Evaluation of Neural Networks for Image Recognition Applications: Designing a 0-1 MILP Model of a CNN to create adversarials

Lucas Schelkes
Bergische Universität Wuppertal

Abstract

Image Recognition is a central task in computer vision with applications ranging across search, robotics, self-driving cars and many others. There are three purposes of this document:

1. We follow up on (Fischetti & Jo, December, 2017) and show how standard convolutional neural network can be optimized to a more sophisticated capsule architecture.
2. We introduce a MILP model based on CNN to create adversarials.
3. We compare and evaluate each network for image recognition tasks.

General knowledge is based on (Goodfellow et al., 2016).

1. Deep Neural Networks

In the following we will model a neural network in terms of a 0-1 Mixed Integer Linear Program (MILP), not for training purposes, but to model well-suited instances for the network. On the contrary we can use the MILP to construct adversarial examples.

This chapter widely cites (Fischetti & Jo, December, 2017).

1.1. Designing a MILP of DNN

1.1.1. Notation

Let a Deep Neural Network (DNN) consist of $0, 1, \ldots, K$ layers, where $0$ marks the input layer and $K$ identifies the output layer. Each layer $k \in \{0, \ldots, K\}$ possesses $n_k$ units (or neurons) and $u(j, k)$ pinpoints the $j^{th}$ unit in layer $k$ for all $j = 1, \ldots, n_k$.

We assume a layered, fully connected network i.e. for all $j = 1, \ldots, n_k$ and $k = 1, \ldots, K$ the unit $u(j, k)$ has $n_{k-1}$ input edges. Each layer $k - 1$ is connected to the next layer $k$ with directed edges, concretely, for $u(i, k - 1)$ there exists an edge $e(u(i, k - 1), u(j, k))$ that connects directly to $u(j, k)$ for all $i = 1, \ldots, n_{k-1}$, for all $j = 1, \ldots, n_k$ and $k = 1, \ldots, K$.

The DNN can be modelled as a graph $G := (X, E)$:

Let $X := \{u(1, 0), \ldots, u(n_0, 0), \ldots, u(1, K), \ldots, u(n_K, K)\}$ be the finite set of units in the DNN.

Let

\[ E := \{ \]
\[ e(u(1, 0), u(1, 1)), \ldots, e(u(1, 0), u(n_1, 1)), \]
\[ e(u(2, 0), u(1, 1)), \ldots, e(u(2, 0), u(n_1, 1)), \]
\[ \vdots \]
\[ e(u(n_0, 0), u(1, 1)), \ldots, e(u(n_0, 0), u(n_1, 1)) \]
\[ e(u(1, K - 1), u(1, K)), \ldots, e(u(1, K - 1), u(n_K, K)), \]
\[ e(u(2, K - 1), u(1, K)), \ldots, e(u(2, K - 1), u(n_K, K)), \]
\[ \vdots \]
\[ e(u(n_{K-1}, K - 1), u(1, K)), \ldots, e(u(n_{K-1}, K - 1), u(n_K, K)) \]

\[ \} \]

be the set of edges in the fully connected DNN.

Let $W^{k-1} \in \mathbb{R}^{n_k \times n_{k-1}}$ be a given weight matrix. Each weight $w \in \mathbb{R}$ is assigned to an edge $e \in E$. We define the weight $w^{k-1}_{ij}$ to be the weight between $u(i, k - 1)$ and $u(j, k)$, which means to be the weight of $e(u(i, k - 1), u(j, k))$. We can associate $W^{k-1} \in \mathbb{R}^{n_k \times n_{k-1}}$ to be the real-valued weights on all edges between layer $k - 1$ and $k$.

Let $b^k_j \in \mathbb{R}$ be given bias inputs for all units $u(j, k)$ with $j = 1, \ldots, n_k$ and $k = 1, \ldots, K$. Note that $b^{k-1}_j \in \mathbb{R}^{n_k}$. $G$ can be modelled as a directed acyclic graph, which means that for any layer $k$ there exists no edge $e(u(\cdot, k), u(\cdot, k - i))$ for all $i = 1, \ldots, k$ and $k = 1, \ldots, K$.

Let $x^k \in \mathbb{R}^{n_k}$ be the output vector of layer $k$ and $x^0_j \in \mathbb{R}$ the output value of $u(j, k)$. Clearly $x^0$ is the input and $x^K$ the output of $G$.

For every layer the output $x^k$ of this layer can be calculated
as follows:

$$x^k := \sigma(W^{k-1}x^{k-1} + b^{k-1}), \quad k = 1, \ldots, K \quad (1)$$

with the output of the previous layer $x^{k-1} \in \mathbb{R}^{n_{k-1} \times 1}$ being multiplied with the given weight matrix $W^{k-1} \in \mathbb{R}^{n_k \times n_{k-1}}$, a given bias term $b^{k-1} \in \mathbb{R}^{n_k}$ being added and this sum being fed into a pre-defined activation function $\sigma$. There are several activation functions, which can be chosen from depending on the specific problem at hand. The activation function e.g. can make negative inputs to zero (like the ReLU function) or e.g. output a probability scale (like the sigmoid function). A non-linear activation function introduces non-linearity to a DNN, encouraging it to learn more complex functions. We set the activation function to be

$$\sigma(x) := \text{ReLU}(x) = \max\{0, x\} \quad \text{with} \quad x \in \mathbb{R}. \quad (2)$$

ReLU is a popular choice, since it is a cheaply performed operation.

Our goal is to create a MILP that modern solvers can solve - this involves not optimizing weight parameters for training, but a model which optimizes the input $x^0$ for given weight parameters, such that this input is the best classifiable instance. On the contrary this allows the construction of adversarial examples.

We need to model ReLU($x$) accordingly. From (1) we have

$$x^k := \text{ReLU}(W^{k-1}x^{k-1} + b^{k-1}), \quad k = 1, \ldots, K \quad (3)$$

in which ReLU is performed on every component of $(W^{k-1}x^{k-1} + b^{k-1}) \in \mathbb{R}^{n_k}$. To this end, we can write the linear conditions

$$W^{k-1}x^{k-1} + b^{k-1} = x^k - s^k \quad (4)$$

with $x^k \geq 0$, $s^k \geq 0$

$$\forall \ k = 1, \ldots, K.$$

To separate the positive and the negative part. If $W^{k-1}x^{k-1} + b^{k-1} \geq 0 \ \forall \ k$, we choose $x^k \geq 0 \ \forall \ k$ thus leading to $s^k = 0 \ \forall \ k$ and $W^{k-1}x^{k-1} + b^{k-1} = x^k$ (the range of $k$ is defined in (4)). If $W^{k-1}x^{k-1} + b^{k-1} \leq 0 \ \forall \ k$, we choose $s^k \geq 0 \ \forall \ k$ thus leading to $x^k = 0 \ \forall \ k$ and $W^{k-1}x^{k-1} + b^{k-1} = 0 - s^k$. Due to (3), this implicates $W^{k-1}x^{k-1} + b^{k-1} = 0 \ \forall \ k$, which reflects the ReLU property for negative input.

The above solution $x^k$ and $s^k$ is not unique, since $x^k + \delta$ and $s^k + \delta$ are also solutions for any positive $\delta$ and $\forall \ k$. To achieve uniqueness, we must find a solution in which $\delta \neq 0$ does not apply. One may think to minimize $x^k + s^k \ \forall \ k$, but this would implicate an undesired minimisation of the ReLU-function. One may also think to introduce $x^ks^k \leq 0 \ \forall \ k$ as a non-linear constraint to achieve that either $x^k$ or $s^k$ will be $0 \ \forall \ k$, however this would contradict the MILP approach of having just linear constraints.

We solve this by introducing an activation variable $z^k \in \{0, 1\}$ to the model:

$$z^k = 1 \quad \Rightarrow \quad x^k \leq M^+(1 - z^k) \quad (5)$$

$$z^k = 0 \quad \Rightarrow \quad s^k \leq M^-z^k \quad (6)$$

$$z^k \in \{0, 1\}, \quad \forall \ k = 1, \ldots, K \quad (7)$$

where $0 < M^+, M^- < \infty$ are pre-calculated bounds such that $-M^- \leq W^{k-1}x^{k-1} + b^{k-1} \leq M^+$ is valid for all $k = 1, \ldots, K$. These bounds are calculated by a MILP solver.

Conditions (5) - (7) is equivalent to

$$z^k = 1 \quad \Rightarrow \quad x^k \leq 0 \quad (5)$$

$$z^k = 0 \quad \Rightarrow \quad s^k \leq 0 \quad (6)$$

$$z^k \in \{0, 1\} \quad \forall \ k = 1, \ldots, K \quad (7)$$

If $z^k = 1 \ \forall \ k$, then $x^k = 0 \ \forall \ k$, which means that the corresponding unit in $G$ is not activated. This would encourage a trivial solution of $G$, however this is undesirable, because our goal is to find well constructed instance that are correctly classified. Consequently, we penalize the instance of $z^k = 1 \ \forall \ k$ in the cost function to avoid a trivial solution. Incorporating the binary variable $z^k$ into the objective function qualifies our model as a 0-1 Mixed Integer Linear Program.

In order for modern solvers to solve a MILP efficiently, we introduce upper- and lower bounds for $x^k$ and $s^k$:

$$lb^k_j = ub^k_j = 0, \quad k = 1, \ldots, K \quad (8)$$

$$lb^k_j \leq ub^k_j \in \mathbb{R}_+ \cup \{+\infty\} \quad (9)$$

One way of calculating tight upper bounds is to step through all units and for every $u(j, k)$ we delete all constraints and variables associated with any other unit in either the same layer or in any higher layer, and then we solve the model (10)-(12) in one round to maximize $x^k_j$ and in a second round to maximize $s^k_j$. This gives a far more accurate tight upper bounds for each units output $x^k_j$ and accelerates MILP solvers.

Putting all pieces together delivers the following 0-1 MILP model as presented by (Fischetti & Jo, December, 2017):
This model formulation is feasible, since for any fixed input $x^0$ i.e. $lb^k_j = ub^k_j \quad \forall j = 1 \ldots n_0$, every $x_k$ is uniquely defined by (3).

