications and the associated mathematical operations is actually what is called the *cross-correlation*. Despite this, neural networks that implement these kinds of neurons are called convolutional even if the underlying mathematical operations is more often than not a close relative of the proper convolution operation.

3.2.2 Multiple channels

Let us return to Eq. (39) for a minute and ask: what happens if we add some channels to the input? Let us denote the number of channels by n_c. The input and kernel are now multi-channel arrays, their individual entries will be denoted as $K(i, j, c)$ with c ranging in $[0 \mathinner{..} n_c - 1]$ To calculate the feature map when multiple channels are present, we have to take into account the contribution of each individual channel. We do this by simply adding them together. Since we only want to add the bias of the filter once per entry in the feature map, the resulting modified formula becomes

$$F(x, y) = \text{activation}\left(\sum_{y}\sum_{x}\left(\sum_{c}\Big(K(x, y, c) \cdot M(x + x, y + y, c)\Big) + b\right)\right) \tag{45}$$

3.2.3 The stride

So far we have considered a feature map produced by scanning the kernel over the input one *step* at a time. As is demonstrated in Figure 4 on page 19. However, we could let the kernel take longer steps instead of moving just one row or one column at a time. The 'length' of each step that the kernel takes while moving over the input is what is known as the *stride*. Which we can further divide into the horisontal stride s_x and vertical stride s_y.

What purpose does altering the stride serve? Increasing the stride reduces the size of the resulting feature map since it is computed at fewer areas. Which has the net effect that the *computational complexity* of the neuron is reduced. However, it also means that the neuron studies its input less which can result in vital information in the input being overlooked. This is not a big problem as long as the features of the input is judged to be larger than the kernel that the neuron uses. So long as this is the case, increasing the stride of a neuron can significantly reduce its computational complexity without resulting in it no longer surving its purpose.

Let us now further develop Eq. (45) to take into account variable strides. It is as simple as multiplying the red indices with the strides.

$$F(x, y) = \text{activation}\left(\sum_{y}\sum_{x}\left(\sum_{c}\Big(K(x, y, c) \cdot M(xs_x + x, ys_y + y, c)\Big) + b\right)\right) \tag{46}$$

3.2.4 Zero padding

If we increase a neurons stride we can shrink the size of the resulting feature map. But it does not allow us to increase the size of the feature map! To do this, a techinque known as *zero padding* is

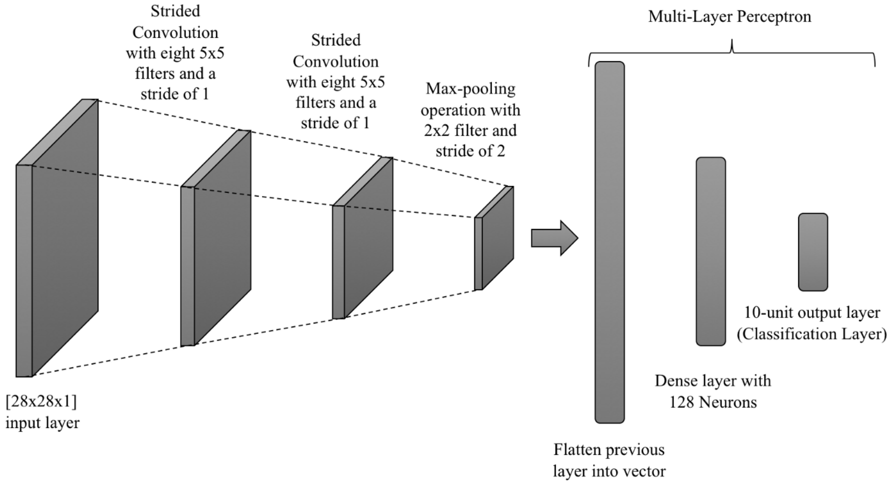
Strided Convolution with eight 5x5 filters and a stride of 1
Strided Convolution with eight 5x5 filters and a stride of 1
Max-pooling operation with 2x2 filter and stride of 2
Multi-Layer Perceptron
[28x28x1] input layer
10-unit output layer (Classification Layer)
Dense layer with 128 Neurons
Flatten previous layer into vector

Figure 5 The architecture of the network

used. Instead of working with the input *as is*, one or more extra outer rows and columns of zeros are added onto it before the kernel is scanned across it. This allows us to increase the size of the resulting feature map.