We seek minimal unit activation values $x_k$ in a matter such that $x^0_j \quad \forall j = 1 \ldots n_0$ is a well constructed example. Surely $x_k^0 = 0$ for all $j = 1 \ldots n_k$ and for all $k = 1 \ldots K$ is a solution, but trivially superfluous. To avoid such superfluous solution to be optimal, we set $z^k$ into the cost function, so that any unit with 0 activation value will be penalized. We seek $z$ to be 0 as often as possible i.e. we penalize the occasion ReLU$(x) = 0$. Even though the parameters for $G$ can be negative, it is only logical for a DNN to have a non-negative output. As said before, applying ReLU$(x) = 0$ encourages a trivial solution for $G$ (namely $x^0 = 0$), so we can set each $x_k^j$ to a chosen non-negative value in accordance of how greatly we want to penalize the possibility of the trivial solution problem occuring.

Furthermore, we can set $c_j^k := 1 \quad \forall j \forall k$. In this way, there are no specific units or layers of $G$ that are penalized sharper or milder.

1.2. Creating adversarial examples

The described model $G$ however is not suited for training. In a DNN we have weight parameters $w^k_j$ and $b_j^k$ to be optimized, but these are fixed in $G$. We do not have any training involved in the model. Instead, $G$ is designed to implicitly compute the best possible input example $x^0_j \quad \forall j = 1 \ldots n_0$, that can best be classified by the network.

Inversely, we can modify $G$ to compute input examples that are worst possibly classified by the network. This will result in slightly different inputs, called adversarial examples, that the DNN will missclassify upon.

One application is the MNIST dataset consisting of handwritten digits as image instances. If an image of a digit $x^0$ is classified correctly as $d$, the goal of $G$ is to find a similar image $\tilde{x}^0$ which is classified as $\tilde{d}$, with $\tilde{d} \neq d$. As (Fischetti & Jo, December, 2017) proposes, we can set $\tilde{d} = (d + 5) \mod 10$, so the adversarial image of a 3 should have label 2.

Say we want the activation of the required wrong digit in layer $K$ to be at least 20% larger than any other activation, we get

$$\tilde{x}^K_{\tilde{d}+1} \geq 1.2 \cdot \tilde{x}^K_{\tilde{d}+1} \quad \forall j = \{0, \ldots, 9\} \setminus \tilde{d}. \quad (13)$$

One can also think of modifying the cost function accordingly

$$\min \sum_{k=0}^{K-1} \min_{j=1}^{n_k} c_j^k \tilde{x}^k_j + \sum_{j=0}^{9} c_j^K \tilde{x}^K_j + \sum_{j=1}^{n_0} \epsilon_j \quad (14)$$

with $c_j^K$ as negative cost: we can encourage the activation of the required wrong digit $\tilde{d}$. Conceivably, we can further penalize high activations of the other units $\tilde{x}^K_j + 1 \forall j = \{0, \ldots, 9\} \setminus \tilde{d}$ with positive costs.

For the adversarial $\tilde{x}^0$ to be as similar as possible to $x^0$, we change every image pixel, such that the difference between them is close to 0:

$$\min \sum_{j=1}^{n_0} \epsilon_j \quad (15)$$

$$-\epsilon_j \leq x^0_j - \tilde{x}^0_j \leq \epsilon_j \quad (16)$$

$$\epsilon_j \geq 0 \quad j = 1, \ldots, n_0 \quad (17)$$

Adding (13) and (18) to $G$ modifies the model to construct adversarials $\tilde{x}^0$.

$$\min \left( \sum_{k=0}^{K-1} \sum_{j=1}^{n_k} c_j^k \tilde{x}^k_j + \sum_{j=0}^{9} c_j^K \tilde{x}^K_j + \sum_{j=1}^{n_0} \epsilon_j \right) \quad (19)$$

$$+ \sum_{k=1}^{K} \sum_{j=1}^{n_k} c_j^k \tilde{x}^k_j + \frac{n_0}{2}$$

$$\sum_{j=1}^{n_0} \epsilon_j \quad (20)$$
\[
\sum_{i=1}^{n_k-1} w_{ij}^{k-1} \tilde{x}_i^{k-1} + b_j^{k-1} = \tilde{x}_j^k - s_j^k
\]
\[
\tilde{x}_j^k, s_j^k \geq 0
\]
\[
\tilde{x}_j^k = 1 \Rightarrow \tilde{x}_j^k \leq 0
\]
\[
\tilde{x}_j^k = 0 \Rightarrow s_j^k \leq 0
\]
\[
\tilde{x}_j^k \in \{0, 1\}
\]
\[
\forall k = 1, \ldots, K, \quad \forall j = 1, \ldots, n_k
\]
\[
-\varepsilon_j \leq x_j^0 - \tilde{x}_j^0 \leq \varepsilon_j
\]
\[
0.2 \geq \varepsilon_j \geq 0
\]
\[
\forall j = 1, \ldots, n_0
\]
\[
\tilde{x}_{K+1}^D \geq 1.2 \cdot \tilde{x}_{j+1}^K \quad \forall j = \{0, \ldots, 9\} \setminus d.
\]
\[
\begin{align*}
\text{l}b_j^k & \leq \tilde{x}_j^k \leq \text{u}b_j^k \\
\text{l}b_j^k & \leq s_j^k \leq \text{u}b_j^k
\end{align*}
\]
\[
\forall k = 0, \ldots, K, \quad \forall j = 1, \ldots, n_k
\]

The additional constraint \(\varepsilon_j \leq 0.2\) for all \(j = 1, \ldots, n_0\) guarantees that no pixel is changed by more than 0.2, thus leading to more pixels to be changed in total.

The resulting adversarial images for the MNIST dataset are presented in Figure (1) and represent a first approach to building adversarials for DNNs with MILP models.

1.3. DNN Training

The general idea of DNNs in the context of supervised learning is to feed input data into the DNN, perform a forward pass, calculate the error between the output value and the desired target value, and then update the weights using the backpropagation algorithm in order to minimize the error. By doing this iteratively many times the DNN will "learn" its weights to match the output values with the desired target values.

A DNNs trainable parameters are its weights \(w_{ij}^{k}\) and biases \(b_j^k\). For training, each layer with corresponding input weights must be initialised, e.g. the \(\text{He et al., February, 2015}\) initialization draws values from a Gaussian distribution centered at 0 with standard deviation \(= \sqrt{\frac{2}{nb_{in}}}\), where \(nb_{in}\) is the number of units of the previous layer.

We define the loss function as the mean squared difference of the total error:

\[
E = \frac{1}{2M} \sum_{m=1}^{M} \sum_{i=1}^{n_K} \left( \text{target}(x_i^K(m)) - x_i^K(m) \right)^2
\]
with $M$ equals the number of instances. The desired target activation of $u(i, K)$ of the $m^{th}$ instance is defined by $\text{target}(x^K_i(m))$. The activation value of $u(i, K)$, namely $x^K_i$, for the $m^{th}$ instance is computed by (3).

Depending on the specific problem at hand, there are different loss functions (Janocha & Czarnecki, February, 2017). A forward pass is given when applying (3) to every unit in the DNN. Using the back-propagation algorithm (Lecun et al., 1989), we can calculate how much a slight shift of an individual weight parameter affects the total error. Therefore, we calculate the derivative of $E$ in respect to $w_k^j$ and solve this with the following chain rule (Avrutskiy, December, 2017).

\[
\frac{\partial E}{\partial w_k^j} = \frac{\partial E}{\partial x^K_1} \cdot \frac{\partial x^K_1}{\partial (w_1^K x_1^K + b_1^K)} \cdot \frac{\partial (w_1^k x_1^k + b_1^k)}{\partial w_j^k} \\
+ \frac{\partial E}{\partial x^K_2} \cdot \frac{\partial x^K_2}{\partial (w_2^K x_2^K + b_2^K)} \cdot \frac{\partial (w_2^k x_2^k + b_2^k)}{\partial w_j^k} \\
+ \cdots \\
+ \frac{\partial E}{\partial x^K_{n_K}} \cdot \frac{\partial x^K_{n_K}}{\partial (w_{n_K}^K x_{n_K}^K + b_{n_K}^K)} \cdot \frac{\partial (w_{n_K}^k x_{n_K}^k + b_{n_K}^k)}{\partial w_j^k}
\]

\[
\forall k = 1, \ldots, K \quad \forall j = 1, \ldots, n_k. \tag{25}
\]

Using a gradient descent method, all $w_j^k$ can be updated repeatedly by taking a step in the direction of steepest decrease of $E$ towards its minimum. For the sake of simplicity, we introduce batch gradient descent (Ruder, June, 2017):

\[
w_j^k := w_j^k - \alpha \cdot \frac{\partial E}{\partial w_j^k}, \tag{27}
\]

where $\alpha$ represents a small fixed learning rate and needs to be chosen carefully. If $\alpha$ is too small, gradient descent works too slowly; if $\alpha$ is too large, gradient descent may overshoot the minimum and may fail to converge (Karpathy & Johnson, Spring, 2018).

Batch gradient descent steps through all instances of the training set, calculates the full loss function over the entire data set, and then performs one round of weight updates (Karpathy & Johnson, Spring, 2018). In practice the training data can have millions of instances, therefore it seems wasteful to use batch gradient descent, because training would be too slow (Karpathy & Johnson, Spring, 2018). Batch gradient descent has its advantages (Karpathy & Johnson, Spring, 2018), but more commonly other optimizers are used for more efficiency (Ruder, June, 2017).

Adaptive Moment Estimation (Adam) is widely used as gradient descent optimizer, as it adaptively computes the learning rate, thus sparing the need of finding an efficient learning rate experimentally (Ruder, June, 2017).

Repeating (27) iteratively, until $E$ reaches its global minimum, is the process of DNN training. Local minima and saddle points are to be avoided (Dauphin et al., June, 2014).

### 1.4. Evaluation

As (Fischetti & Jo, December, 2017) shows, it is possible to use MILP models of DNNs to satisfyingly construct adversarial examples. The MILP approach, if at all, is not suited for finding optimal weight parameters.

We will see why DNNs are not efficient for image applications and follow up on the current state-of-the-art in image classification: Convolutional Neural Networks.
2. Convolutional Neural Networks

2.1. Introduction

In the following, we will follow up on how Convolutional Neural Networks work, access them mathematically and finally introduce a first approach of a MILP model to create adversarial examples.

It turns out that standard neural nets are inefficient in practise for image classification. One of the main reasons for this are:

1. They perform weakly with highly invariant data such as different positions of the objects in the image (Karpathy & Johnson, Spring, 2018) (see figure (2)).

2. There are extensively large amounts of network parameters to calculate e.g. an image in colour of more respectable size, e.g. 200x200x3, would mean that each neuron in the following hidden layer has $200 \times 200 \times 3 = 120,000$ weight parameters. Such great numbers of parameters increases the risk of overfitting (Karpathy & Johnson, Spring, 2018).

Convolutional neural networks (CNNs) have proven to be the current state-of-the-art architecture for classifying images using deep learning (Wikipedia, 2018c). CNNs have the desirable property of being able to detect objects, even though they might be invariantly shifted (Wikipedia, 2018c).

The best so far performance marked on the MNIST database scores an error rate of 0.21% (Wikipedia, 2018d), using complex CNNs and preprocessing measures.

We reflect more detailedly on how CNNs work in the following. Our goal is to introduce a mathematical approach on constructing adversarials with a MILP.

CNNs take advantage of image data provided in 3 dimensions: width, height and depth (number of colour channels). Concretely, CNNs consist of a sequence of layers, namely Convolutional Layer, ReLu, Pooling Layer and Fully-Connected Layer (Karpathy & Johnson, Spring, 2018).

These layers are stacked upon each other, forming the CNN’s architecture. We will see how a basic CNN architecture looks like.

2.2. Architecture

2.2.1. Input Layer

We define the input images as $A^{1}_j(i, j)$ for the height dimension $i = 1, \ldots, h$, the width dimension $j = 1, \ldots, w$ and the depth dimension $\beta = 1, \ldots, \alpha(A^1)$, where $\alpha(A^1)$ is the number of input maps (not to be confused with batch size). We assume that the next layer is convolutional (identified by superscript 1 in paragraph (2.2.2)). We think of the input images as tensors:

\[
A^1_1 = \begin{bmatrix} A^1_1(1, 1) & \ldots & A^1_1(1, w) \\ A^1_1(2, 1) & \ldots & A^1_1(2, w) \\ \vdots & \vdots & \vdots \\ A^1_1(h, 1) & \ldots & A^1_1(h, w) \end{bmatrix}
\]

(28)

\[
A^1_2 = \begin{bmatrix} A^1_2(1, 1) & \ldots & A^1_2(1, w) \\ A^1_2(2, 1) & \ldots & A^1_2(2, w) \\ \vdots & \vdots & \vdots \\ A^1_2(h, 1) & \ldots & A^1_2(h, w) \end{bmatrix}
\]

(29)

Figure 2. (Kaggle, 2014)

A DNN, which is trained with the dog in the first image, does not recognize the dog in the other images, because it is not able to learn translational invariance. It would have to learn all of these images to recognize the same dog at different positions.
Let \( A_{\alpha}^{1}(i,j) \in [0, 255] \) define the pixel integer value in the \( i \)th height, the \( j \)th width and the \( \beta \)th depth of the input image. For RGB colour images, we have \( \alpha(A^{1}) = 3 \). For these 3 colour maps, each is a \( h \times w \) tensor with integer values between [0, 255], because each colour map displays its colour array as an 8-Bit integer (Goodfellow et al., 2016). Even though this document treats image applications i.e. \( \alpha(A^{1}) \in \{1, 3\} \), we will keep the depth dimension generic.

\[
A_{\alpha}^{1}(A^{1}) = \begin{bmatrix}
A_{\alpha}^{1}(1,1) & \ldots & A_{\alpha}^{1}(1,w) \\
A_{\alpha}^{1}(2,1) & \ldots & A_{\alpha}^{1}(2,w) \\
\vdots & & \vdots \\
A_{\alpha}^{1}(h,1) & \ldots & A_{\alpha}^{1}(h,w)
\end{bmatrix}
\]

(30)

2.2.2. Convolutional Layer

**Intuition** The convolutional layer is the core element of a CNN. They consist of pre-initialised convolutional kernels of square size \( f \times f \) that perform linear combinations of pixel values (Wikipedia, 2018c).

Applying convolutional operations to an image is the biological simulation of how the human eye works. The human visual cortex incorporates receptive fields, a cluster of neurons that result in one firing neuron. The receptive field helps the retina to identify objects in their shape and colour, adjust sharpness of vision and reduce the flood of incoming information so that signals can be processed more easily (Wikipedia, 2018a). Convolutional kernels imitate properties of the receptive field. The idea is that a kernel with certain components is able to detect a certain figure in the image. According to the given components of the kernel, the kernel is able to recognize the presence of a specific figure, edge or characteristic in the image. The more accurate these components of the kernel are, the more precise it can spot that specific figure. These components can be trained, making the components the weight parameters of this layer. A kernel is an operator performing dot multiplications along the pixel values according to its kernel components, the result is a real value (figure (4)). Then the kernel moves one stride further to perform the next operation. Effectively, the kernel slides through the input maps with a certain stride length (Karpathy & Johnson, Spring, 2018), layer by layer, over the entire width and height of the input volume, while at each position an operation is executed. This is performed on all depth maps. For instance, on the left hand side figure (3) shows an image of two different coloured boxes. This kernel e.g. measures the differences between the borders of the two squares:

\[
\begin{bmatrix}
0 & 0 & 0 \\
1 & 0 & -1 \\
0 & 0 & 0
\end{bmatrix}
\]

(Bourdakos, February, 2018)

Figure 3. (Bourdakos, February, 2018)

Suppose the image is given in pixel values as in (28). The kernel given above will calculate the pixel differences at each pixel cluster - the red dot clusters have disparate differences, as do all other clusters at the borders. The kernel outputs an integer for each border cluster and zero for all other clusters. The right hand figure (3) shows the output when applying this kernel. Only vertical lines are highlighted. This illustrates how this kernel works as a receptive field in order to identify vertical lines.

We can apply further different kernels to detect horizontal lines, edges, angles etc. The task of CNNs is to learn the values of all given kernels, such that each kernel is able to identify a certain figure in the image. Multiple convolutional layers give higher level convolutional kernels that are able to detect more complex shapes and sophisticated figures (Karpathy & Johnson, Spring, 2018). The further we move up in layers, the more complex and higher level the extracted features detected by kernels get.

Since each kernel is moving across the input tensors, it is sharing its weights with many input units. This makes CNN predestined to image recognition, since it reduces the amount of computation significantly. Importantly, convolutional kernels enable already learned data to be shared across space, meaning they enable translational invariance of the input data, such as figure (2): recognizing the dog in every corner of the image, even though only one of them has been trained upon.

**Notation** Let \( 1, \ldots, C \) be the number of convolutional layers in the CNN, where \( C \) is a hyperparameter to the CNNs architecture. For each convolutional layer \( c \in \{1, \ldots, C\} \) there exist \( m^{c} \in \mathbb{N} \) kernels, where \( m^{c} \) is also a hyperparameter. Let \( k_{\gamma}^{c} \) pinpoint the \( \gamma \)th kernel in convolutional layer \( c \in \{1, \ldots, C\} \) for all \( \gamma = 1, \ldots, m^{c} \). For all
\[ \gamma = 1, \ldots, m^c \text{ and } c \in \{1, \ldots, C\} \text{ we define } k^c_\gamma \text{ to be a tensor with square size } f^c \times f^c \in \mathbb{N} \times \mathbb{N}: \]
\[ k^c_\gamma = \begin{bmatrix} k^c_\gamma(1, 1) & \ldots & k^c_\gamma(1, f^c) \\ k^c_\gamma(2, 1) & \ldots & k^c_\gamma(2, f^c) \\ \vdots & \ddots & \vdots \\ k^c_\gamma(f^c, 1) & \ldots & k^c_\gamma(f^c, f^c) \end{bmatrix}, \quad (31) \]

where \( k^c_\gamma(i, j) \in \mathbb{R} \) marks the \( i^{th} \) row and \( j^{th} \) column component of kernel \( k^c_\gamma \) with \( i = 1, \ldots, f^c \) and \( j = 1, \ldots, f^c \). Note that \( f^c \) is a hyperparameter and needs to be chosen for each layer \( c \).

Suppose that \( A^c \) denotes the input of a convolutional layer \( c \) and \( A^c_1, \ldots, A^c_{\alpha(A^c)} \) are the output maps, where \( \alpha(A^c) \in \mathbb{N} \) is the number of input maps of \( c \).

Similarly, suppose that \( B^c \) denotes the output of a convolutional layer \( c \) and \( B^c_1, \ldots, B^c_{m^c \cdot \alpha(A^c)} \) are the output maps. There are \( m^c \cdot \alpha(A^c) \) output maps of layer \( c \).

As (Zhang, 2016) suggests, one can initialize the kernels with random values drawn from a uniform distribution
\[ k^c_\gamma \sim U \left( \pm \sqrt{\frac{f^c}{(\alpha(A^c) + f^c) \cdot (m^c)^2}} \right) \quad \forall \gamma = 1, \ldots, m^c, \quad \forall c = 1, \ldots, C, \]
where \( U(\pm x) \) denotes a uniform distribution with upper and lower bounds of \( \pm x \). Bias values can also be taken into account here, however we will omit these for the sake of simplicity. Each kernel can be associated with a certain form in the image that it can analyse. Therefore it is advisable to add many different convolutional filters to the layer to achieve high accuracy of identifying details (Zhang, 2016).

The convolutional operations for every layer \( c \in \{1, \ldots, C\} \) are performed as follows:

\[ \text{for } \gamma = 1, \ldots, m^c \]
\[ \text{for } \beta = 1, \ldots, \alpha(A^c) \]
\[ \text{set } \delta := \beta + (\gamma - 1) \cdot \alpha(A^c) \]
\[ B^\delta_\gamma(1, 1) = \sum_{i=1}^{f^c} \sum_{j=1}^{f^c} A^\delta_\beta(i, j) \cdot k^c_\gamma(i, j) \quad \quad (32) \]
\[ B^\delta_\gamma(1, 2) = \sum_{i=1}^{f^c} \sum_{j=1}^{f^c} A^\delta_\beta(i + S^c, j) \cdot k^c_\gamma(i, j) \quad \quad (33) \]
\[ B^\delta_\gamma(2, 1) = \sum_{i=1}^{f^c} \sum_{j=1}^{f^c} A^\delta_\beta(i, j + S^c) \cdot k^c_\gamma(i, j) \quad \quad (34) \]
\[ B^\delta_\gamma(2, 2) = \sum_{i=1}^{f^c} \sum_{j=1}^{f^c} A^\delta_\beta(i + S^c, j + S^c) \cdot k^c_\gamma(i, j) \quad \quad (35) \]
\[ \forall c \in \{1, \ldots, C\} \]