This techinque is predominantly used to make the feature map have the same dimentions as the input image in applicatons where this is desirable.

3.3 The pooling 'neuron'

Downsampling and purpose of downsampling. Noise reduction and computational reduction.

3.4 Backward Propagation

Obtain formula for derivative of convolution and pooling.

4 Python Implementation

Knwon that we have taken a look at the theory behind convolutional neural networks, let us look at an implementation of one. The nerual network we are going to be looking at is one written by Alejandro Escontrela. It is written in Python using the NumPy library and is publicly avaliable at `https://github.com/Alescontrela/Numpy-CNN.git`

The network is going to tackle the classic neural network problem. Categorizing the handwritten digits in the MNIST database. Compared to other problems convolutional neural networks are commonly faced with, categorizing the MNIST images is rather simple. So the network will use a comperativly simple architecture as shown in Figure 5 on page 21. The network uses no zero padding. The first two layer consists of two indentical convolutional layers. They are in turn follwed by a single max-polling layer which notably, uses a stride of 2. The output is then flattened and passed to a dense layer with 128 neurons. The output of this layer is passed to the final dense layer, which necessarly consists of 10 neurons.

4.1 Convolution function

Here is the python code for the forward convolution operation of the network

```
def convolution(image, filt, bias, s=1):
    '''
    Confolves `filt` over `image` using stride `s`
    '''
    (n_f, n_c_f, f, _) = filt.shape # filter dimensions
    n_c, in_dim, _ = image.shape # image dimensions

    out_dim = int((in_dim - f)/s)+1 # calculate output dimensions

    assert n_c == n_c_f, "Dimensions of filter must match dimensions of input image"

    out = np.zeros((n_f,out_dim,out_dim))

    # convolve the filter over every part of the image, adding the bias at each step.
    for curr_f in range(n_f):
        curr_y = out_y = 0
        while curr_y + f <= in_dim:
            curr_x = out_x = 0
            while curr_x + f <= in_dim:
                out[curr_f, out_y, out_x] = np.sum(filt[curr_f] * image[:,curr_y:curr_y+f,
curr_x:curr_x+f]) + bias[curr_f]
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1

    return out
```