where \( S^c \in \mathbb{N} \) is the stride step to which the kernel iterates through the rows and columns of \( A^\delta_\beta \) and \( P^c \) is the number of zero-paddings. The kernel slides through the rows and columns in steps of \( S^c \); note that each layer \( c \) has a fixed stride \( S^c \), such that \( S^c \) applies to all kernels \( k^c_\gamma, \gamma = 1, \ldots, m^c \). Clearly \( S^c \leq f^c \) \( \forall c \) must always be valid, otherwise the kernels would not overlap with the maps, thus losing information. Applying strides reduces the size of the output maps. Zero-padding \( P^c \in \mathbb{N} \) is the number of additional frames that are applied to \( A^\delta_\beta \), such that a left vertical vector of zeros, a right vertical vector of zeros, a top horizontal vector of zeros and a bottom horizontal vector of zeros are appended to tensor \( A^\delta_\beta \). This
hyperparameter can be used to control the output map size. Furthermore, it becomes evident that certain strides are not valid, since \( \frac{w^c - f^c + 2Pe^c}{Sc^c} + 1 \) needs to be an integer number\(^2\) so that convolutions on only whole pixels are provided. Concretely, the constraint
\[
(w^c - f^c) \mod S^c = 0 \quad \forall \ c = 1, \ldots, C^2 \quad (42)
\]
needs to be given for a valid stride \( S^c \). If (42) is not valid for a fixed \( S^c \), then we can see for which \( P^c \) the constraint
\[
(w^c - f^c + 2Pe^c) \mod S^c = 0 \quad \forall \ c = 1, \ldots, C^2 \quad (43)
\]
becomes valid. It is important that these constraints must simultaneously hold for each dimension.\(^2\) Note that
\[
w^c \geq \frac{w^c - f^c + 2Pe^c}{Sc^c} + 1 \quad \forall \ c = 1, \ldots, C^2 \quad (44)
\]
needs to be fulfilled in order for the size of the output maps not to become larger - this would add redundancy with many zero padds thus increasing computation. High level machine learning libraries are capable to automatically adjust (43) by adding zero-padding or cutting down \( w^c \) and \( h^c \) to make it fit (Karpathy & Johnson, Spring, 2018).

The calculations (35)-(37) define the convolutions when the filter slides column for column through the first row. Then the kernel slides column for column through the second row ((38)-(39)). This continues until the kernel convolutes through the last row ((40)-(41)). This procedure (35)-(41) represents how a kernel (32) convolutes entirely over an input map (33) and produces an output map (34) \( B^c_\delta \) (see figure (4) for illustration). Then the kernel performs the entire convolution upon all next input maps (33), and then the whole procedure is done all over again with the next kernel (32). The result are \( B^c_1, \ldots, B^c_{m^c \cdot \alpha(A^c)} \) output maps (34) each of which have size \( \left( \frac{w^c - f^c + 2Pe^c}{Sc^c} + 1 \right) \times \left( \frac{h^c - f^c + 2Pe^c}{Sc^c} + 1 \right) \) and \( B^c_\delta(i, \cdot) \in \mathbb{R} \) for all \( \delta = 1, \ldots, m^c \cdot \alpha(A^c) \) and \( c = 1, \ldots, C \).

ReLU The Rectified Linear Unit Layer (ReLU) makes all values of the convoluted image non-negative. It is an activation function defined in (2) on page 2. ReLU is not really a separate layer, but more an operation performed on the previous convolutional layer. Commonly it will be performed right after the convolutional layer and therefore counts as part of the convolutional layer (Goodfellow et al., 2016).

Let
\[
B^c_1, \ldots, B^c_{m^c \cdot \alpha(A^c)} \in \mathbb{R} \left( \frac{w^c - f^c + 2Pe^c}{Sc^c} + 1 \right) \times \left( \frac{h^c - f^c + 2Pe^c}{Sc^c} + 1 \right) \quad (45)
\]
be the output maps of a convolutional layer \( c \) and its corresponding hyperparameters, as defined in (35) - (41). We will use the notation of (11) on page 3 to model ReLU:
\[
\hat{B}^c_\delta(i, j) = \begin{cases} B^c_\delta(i, j) - s^c_\delta(i, j) & \text{if } \hat{B}^c_\delta(i, j) < 0 \\ B^c_\delta(i, j) & \text{otherwise} \end{cases} \quad (46)
\]
\[
\forall \mu = 1, \ldots, \left( \frac{h^c - f^c + 2Pe^c}{Sc^c} + 1 \right) \cdot \left( \frac{w^c - f^c + 2Pe^c}{Sc^c} + 1 \right) \quad (47)
\]
\[
\forall \delta = 1, \ldots, m^c \cdot \alpha(A^c), \quad \forall \ c = 1, \ldots, C. \quad (48)
\]
Concretely, we introduce an activation variable \( z^c_p(i, j) \in \{0, 1\} \) for every unit of map \( B^c_\beta \), for every map \( \delta = 1, \ldots, m^c \cdot \alpha(A^c) \) and every convolutional layer \( c = 1, \ldots, C \). Similar to (4) on page 2, the activation value equals 1 if the associated \( B^c_\beta(i, j) \) is negative, thus turning the associated \( B^c_\beta(i, j) \) into 0. Else, if the associated \( B^c_\beta(i, j) \) is positive, the activation value is 0, thus legitimizing the associated \( B^c_\beta(i, j) \) to be \( \hat{B}^c_\beta(i, j) \).

### 2.2.3. Pooling Layer

The function of pooling layers in a CNN architecture, is to down-size the input maps in order to reduce the amount of parameters to avoid overfitting and reduce computation (Goodfellow et al., 2016).

Pooling is similar to a kernel: it is an operation that takes pixel clusters of the input maps and combines it to one single pixel in the next layer, as can be seen in figure (6). Unlike convolutional kernels, pooling kernels do not have any component values. There are several pooling functions, we will focus on max pooling as it is commonly used. Max pooling will take the maximum value of a kernel-sized pixel cluster and project it onto the output tensor. These maximum pixel values represent the most dominant and evident shapes in the original input map, only they are passed along and the rest of the pixel information is disregarded (Karpathy & Johnson, Spring, 2018).

![figure](image_url)

**Figure 6.** Illustration of max pooling (quora, 2017)

#### Notation

Let \( 1, \ldots, \mathcal{P} \) be the number of max pooling layers in the CNN. Each max pooling layer \( p \in \{1, \ldots, \mathcal{P}\} \) consists of one max pooling kernel of which each has a predefined squared size \( \mathcal{P}^p \times \mathcal{P}^p \in \mathbb{N} \times \mathbb{N} \). Note that \( \mathcal{P} \) is a hyperparameter and needs to be chosen for each layer \( p \).

Commonly \( \mathcal{P} \in \{2, 3\} \), \( \forall \mathcal{P} = 1, \ldots, \mathcal{P} \), or else too much information is lost (Karpathy & Johnson, Spring, 2018).

Suppose that \( A^p \) denotes the input of a pooling layer \( p \) and \( A^p_1, \ldots, A^p_\alpha(A^p) \) are the input maps, where \( \alpha(A^p) \in \mathbb{N} \) is the number of input maps of \( p \). Let \( A^p_\beta \) have size \( \mathcal{P}^p \times \mathcal{P}^p \) for all \( \beta = 1, \ldots, \alpha(A^p) \) and \( p \in \{1, \ldots, \mathcal{P}\} \).

We can model a max pooling layer as follows:

\[
\text{for } \beta = 1, \ldots, \alpha(A^p) \quad B^p_\beta(1, 1) = \max \left\{ A^p_\beta(i, j) \mid \forall i, j = 1, \ldots, \mathcal{P} \right\} 
\]

\[
B^p_\beta(1, 2) = \max \left\{ A^p_\beta(i, j + \mathcal{P}) \mid \forall i, j = 1, \ldots, \mathcal{P} \right\} 
\]

\[
\vdots
\]

\[
B^p_\beta(2, 1) = \max \left\{ A^p_\beta(i + \mathcal{P}, j) \mid \forall i, j = 1, \ldots, \mathcal{P} \right\} 
\]

\[
\vdots
\]

\[
B^p_\beta(2, 2) = \max \left\{ A^p_\beta(i + \mathcal{P}, j + \mathcal{P}) \mid \forall i, j = 1, \ldots, \mathcal{P} \right\} 
\]

\[
\vdots
\]

where \( \mathcal{P} \in \mathbb{N} \) defines the specified stride length of each max pooling layer \( p \in \{1, \ldots, \mathcal{P}\} \). For each stride, \( \mathcal{P} \leq \mathcal{P} \) must be valid for all \( p = 1, \ldots, \mathcal{P} \), otherwise input pixels are skipped (Karpathy & Johnson, Spring, 2018). The stride needs to be chosen in a way that

\[
\mathcal{P}^p - \mathcal{P}^p + 1 \quad \forall p = 1, \ldots, \mathcal{P} 
\]

\[
\text{same for height } \mathcal{P}^p \text{ respectively.}
\]
is integer, in other words such that

\[ mw^p - fp^p \mod S^p = 0 \quad \forall p = 1, \ldots, P. \]  

Given a \( \beta \in 1, \ldots, \alpha(A^p) \), we are given the input map \( A^p_\beta \) (49), on which we apply the max pooling kernel at the first row through all columns (50)-(52) and storing the max value of each \( p^p \times p^p \)-pixel cluster of \( A^p_\beta \) as a single value \( B^p_\beta(1, \cdot) \). Then max pooling is performed at the second row through all columns (53)-(54), and the max values are stored in \( B^p_\beta(2, \cdot) \). This continues for all rows, until finally max pooling is done on the last row through all columns (55)-(56) with the corresponding max values stored in \( B^p_\beta(P, \cdot) \). This whole procedure is done for all input maps (49).

The result of a max pooling layer \( p \in \{1, \ldots, P\} \) are \( \alpha(A^p) \) output maps \( B^p_\beta(1, \cdot) \), \( B^p_\beta(2, \cdot) \), \( B^p_\beta(P, \cdot) \), each of which have size \( \left( \frac{m^p - fp^p}{S^p} + 1 \right) \times \left( \frac{n^p - fp^p}{S^p} + 1 \right) \) and \( B^p_\beta(i, \cdot) \in \mathbb{R} \) for all \( \beta = 1, \ldots, \alpha(A^p) \).

Modeling max pooling in a way that it can be used by modern MILP solvers, requires the use of binary activation variables e.g. given the first max pooling operation (50), we can transform it into:

\[
\begin{align*}
\sum_{\mu=1}^{p^p} z^p_{\mu} & = 1 \\
B^p_\beta(1, 1) & \geq A^p_\beta(i, j) \\
\phi^p & = 1 \rightarrow B^p_\beta(1, 1) \leq A^p_\beta(i, j) \\
z^p_{\mu} & \in \{0, 1\} \\
\forall i, j = 1, \ldots, p^p \\
\forall \beta = 1, \ldots, \alpha(A^p), \forall p \in \{1, \ldots, P\}.
\end{align*}
\]

For this first max pooling kernel, we assign \( p^p \times p^p \) times activation variables \( z^p_{\mu} \). They correspond to the first pixel cluster of the input map \( A^p_\beta \) - only one of them will be the maximum value (59), thus having activation value 1, while the rest have 0. The maximum value will satisfy (60) for all \( i, j \) (63). Specifically \( B^p_\beta(1, 1) \) is equals to the maximum, as to which the activation variable triggers (61) - the remaining activation variables equal 0 (62). This works for every input map (64) and every max pooling layer (64).

Similarly we can construct such MILP constraints for every max pooling operation (50)-(56), which we will write more generically in section (2.4) on page 12.

### 2.2.4. Fully-Connected Layer

All the pixel representations of the last layer (either convolutional or pooling) are reshaped into one long unit layer - the so called flattend layer (Wikipedia, 2018c). Then each unit of the flattend layer connects entirely to the fully-connected layer. Unlike a convolutional layer, the units in the fully-connected layer do not share weights ((Karpathy & Johnson, Spring, 2018)).

**Notation** Let \( \mathbb{A}_1, \ldots, \mathbb{A}_{\alpha(\mathbb{A})} \) be the output of the last convolutional or pooling layer of the CNN. Suppose \( \eta \times \omega \) is the size of each \( \mathbb{A}_\beta \) with \( \beta = 1, \ldots, \alpha(\mathbb{A}) \). Define

\[
\pi(\beta, (\omega \cdot (i - 1) + j)) = \mathbb{A}_\beta(i, j)
\]

for all \( i = 1, \ldots, \eta, \ j = 1, \ldots, \omega \) and \( \beta = 1, \ldots, \alpha(\mathbb{A}) \). This gives a flattend layer \( \pi \) of size \( \mathbb{R}^{(\alpha(\mathbb{A}) \times \eta \times \omega)} \) and represents merely a reshape of the pixel maps \( \mathbb{A}_1, \ldots, \mathbb{A}_{\alpha(\mathbb{A})} \) into a single long unit layer.

Let the next layer be the fully-connected layer \( \phi \) with size \( \mathbb{R}^{n_\phi \times 1} \), where \( n_\phi \in \mathbb{N} \) is the number of units of the fully-connected layer. Since we are one layer behind the output class layer, it is advisable to choose \( n_\phi < \alpha(\mathbb{A}) \cdot \eta \cdot \omega \), we want to decrease the units to get closer to the number of units in our output class layer. The flattend layer is fully-connected with the fully-connected layer i.e. there are trainable weight parameters on each edge connecting each flattend layer unit with every fully-connected layer unit. Set \( W_{\pi}^{n_\phi \times (\alpha(\mathbb{A}) \cdot \eta \cdot \omega)} \in \mathbb{R}^{n_\phi \times (\alpha(\mathbb{A}) \cdot \eta \cdot \omega)} \) to associate the weights on all edges between \( \pi \) and \( \phi \), concretely \( w_{\pi}(i, j) \in \mathbb{R} \) is the weight between unit \( i \) of the flattend layer and unit \( j \) of the fully-connected layer. The value of a fixed unit \( i \) of \( \phi \) is calculated by

\[
\phi(i) = \text{ReLU} \left( \sum_{k=1}^{\alpha(\mathbb{A}) \cdot \eta \cdot \omega} \pi(k) \cdot w_{\pi}(k, i) \right) \quad \forall i = 1, \ldots, n_\phi,
\]

we choose ReLU to be the activation function for \( \phi \).

#### 2.2.5. Output Layer

The output class layer \( \psi \in (0, 1]^{n_\psi \times 1} \) is the final layer of the CNN and corresponds to the classification of the input image, where \( n_\psi \) is the number of classes of the CNN. This layer is fully-connected to \( \phi \), i.e. we associate all weights between layer \( \phi \) and \( \psi \) as weight matrix \( W_{\phi}^{n_\phi \times n_\psi} \in \mathbb{R}^{n_\phi \times n_\psi} \). The value of a fixed unit \( i \) of \( \psi \) is calculated by

\[
\psi(i) = \text{softmax} \left( \sum_{k=1}^{n_\psi} \phi(k) \cdot w_{\phi}(k, i) \right) \quad \forall i = 1, \ldots, n_\psi,
\]
where softmax (Wikipedia, 2018e) is defined by
\[
\text{softmax} \left( \frac{\exp \left( \sum_{k=1}^{n_{\phi}} \phi(k) \cdot w_{\phi}(k, i) \right)}{\sum_{i=1}^{n_{\psi}} \exp \left( \sum_{k=1}^{n_{\phi}} \phi(k) \cdot w_{\phi}(k, i) \right)} \right) = \frac{\exp \left( \sum_{k=1}^{n_{\phi}} \phi(k) \cdot w_{\phi}(k, i) \right)}{\sum_{i=1}^{n_{\psi}} \exp \left( \sum_{k=1}^{n_{\phi}} \phi(k) \cdot w_{\phi}(k, i) \right)}. \tag{68}
\]

Each entry is a value between \((0, 1]\) and all entries add up to 1. Applying softmax as activation function to \(\psi\) allows the entries to be activation values, thus allows to interpret the activation of the output class layer as probability predictions of each class.

This part of the CNN is similar to a standard neural network and every method and technique, which applies to standard vanilla neural nets, can also be applied to this part e.g. softmax as activation, dropout measures etc (Karpathy & Johnson, Spring, 2018). Each output neuron will give a prediction in form of a probability as to whether this object class is recognized in the image. The highest probability (closest to 1) will be the total prediction for the image (Wikipedia, 2018c).

### 2.3. CNN training

Choosing a CNNs layers and hyperparameters remains an uncertainty, commonly however there are rules of thumb to the hyperparameters (Karpathy & Johnson, Spring, 2018). A basic sequence of CNN layers can be seen in figure (7).

![Figure 7. (Peng et al., 2016)](image_url)

A CNN can hold several convolutional and max pooling layers, which alternate, as in figure (8) This way, a CNN can be build to a much deeper architecture and hopes are that performance will be more accurate - not necessarily though due to overfitting and many more options to tweak (Brownlee, September, 2016).

Similarly to DNNs, the values of the convolutional filters as well as the weight matrices of the fully-connected layer are trainable parameters for CNNs. We use backpropagation and gradient descent to improve each of the filters weights (Zhang, 2016) (1.3). Cross-entropy as loss function is commonly used (Karpathy & Johnson, Spring, 2018).

![Example d’architecture](image_url)

**Example d’architecture**

ReLU is the most popular activation function in deep neural networks.

![Figure 8. (Chevalyre, 2017)](image_url)

#### 2.4. MILP formulation for a CNN

Analogously to designing a 0-1 MILP formulation for DNNs (10)-(12), we can formulate a 0-1 MILP representation for CNNs.

We set several assumptions:

- The CNN consists of \(C\) layers and each \(c \in \{1, \ldots, C\}\) is is a block: a convolutional layer (including ReLU) is followed rigidly by a max pooling layer i.e. the input of \(c\) is the output of the max pooling layer of block \(c - 1\) and the output of \(c\) is the input of the convolutional layer of block \(c + 1\). This allows the MILP to be written in an iterative structure.

The consequence is we have hyperparameters \(S^c, f^c, \ldots\) where superscript \(c\) points that this pooling layer is part of block \(c\). Note that \(\mathcal{P} = C\) for this MILP. The MILP is customizable to allow a more dynamic use of alternating layers.

- The flattened layer \(\pi\), the fully-connected layer \(\phi\) and the output class layer \(\psi\) are not part of a block and are defined separately. The flattened layer follows after the last block \(C\).

- For the sake of simplicity, bias units are omitted.

- For the sake of simplicity, we introduce substitutions (70)-(73).

- It is problematic, that the use of the non-linear softmax activation function cannot be used in a MILP model. Due to the unchanged derivative the linear approximation does not work either. This also rules out the sigmoid function as a possible alternative. This is why we will use ReLU in the output class layer as activation.

- Every convolutional kernel value \(k^c_y(i, j)\) for all \(i, j =
Hyperparameters to set:
- The number of blocks $C$ consisting of a convolutional layer, ReLU activation and max pooling.
- The size $h^1 \times w^1$ of the input images.
- The number of input channels $\alpha(A^1)$.
- The number of convolutional kernels $m^c$ for each block $c$.
- The size of the convolutional kernels $f^c$ for each block $c$.
- The size of the convolutional stride $S^c$ for each block $c$.
- The number of convolutional zero-padding $P^c$ for each block $c$.
- The size of max pooling kernel $f^c$ for each block $c$.
- The size of the max pooling stride $S^c$ for each block $c$.
- The number of units $n_\psi$ in the fully-connected layer.
- The number of classes $n_\psi$ in the output class layer.



substitutions:

\[
\begin{align*}
\tilde{h}^c & := \left( h^c - f^c + 2P^c \right) \frac{1}{S^c} + 1 \\
\tilde{w}^c & := \left( w^c - f^c + 2P^c \right) \frac{1}{S^c} + 1 \\
\tilde{h}^c & := \left( \tilde{h}^c - f^c \right) \frac{1}{S^c} + 1 \\
\tilde{w}^c & := \left( \tilde{w}^c - f^c \right) \frac{1}{S^c} + 1
\end{align*}
\]

\[
\min \left( \sum_{c=1}^{C} \left( \sum_{\beta=1}^{\alpha(A^c)} \sum_{i=\lambda}^{w^c} \sum_{\xi=1}^{1} c^{c}\beta(\lambda, \xi) \cdot A^{c}_\beta(\lambda, \xi) + \\
\sum_{\delta=1}^{m \cdot \alpha(A^c)} \sum_{\lambda=1}^{\tilde{h}^c} \sum_{\xi=1}^{\tilde{w}^c} g^{c}\delta(\tilde{\lambda}, \tilde{\xi}) \cdot B^{c}_\delta(\tilde{\lambda}, \tilde{\xi}) \right) \right)
\]

\[
\sum_{\delta=1}^{m \cdot \alpha(A^c)} \sum_{\lambda=1}^{\tilde{h}^c} \sum_{\xi=1}^{\tilde{w}^c} g^{c}\delta(\tilde{\lambda}, \tilde{\xi}) \cdot B^{c}_\delta(\tilde{\lambda}, \tilde{\xi}) +
\]
This MILP formulation is feasible, since for any fixed input $A^1\beta$, i.e. $lb(A^1_\beta)(\lambda, \xi) = ub(A^1_\beta)(\lambda, \xi)$, every other unit in the system is uniquely defined by (81)-(101).