4.2 Max pooling

Here is the python code for the forward max-pooling operation of the network

4.3 Various functions

The network uses ReLU for its activation function,

```
def maxpool(image, f=2, s=2):
    '''
    Downsample `image` using kernel size `f` and stride `s`
    '''
    n_c, h_prev, w_prev = image.shape

    h = int((h_prev - f)/s)+1
    w = int((w_prev - f)/s)+1
    downsampled = np.zeros((n_c, h, w))
    for i in range(n_c):
        # slide maxpool window over each part of the image and assign the max value
at each step to the output
        curr_y = out_y = 0
        while curr_y + f <= h_prev:
            curr_x = out_x = 0
            while curr_x + f <= w_prev:
                downsampled[i, out_y, out_x] = np.max(image[i, curr_y:curr_y+f, curr_x:curr_x+f])
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1
    return downsampled

def convolutionBackward(dconv_prev, conv_in, filt, s):
    '''
    Backpropagation through a convolutional layer.
    '''
    (n_f, n_c, f, _) = filt.shape
    (_, orig_dim, _) = conv_in.shape
    ## initialize derivatives
    dout = np.zeros(conv_in.shape)
    dfilt = np.zeros(filt.shape)
    dbias = np.zeros((n_f,1))
    for curr_f in range(n_f):
        # loop through all filters
        curr_y = out_y = 0
        while curr_y + f <= orig_dim:
            curr_x = out_x = 0
            while curr_x + f <= orig_dim:
                # loss gradient of filter (used to update the filter)
                dfilt[curr_f] += dconv_prev[curr_f, out_y, out_x] * conv_in[:, curr_y:curr_y+f,
curr_x:curr_x+f]
                # loss gradient of the input to the convolution operation (conv1 in
the case of this network)
                dout[:, curr_y:curr_y+f, curr_x:curr_x+f] += dconv_prev[curr_f, out_y,
out_x] * filt[curr_f]
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1
        # loss gradient of the bias
        dbias[curr_f] = np.sum(dconv_prev[curr_f])

    return dout, dfilt, dbias

def maxpoolBackward(dpool, orig, f, s):
    '''
    Backpropagation through a maxpooling layer. The gradients are passed through the
indices of greatest value in the original maxpooling during the forward step.
    '''
    (n_c, orig_dim, _) = orig.shape

    dout = np.zeros(orig.shape)

    for curr_c in range(n_c):
        curr_y = out_y = 0
        while curr_y + f <= orig_dim:
            curr_x = out_x = 0
            while curr_x + f <= orig_dim:
                # obtain index of largest value in input for current window
                (a, b) = nanargmax(orig[curr_c, curr_y:curr_y+f, curr_x:curr_x+f])
                dout[curr_c, curr_y+a, curr_x+b] = dpool[curr_c, out_y, out_x]

                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1

    return dout
def conv(image, label, params, conv_s, pool_f, pool_s):

    [f1, f2, w3, w4, b1, b2, b3, b4] = params

    ################################################
    ############## Forward Operation ###############
    ################################################
    conv1 = convolution(image, f1, b1, conv_s) # convolution operation
    conv1[conv1<=0] = 0 # pass through ReLU non-linearity

    conv2 = convolution(conv1, f2, b2, conv_s) # second convolution operation
    conv2[conv2<=0] = 0 # pass through ReLU non-linearity

    pooled = maxpool(conv2, pool_f, pool_s) # maxpooling operation

    (nf2, dim2, _) = pooled.shape
    fc = pooled.reshape((nf2 * dim2 * dim2, 1)) # flatten pooled layer

    z = w3.dot(fc) + b3 # first dense layer
    z[z<=0] = 0 # pass through ReLU non-linearity

    out = w4.dot(z) + b4 # second dense layer

    probs = softmax(out) # predict class probabilities with the softmax activation
function

    ################################################
    #################### Loss ######################
    ################################################

    loss = categoricalCrossEntropy(probs, label) # categorical cross-entropy loss

    ################################################
    ############# Backward Operation ###############
    ################################################
    dout = probs - label # derivative of loss w.r.t. final dense layer output
    dw4 = dout.dot(z.T) # loss gradient of final dense layer weights
    db4 = np.sum(dout, axis = 1).reshape(b4.shape) # loss gradient of final dense layer
biases

    dz = w4.T.dot(dout) # loss gradient of first dense layer outputs
    dz[z<=0] = 0 # backpropagate through ReLU
    dw3 = dz.dot(fc.T)
    db3 = np.sum(dz, axis = 1).reshape(b3.shape)

    dfc = w3.T.dot(dz) # loss gradients of fully-connected layer (pooling layer)
    dpool = dfc.reshape(pooled.shape) # reshape fully connected into dimensions of
pooling layer

    dconv2 = maxpoolBackward(dpool, conv2, pool_f, pool_s) # backprop through the max-pooling
layer(only neurons with highest activation in window get updated)
    dconv2[conv2<=0] = 0 # backpropagate through ReLU

    dconv1, df2, db2 = convolutionBackward(dconv2, conv1, f2, conv_s) # backpropagate
previous gradient through second convolutional layer.
    dconv1[conv1<=0] = 0 # backpropagate through ReLU

    dimage, df1, db1 = convolutionBackward(dconv1, image, f1, conv_s) # backpropagate
previous gradient through first convolutional layer.

    grads = [df1, df2, dw3, dw4, db1, db2, db3, db4]

    return grads, loss
```