2.4.1. Explanation

This MILP model is designed to be a minimization problem (74)-(80), in which the value of each unit in every layer is minimized.

Explanation of (74): as mentioned, this MILP is a minimization problem. The variables are mixed i.e. they are real or binary integers. The objective function and the constraints must be linear. All variables of blocks $c = 1, \ldots, C$ must be minimized, as well as all pixel values of each map, $A^c_\beta(\lambda, \xi)$, of all input maps $\beta$. The CNN’s input maps are marked as $A^0(\lambda, \xi)$, the size is $h^1 \times w^1$. The cost parameters $c^c_\delta(\lambda, \xi) \in \mathbb{R}$ can be set as 1 for all $c$, $\beta$, $\lambda$, and $\xi$ (110)-(113).

Explanation of (75): the units in $B_\delta(\tilde{\lambda}, \tilde{\xi})$ represent the convoluted units. The number of maps make up the number of input maps times the number of kernels used. The size of these maps have changed depending on the kernel size, the stride and the padding. The cost parameters $q^c_\delta(\lambda, \xi) \in \mathbb{R}$ can be set as 1 for all $c$, $\delta$, $\lambda$, and $\xi$ (115)-(117).

Explanation of (76): the convoluted maps are fed through the ReLU function, which gives us the unit maps. The number of maps make up the number of input maps times the number of kernels used. The size of these maps have changed depending on the kernel size, the stride and the padding. The cost parameters $l^c_\delta(\lambda, \xi) \in \mathbb{R}$ can be set as 1 for all $c$, $\delta$, $\lambda$, and $\xi$ (115)-(117).

Explanation of (77): the binary variable $Z^c_\delta(\tilde{\lambda}, \tilde{\xi})$ corresponds as an activation variable used for the ReLU function. We penalize the occurrence of $Z^c_\delta(\tilde{\lambda}, \tilde{\xi}) = 1$, we may initialize the cost parameters $v^c_\delta(\lambda, \xi)$ in a way in which how much we want to penalize this occurrence, possibly between $[0, 1]$.

Explanation of (78): this minimizes the binary activation variable $\tilde{z}^c_\delta(\tilde{\lambda}, \tilde{\xi})$ used for max pooling. For each pooling kernel $\Delta$ (118) in an input map $\delta$, we need to associate each entry $\tilde{\lambda}$, $\tilde{\xi}$ (119)-(120) with this 0-1 activation variable.

We may set the cost parameters e.g. $O^c_{\tilde{\lambda}, \delta}(\tilde{\gamma}, \tilde{\xi}) \in [0, 1]$.
before the next kernel (121) repeats this procedure. As a result we have \( B_k^f(\lambda, \xi) \) with changed size.

Explanation of (82): by introducing variable \( s_k^f(\lambda, \xi) \), we can tackle a non-linear activation function to fit into a MILP. For this we introduce an activation variable \( z_k^f(\lambda, \xi) \).

The purpose of (83)-(86) matches the thoughts of (5)-(7).

Explanation of (87): we define an activation variable \( \zeta_{\delta, j}(i, j) \) (90) for each input map \( \delta \), we have \( \Delta \) pooling kernels and each kernel has \( f \times f \) (122) components. We need to step through each component and guarantee that each kernel only activates one \( \zeta \) (87). This activation identifies the max value of the pixel cluster in maps \( B_k^f \) (88)-(89).

The constraints (81)-(90) characterize the convolution-, the ReLU and the max pooling layers in a block \( c = 1, \ldots, C \). The last block \( C \) will give us maps \( A_{\tilde{\lambda}}^{C+1} = \tilde{\lambda}, \tilde{\xi} \) for all \( \tilde{\lambda} \) (119) and \( \tilde{\xi} \) (120). We use these maps and flatten them out, row by row, map by map, into the flattened layer \( \pi \) (91).

Explanation of (92): we then construct the fully-connected layer \( \phi \). Every component of \( \phi \) (123) is a linear combination of each flattend layer component together with a connecting weight. Every component \( f \) of the flattend layer is connected with every component \( i_\xi \) of the fully-connected layer with corresponding weight parameters \( w_{\pi}(f, i_\xi) \) \( \in \mathbb{R} \). Note that the size of the maps \( \tilde{\lambda}^C \times \tilde{\xi}^C \) is unchanged from the last block. For ReLU activation, we introduce variable \( \tilde{s} \) (93) and binary activation variable \( \tilde{\zeta} \) (94). For (95)-(96) we proceed in similar terms to (85)-(86).

Explanation of (97): finally we introduce the output class layer \( \psi \) with \( n_\psi \) classes (124), whose weights are fully connected to the previous fully-connected layer. As pointed out in the assumptions, we cannot use sigmoid or softmax as activation functions, but we know how to construct ReLU as linear constraint for this MILP: (98)-(101) is an identical approach to the ReLU activation above.

Explanation of (102)-(109): similar to (12), we introduce upper- and lower bounds for each unit in order for modern MILP solvers to work more efficiently:

\[
lb(A_{\tilde{\lambda}}^f)(\cdot, \cdot) = 0 \quad \forall \; c > 1, \; \forall \; f \quad (125)
\]

\[
lb(B_{\tilde{\lambda}}^f)(\cdot, \cdot) = 0 \quad \forall \; \delta, \; \forall \; c \quad (126)
\]

\[
lb(\tilde{B}_{\tilde{\lambda}}^f)(\cdot, \cdot) = 0 \quad \forall \; \delta, \; \forall \; c \quad (127)
\]

\[
lb(\pi)(\cdot) = 0 \quad (128)
\]

\[
lb(\phi)(\cdot) = 0 \quad (129)
\]

\[
lb(s_k^f)(\cdot, \cdot) = 0 \quad \forall \; \delta, \; \forall \; c \quad (130)
\]

\[
lb(\tilde{s})(\cdot) = 0 \quad (131)
\]

\[
lb(\tilde{\tilde{s}})(\cdot) = 0 \quad (132)
\]

\[
ub(A_{\tilde{\lambda}}^f)(\cdot, \cdot) \in \mathbb{R}_+ \cup \{+\infty\} \quad \forall \; c > 1, \; \forall \; f \quad (133)
\]

\[
ub(B_{\tilde{\lambda}}^f)(\cdot, \cdot) \in \mathbb{R}_+ \cup \{+\infty\} \quad \forall \; \delta, \; \forall \; c \quad (134)
\]

\[
ub(\tilde{B}_{\tilde{\lambda}}^f)(\cdot, \cdot) \in \mathbb{R}_+ \cup \{+\infty\} \quad \forall \; \delta, \; \forall \; c \quad (135)
\]

\[
ub(\pi)(\cdot) \in \mathbb{R}_+ \cup \{+\infty\} \quad (136)
\]

\[
ub(\phi)(\cdot) \in \mathbb{R}_+ \cup \{+\infty\} \quad (137)
\]

\[
ub(s_k^f)(\cdot, \cdot) \in \mathbb{R}_+ \cup \{+\infty\} \quad \forall \; \delta, \; \forall \; c \quad (138)
\]

\[
ub(\tilde{s})(\cdot) \in \mathbb{R}_+ \cup \{+\infty\} \quad (139)
\]

\[
ub(\tilde{\tilde{s}})(\cdot) \in \mathbb{R}_+ \cup \{+\infty\} \quad (140)
\]

One way of calculating tight upper bounds is to step through all units: we fix a unit and delete all constraints and variables associated with any other unit in either the same layer or in any higher layer, and then we solve the model (74)-(124) in one round to maximize each unit. This gives as a far more accurate tight upper bounds for each unit’s output and accelerates MILP solvers.

### 2.5. Creating adversarial examples

The described MILP model in (2.4) however is not suited for training. We have kernel values \( k_\xi^f(i, j) \) and weight parameters in the fully-connected layer \( w_\pi(f, i_\xi) \) and \( w_\phi(f, i_\psi) \) to be optimized, but these are fixed initializations in (2.4). We do not have any training elements involved. Instead, the MILP is designed to implicitly compute the best possible input example \( A_{\tilde{\lambda}}^f(\lambda, \xi) \) \( \forall \; \lambda = 1, \ldots, h_1, \; \forall \; \xi = 1, \ldots, h_1 \) that can best be classified by the network.

Inversely, we can modify the MILP to compute input examples that are worst possibly classified by the network. This will result in slightly different inputs, called adversarial examples, that the CNN will misclassify upon. This procedure is analogous to (1.2) where we found adversarials with DNNs. Again, we will take the MNIST dataset to base the MILP upon. If an image of a digit \( A_{\tilde{\lambda}}^f \) is classified correctly as \( d \), the goal of our CNN MILP is to find a similar image \( \hat{A}_{\tilde{\lambda}}^f \) which is classified as \( \hat{d} \), with \( \hat{d} \neq d \). As already done by (Fischetti & Jo, December, 2017), one way is to set \( \hat{d} = (d + 5) \mod 10 \), so the adversarial image of a 3 should have label 2.

Say we want the activation of the required wrong digit in the output class layer to be at least 20% larger than any other activation, we get

\[
\tilde{\psi}(\hat{d}) \geq 1.2 \cdot \tilde{\psi}(i_\tilde{\psi}) \quad \forall \; i_\tilde{\psi} = \{1, \ldots, 10\} \setminus \hat{d}, \quad (141)
\]

where \( \tilde{\psi}(\cdot) \) corresponds to the output class layer units of the adversarial. One can also think of modifying the cost function accordingly

\[
\min \sum_{i_\tilde{\psi}=1}^{10} \hat{c}(i_\tilde{\psi}) \cdot \tilde{\psi}(i_\tilde{\psi}) \quad (142)
\]

with \( \hat{c}(\hat{d}) \) as negative cost: we can encourage the activation of the required wrong digit \( \hat{d} \). Conceivably, we can further
penalize high activations of the other units $\tilde{\psi}(i_{\tilde{\psi}}) \forall i_{\tilde{\psi}} = \{1, \ldots, 10\}\setminus \tilde{d}$ with positive costs.

For the adversarial $A_\beta^1$ to be as similar as possible to $A_\beta^2$, we change every image pixel, such that the difference between them is close to 0:

$$\min \sum_{\beta=1}^{\alpha(A^1)} \sum_{h=1}^{w^1} \sum_{\lambda=1}^{A^1} \sum_{\xi=1}^{A^1} \epsilon_\beta(\lambda, \xi)$$

(143)

$$- \epsilon_\beta(\lambda, \xi) \leq A_\beta^1(\lambda, \xi) - A_\beta^2(\lambda, \xi) \leq \epsilon_\beta(\lambda, \xi)$$

(144)

$$\epsilon_\beta(\lambda, \xi) \geq 0$$

(145)

$$\beta = 1, \ldots, \alpha(A^1)$$

(146)

$$\lambda = 1, \ldots, h^1$$

(147)

$$\xi = 1, \ldots, w^1.$$  

(148)

An additional constraint $\epsilon_\beta(\lambda, \xi) \leq 0.2$ for all (146)-(148) guarantees that no pixel is changed by more than 0.2. This means that instead of few significantly changed pixels, instead more pixels are changed in total less significantly.

By adding (142) and (143) to the objective function of the MILP, as well as constraints (144)-(148) and (141) to the MILP, we can construct such an adversarial example $A_\beta^1$.

### 2.6. Evaluation

Historically CNNs are the most popular form of artificial neural network to perform image application tasks e.g. LeNet-5 (LeCun et al., 1998) as one of the first CNNs by Yann Lecun (Lecun et al., 1989), deep CNN AlexNet (Krizhevsky, 2009) used for CIFAR dataset classification and first introduced ReLU activation (Karpathy & Johnson, Spring, 2018), or deep CNN ResNet (He et al., December, 2015) using network layers to fit a residual mapping (Karpathy & Johnson, Spring, 2018). For good reason, because they all achieved state-of-the-art accuracy at moderate to affordable computational cost (Karpathy & Johnson, Spring, 2018). They maintain translational invariance and shared weight parameters is more efficient than vanilla DNNs.

Feeding the MILP model (2.4) into a modern MILP solver (CPLEX) to create adversarial examples remains to be done. These may then be compared to the results of the DNN MILP presented by (Fischetti & Jo, December, 2017), in order to arrive at an evaluation which MILP model is more suitable.

### 3. Capsule Networks

This chapter is widely based on (Hinton et al., 2017) by Geoffrey Hinton.

#### 3.1. Equivariance

With the help of convolutional kernels, CNNs manage to detect objects in images that are translational invariant (2). The convolutional filters move iteratively through all image areas, which makes it possible to translate the same structure detected in one region across the entire image and keep that information stored within its kernel weights. Training a CNN with a center-positioned object, enables the CNN to classify that same object to the left, right, up or down (2). This is why CNNs are robustly resistant to translational invariance (Goodfellow et al., 2016). However, this does not apply for any transformation applied to the object in the image. If the object is rotated, flipped, scaled, deformed, reassembled etc. the CNN fails to classify the object correctly (Hinton et al., 2017). Further training with these transformed objects would be needed and further kernels would be needed to cover all the different transformations. In short: CNN lack equivariance (Hinton et al., 2017).

Equivariance is a concept to describe how objects still stay the same, eventhough they are transformed, rotated, in a different light etc. and thus need to be classified as the objects they are, regardless of any transformation (Hinton et al., 2017). This concept matches how human vision works: when an object is viewed, the human eye sets fixation points to process the area of the object at an high resolution while ignoring irrelevant details (Hinton et al., 2017). Furthermore, the brain is capable to understand the instantiation of a transformed object, in order to identify it. This is the reason and motivation for (Hinton et al., 2017) to introduce a new kind of architecture to classify objects equivariantly - in the following, we follow up on this innovative architecture for computer vision, namely Capsule Networks (CapsNets) (Hinton et al., 2017). CapsNets expand on human vision such that objects are detected as such, regardless of any transformation in the image. Also, in crowded scenes with overlapping elements each element can segmentedly be classified (Hinton et al., 2017).

CapsNets are based on CNNs, replacing max pooling by an routing-by-agreement algorithm (Hinton et al., 2017). Max pooling extracts the major element in a conv map and erases the rest. This causes a lot of information to be lost, moreover it does not serve to achieve equivariance (Hinton et al., 2017). Maintaining translational invariance however in addition to equivariance is certainly desired for CapsNets.

In the following, we discuss the CapsNet architecture presented in (He et al., February, 2015) and our goal is to arrive to an evaluation, comparing CapsNet to state-of-the-art
CNNs in image recognition applications.

The dataset on which CapsNet in (Hinton et al., 2017) is based on, is the MNIST dataset ((Wikipedia, 2018d)).

3.2. CapsNet Architecture

3.2.1. INPUT LAYER

Let the input be a grey-scaled $28 \times 28$ MNIST image i.e. a tensor of integer values ranging between $[0, 255]$.

3.2.2. CONVOLUTIONAL LAYER

The first layer is a standard convolutional layer with included ReLU operation. We maintain convolutional layers for CapsNets, because we want to "replicate learned knowledge across space" (Hinton et al., 2017), thus preserving translational invariance. This first layer of convolutional operations is looking for low level edges and simple curves in the input image. 256 different $9 \times 9$ convolutional kernels are applied, with a stride of 1 and no padding. This gives an output of $256 \times [20 \times 20]$ maps.

Let $m_\beta$ denote these 256 maps for $\beta = 1, \ldots, 256$, and $m_\beta(k, l)$ be the value in row $k$ and column $l$ of the $\beta$th map, $\forall \beta = 1, \ldots, 256, \forall k = 1, \ldots, 20, \forall l = 1, \ldots, 20$. ReLU (2) is applied as activation function on all $m_\beta(k, l)$ $\forall \beta = 1, \ldots, 256, \forall k = 1, \ldots, 20, \forall l = 1, \ldots, 20$. So we have 256 convolutional maps of size $9 \times 9$ consisting of non-negative integers. An additional bias term leads to $(9 \times 9 + 1) \times 256$ trainable parameters in this layer (Pechyonkin, February, 2018).

3.2.3. PRIMARY CAPS LAYER

So called PrimaryCapsules (Hinton et al., 2017) is a convolutional layer with 32 "primary capsules" of convolutional 8-D capsules. Concretely, we have 32 multidimensional "kernels" (or primary capsules), each consisting of 8 convolutional $9 \times 9$ kernels and a stride of 2, that sees the whole input of the $m_1, \ldots, m_{256}$ maps. The result is 8-D $32 \times [6 \times 6]$ maps. Basically the 256 input maps are restacked into 32 "decks" with 8 maps each deck. This gives us 32 times $[6 \times 6]$ stacks, each consisting of 8-D vector components.

Let $\tilde{m}_1, \ldots, \tilde{m}_{32}$ be the stacked decks, and $\tilde{m}_\beta(k, \tilde{l})$ be the vector in row $k$ and column $\tilde{l}$ in the $\beta$th deck, $\forall \beta = 1, \ldots, 32, \forall k = 1, \ldots, 6, \forall \tilde{l} = 1, \ldots, 6$. Then the 8-D vector $v = (v_1, \ldots, v_8)$ in $\tilde{m}_\beta(k, \tilde{l})$ is defined by

$$v_1 = m_1^{\beta}(k, \tilde{l}),$$
$$v_2 = m_2^{\beta}(k, \tilde{l}),$$
$$v_3 = m_3^{\beta}(k, \tilde{l}),$$
$$v_4 = m_4^{\beta}(k, \tilde{l}),$$
$$v_5 = m_5^{\beta}(k, \tilde{l}),$$
$$v_6 = m_6^{\beta}(k, \tilde{l}),$$
$$v_7 = m_7^{\beta}(k, \tilde{l}),$$
$$v_8 = m_8^{\beta}(k, \tilde{l}),$$

$\forall \beta = 1, \ldots, 32, \forall k = 1, \ldots, 20, \forall \tilde{l} = 1, \ldots, 20$, where $m^\beta$ represents the convoluted map of $m$. Each deck $\tilde{m}_\beta$ is called capsule layer and each component $\tilde{m}_\beta(k, \tilde{l})$ is called capsule. Clearly each capsule layer has 36 capsules. One can think of an capsule as a group of neurons that collectively produce an activity vector with one element for each neuron. On the one hand, vanilla DNNs output a scalar value for each neuron, which illustrates how high the activation of that neuron is, in other words whether a certain entity of an object is present or not (Wikipedia, 2018c). With CapsNets, on the other hand, a capsule outputs an activity vector (Hinton et al., 2017).

Since each of the 32 primary capsules applies eight $9 \times 9 \times 256$ convolutional kernels to the layer’s input, this leads to $32 \times 8 \times (9 \times 9 + 1) \times 256$ trainable parameters in this layer (including a bias term) (Pechyonkin, February, 2018) that are trained by backpropagation.

**Activity vector** Computer graphic programs take instantiation parameters of an object as input and then output the rendered image. Inverse rendering implies taking an image and finding out the instantiation parameters of existing objects. It is the major task of CapsNets to learn instantiation pa-
rameters of objects in an image, in order to achieve equivariance (Géron, 2017). An objects instantiation parameters include exact position, size, deformation, rotation degree, velocity, lighting, albedo, hue, texture etc (Géron, 2017). The instantiation parameters are represented by the orientation of the activity vector of a capsule (Hinton et al., 2017). The length of the activity vector marks the probability that a certain entity exists, analogous to the activation value of single-scalared DNN neurons (Hinton et al., 2017).

3.2.4. DIGITCAPS LAYER

In the following procedure the activity vectors are "squashed" applying a squashing function (Hinton et al., 2017):

\[
v_j = \frac{||s_j||_2^2}{1 + ||s_j||_2^2} s_j. \tag{149}
\]

The squashing function scales a vector \( s_j \) to have length between 0 and 1, while maintaining the orientation. It ensures that long vectors get shrunk to a length slightly below 1 and short vectors get shrunk to almost zero length (Hinton et al., 2017).

The squashing function (149) introduces non-linearity to the CapsNet and acts as an activation function: since the length of the activity vector represents the probability that an object exists, it is desired to have vector lengths not exceeding 1 and not be inferior to 0.

We apply the squashing function to all 8-D \( 32 \times [6 \times 6] \) capsules; we obtain normalized activity vectors, the orientation stays identical.

Then the 8-D \( 32 \times [6 \times 6] \) capsules are reshaped into 8-D \( [1 \times 1] \times 1152 \) capsules called \( u_i \in \mathbb{R}^{8 \times 1} \) \( \forall i = 0, \ldots, 1151 \), with

\[
u_0 = \tilde{m}_1(1,1)_{sq} \\
u_2 = \tilde{m}_1(1,2)_{sq} \\
vdots \\
u_6 = \tilde{m}_1(1,6)_{sq} \\
u_7 = \tilde{m}_1(2,1)_{sq} \\
vdots \\
u_{12} = \tilde{m}_1(2,6)_{sq} \\
vdots \\
u_{36} = \tilde{m}_1(6,6)_{sq} \\
u_{37} = \tilde{m}_2(1,1)_{sq} \\
vdots
\]

where \( \tilde{m}_2(\hat{k},\hat{l})_{sq} \) notates the squashed capsule of \( \tilde{m}_2(\hat{k},\hat{l}) \) \( \forall \hat{k} = 1, \ldots, 32, \forall \hat{l} = 1, \ldots, 6 \).