5 Batch Optimization

When you have a large training set, it can be time consuming to evaluate a cost function on the data set in its entirety. A common way to avoid this is by instead evaluating the cost function in batches, or subsets of the training data. In other words, for each batch, we evaluate the cost function and adjust the paramaters accordingly. After using all the training data to tweak the paramaters, we have finished an epoch. This process can then be repeated over several epochs, but in our case, we have found that two epochs works sufficiently. The fact that we adjust the paramaters after each batch, means that we have to choose a suitable optimization algorithm such as Stochastic Gradient Descent. In this implementation, however, the author has used an algorithm that is shown to be very efficent in terms of batch optimization, namely the Adam Gradient Descent algorithm. We will not spend time explaining how it works, but its main principles are very similar to that of the classic gradient descent. For each training data in the batch, we evaluate the output of the loss by the forward opartion and retrieve the gradients using the backward operation described earlier. Afterwards, we evaluate the cost function of the batch and adjust the parameters. The implementation of this concept in python, is summarized in figure ()

```
def adamGD(batch, num_classes, lr, dim, n_c, beta1, beta2, params, cost):
    '''
    update the parameters through Adam gradient descnet.
    '''
    [f1, f2, w3, w4, b1, b2, b3, b4] = params

    X = batch[:,0:-1] # get batch inputs
    X = X.reshape(len(batch), n_c, dim, dim)
    Y = batch[:,-1] # get batch labels

    cost_ = 0
    batch_size = len(batch)

    # initialize gradients and momentum,RMS params

    for i in range(batch_size):

        x = X[i]
        y = np.eye(num_classes)[int(Y[i])].reshape(num_classes, 1) # convert label
to one-hot

        # Collect Gradients for training example
        grads, loss = conv(x, y, params, 1, 2, 2)
        [df1_, df2_, dw3_, dw4_, db1_, db2_, db3_, db4_] = grads

        df1+=df1_
        db1+=db1_
        df2+=df2_
        db2+=db2_
        dw3+=dw3_
        db3+=db3_
        dw4+=dw4_
        db4+=db4_

        cost_+= loss

    # Parameter Update



    cost_ = cost_/batch_size
    cost.append(cost_)

    params = [f1, f2, w3, w4, b1, b2, b3, b4]
    return params, cost

def train(num_classes = 10, lr = 0.01, beta1 = 0.95, beta2 = 0.99, img_dim = 28, img_depth
= 1, f = 5, num_filt1 = 8, num_filt2 = 8, batch_size = 32, num_epochs = 2, save_path
= 'params.pkl'):

    # training data
    m =50000
    X = extract_data('train-images-idx3-ubyte.gz', m, img_dim)
    y_dash = extract_labels('train-labels-idx1-ubyte.gz', m).reshape(m,1)
    X-= int(np.mean(X))
    X/= int(np.std(X))
    train_data = np.hstack((X,y_dash))

    np.random.shuffle(train_data)

    ## Initializing all the parameters
    f1, f2, w3, w4 = (num_filt1 ,img_depth,f,f), (num_filt2 ,num_filt1,f,f), (128,800),
(10, 128)
    f1 = initializeFilter(f1)
    f2 = initializeFilter(f2)
    w3 = initializeWeight(w3)
    w4 = initializeWeight(w4)

    b1 = np.zeros((f1.shape[0],1))
    b2 = np.zeros((f2.shape[0],1))
    b3 = np.zeros((w3.shape[0],1))
    b4 = np.zeros((w4.shape[0],1))

    params = [f1, f2, w3, w4, b1, b2, b3, b4]

    cost = []

    print("LR:"+str(lr)+", Batch Size:"+str(batch_size))

    for epoch in range(num_epochs):
        np.random.shuffle(train_data)
        batches = [train_data[k:k + batch_size] for k in range(0, int(train_data.shape[0]/16),
batch_size)]

        t = tqdm(batches)
        for x,batch in enumerate(t):
            params, cost = adamGD(batch, num_classes, lr, img_dim, img_depth, beta1,
beta2, params, cost)
            t.set_description("Cost: %.2f" % (cost[-1]))

    to_save = [params, cost]

    with open(save_path, 'wb') as file:
        pickle.dump(to_save, file)
    return cost
```

6 Results

Using our own laptops, we initially trained the network on the entire MNIST data set, which is in fact 10.000 labeled images of handwritten digits. The results are are represented on figure (). Though the network's predictions were accurate, we found that the eight-hour total run-time of the training phase was too long. To remedy this, we simply reduced the training set by a factor of 2 then by a factor of 16, respectively achieving a run-time of 4 hours and then 30 minutes.

- The problem the network is going to tackle (classic MNIST image recognition)
- The architecture of the network
- Choice of various functions and their associated code
- Results with time performance
- Possible Modifications

7 Conclusion

The advantages of a CNN over a NN.