Now each capsule \( u_i \) is multiplied with an individual weight matrix \( W_{ij} \in \mathbb{R}^{16 \times 8} \)

\[
\hat{u}_{ji} = W_{ij} \cdot u_i, \tag{150}
\]

with \( u_i \) as input capsule \( \forall i = 0, \ldots, 1151 \) and \( j = 0, \ldots, 9 \) the number of digit classes. The weight matrix is an affine transformation matrix and stores learnable information about the exact part-whole relationship of each entity of the entire object. It is initialized at the beginning and the values are learned with backpropagation iteratively. Each capsules activity vector stores instantiation parameters of an objects entity and the weight matrix stores data for the exact spatial relationship of the entity regarding the entire object. E.g. if the input image is a face and capsule \( u_i \) stores information about the eye, then the weight matrix gives information on how exactly the eye is to be spatially positioned in the face, such that essentially a proper face is detected. In CNNs, such spatial information is not stored, which is why Pablo Picassos famous "Portrait of woman in d'hermine pass" figure (10) will falsely be classified by CNNs as a face. This makes CNNs vulnerable to adversarial attacks.

The matrices \( W_{ij} \) represent a part-whole relationship. This tells us in what manner a lower level entity, a capsule \( u_i \), fits in to with a higher level entity/object. E.g. \( u_i \) may store information about a curve, and \( W_{i0} \) represents in what posture curves fit into the digit "0", then \( W_{i0} u_i = \hat{u}_{i0|i} \) has significant length; whereas there are no curves in the digit "1", consequently \( W_{i1} u_i = \hat{u}_{i1|i} \) is a short activity vector for that same curve capsule \( u_i \).

As in (150), for all \( i = 0, \ldots, 1151 \) every capsule \( u_i \) is transformed into \( \hat{u}_{i|j} \in \mathbb{R}^{16 \times 1} \) via matrix-vector multiplication\(^4\) with every digit class \( j = 0, \ldots, 9 \). The result is 16-D \( 10 \times [1152 \times 1] \) capsules, see figure (11), noted as

\[
\hat{u}_{0|0}, \ldots, \hat{u}_{0|1151} \quad \text{for digit 0} \\
\hat{u}_{1|0}, \ldots, \hat{u}_{1|1151} \quad \text{for digit 1} \\
vdots \\
\hat{u}_{9|0}, \ldots, \hat{u}_{9|1151} \quad \text{for digit 9}.
\]

So we have 1152 16-D capsules for each of the ten digits.

\(^4\)This works, since \( W_{ij} \in \mathbb{R}^{16 \times 8} \) and \( u_i \in \mathbb{R}^{8 \times 1} \).
Each of the 1152 $u_i$ capsules is multiplied with an own $16 \times 8$ matrix $W_{ij}$, so we have $1152 \times 16 \times 8$ trainable weight parameters here, that need to trained using back-propagation.

The concept that higher level capsule layers produce higher dimensional capsules matches the idea of higher level convolutional filters producing more complex forms and shapes. High dimensional capsules can store considerably more instantiation parameters for a more complex entity of the object. Finally the last layer of capsules, DigiCaps, represent ten capsules for each class to be detected. They are created by using routing-by-agreement algorithm.

Routing-by-agreement algorithm For each $j = 0, \ldots, 9$ the 16-D capsules $\hat{u}_{j0}, \ldots, \hat{u}_{j1151}$ are summed up in a way that highly significant capsules with good predictions are weighted profoundly, and less significant capsules have less influence on the next capsule layer (Hinton et al., 2017). Hinton’s paper (Hinton et al., 2017) introduces a routing by agreement algorithm, that iteratively sets these weights, $c_{ij}$, and serves as a better way than max pooling to prioritize the dominant features, while at the same time preserving the less relevant data.

**Algorithm 1** Routing algorithm (Hinton et al., 2017)

For every capsule $u_i$ in layer $l$ we have the prediction $\hat{u}_{j|i}$ available, which is calculated by (150).

On the one hand, if the entity represented of capsule $u_i$ in layer $l$ is not related in any way to the higher level entity of capsule $v_j$ in layer $l + 1$, then $\hat{u}_{j|i}$ will only have marginal impact, thus it is the goal of (1) to assign only marginal $c_{ij}$. On the other hand, if many capsules $u_i$ in layer $l$ have similar $\hat{u}_{j|i}$ for capsule $v_j$ in layer $l + 1$, then the corresponding $c_{ij}$ will be major, because the capsules agree on what the object or entity of an object looks like, thus there is no need to send large weight $c_{ij}$ to any other capsule. This would only cause noise.

With their calculation $\hat{u}_{j|i}$ the capsules $u_i$ in layer $l$ try to predict the capsule $v_j$ in layer $l + 1$.

3.2.5. EXAMPLE

In the following we follow up an example, introduced by (Géron, 2017), to better understand the dynamic routing between capsule.

Suppose a boat image such as in (12) is the input to CapsNet. As we see in (13), a triangle capsule $u_1$ (blue) and a rectangle capsule $u_2$ (black) become active. The corresponding activity vectors are present, the orientation marks the instantiation of that entity and the length marks the probability. Note that other image areas will result in short vectors for these two specific capsules, we omit these in the framework for the sake of simplicity.

Suppose the capsule class layer $l + 1$ consists of only two capsule classes: one capsule $v_1$ identifying houses, the other $v_2$ identifying boats.

For $u_1$:

According to the orientation of $u_1$, the house capsule $v_1$ in layer $l + 1$, will predict an upside-down house, this is stored in $\hat{u}_{111}$. The information explaining the way the roof fits to the house is stored in the weight matrix $W_{11}$. On the other hand, the boat capsule $v_2$ predicts a slightly
Neural Networks for Image Recognition

Figure 11. A visualization of PrimaryCaps and DigitCaps (Ross, November, 2017)
Caution, this figure has an error: $\hat{u}_{i|j}$ instead of $\hat{u}_{j|i}$

Figure 12. (Géron, 2017)

tilted boat for $u_1$, this is stored in $\hat{u}_{2|1}$. The information explaining the way the sail fits to the boat is stored in $W_{12}$. For $u_2$:

According to the orientation of $u_2$, the house capsule $v_1$ will recognize a slightly tilted house, this is stored in $\hat{u}_{1|2}$. Whereas the boat capsule $v_2$ recognizes a slightly tilted boat from $u_2$, this is stored in $\hat{u}_{2|2}$.

It is evident that the capsules in layer $l$ strongly agree on what a boat should look like ($\hat{u}_{2|1}$ and $\hat{u}_{2|2}$ match) and strongly disagree on what a house should look like ($\hat{u}_{1|1}$ and $\hat{u}_{1|2}$ differ). Therefore it is quite likely that the unfamiliar object is essentially a boat. Both capsules in layer $l$ should therefore send most of their output to the boat capsule $v_2$ in layer $l+1$ and only little to the house capsule $v_1$. Concretely, by applying the (1) routing-by-agreement algorithm, we can calculate appropriate weights $c_{ij}$ in order to distribute high weights to well predicting capsules and low weights to poor predicting capsules.

We can think of $u_1$ and $u_2$ to be the activity vectors shown in figure (13). Calculate each $\hat{u}_{j|i}$:

$$\hat{u}_{1|1} = W_{11} u_1$$
$$\hat{u}_{1|2} = W_{21} u_2$$
$$\hat{u}_{2|1} = W_{12} u_1$$
$$\hat{u}_{2|2} = W_{22} u_2,$$

which is the input to algorithm (1). We can think of the $\hat{u}_{1|1}$, $\hat{u}_{1|2}$, $\hat{u}_{2|1}$, $\hat{u}_{2|2}$ to be the activity vectors shown in figure (13). Following the first step, we receive

$$b_{11} = 0$$
$$b_{12} = 0$$
$$b_{21} = 0$$
Softmax is similar to the sigmoid function: delegate value between 0 and 1 and all values sum up to 1. It must be $\sum_i c_{ij} = 1$ (capsule $i$ in layer $l$) for all capsules $j$ in layer $l + 1$. I.e. $\sum_{i=1}^{2} c_{i1} = 1$ and $\sum_{i=1}^{2} c_{i2} = 1$.

The next step is calculating $s_1$:

$$s_1 = \sum_i c_{i1} \hat{a}_{1|i} = \frac{1}{2} \hat{a}_{1|1} + \frac{1}{2} \hat{a}_{1|2}. \quad (151)$$

Now $b_{12}$ and $b_{22}$ have much greater value. In iteration $r = 2$ and $r = 3$, the weights $c_{12}$ and $c_{22}$ will be rated even higher, since they will have greater impact on $(151),(152)$.

Finally, the next-layer capsules, $v_j$ of layer $l + 1$ are returned: a short red $v_1$ capsule and long yellow $v_2$ capsule (13). These capsules represent the best over-all predictions made by all the primary capsules in layer $l$.

For each class capsule $v_j$, there are 1152 $16 \times 8$ weight matrices. In addition, the routing-by-agreement algorithm needs to 1152 variables for $c_{i1}$ and another 1152 variables for $b_{ij}$. For 10 class capsules, this sums up to $(1152 \times 16 \times 8 + 1152 \times 2) \times 10$ trainable parameters in the Digit Caps layer - the weight matrices are learned by backpropagation, the routing parameters are learned by routing-by-agreement.

Following up on (3.2.4), the 11520 $\times$ 16D capsules are summed up in the manner of $\hat{a}_{j|1}, \ldots, \hat{a}_{j|1152} \forall j = 0, \ldots, 9$ such that each of the weights $c_{ij}$ is properly set according to the agreement $\hat{a}_{j|i} \cdot \text{squash}(\sum_i c_{ij} \hat{a}_{j|i})$. Routing by agreement delivers not only suitable weights $c_{ij}$, but also returns $v_j \forall j = 0, \ldots, 9$ - the result are 10 $\times$ 16D squashed class capsules $v_j$, $j = 0, \ldots, 9$ for each class digit.

### 3.2.6. Margin Loss

To allow the prediction of multiple classes, we use a separate margin loss, $L_k$ for each digit capsule $k$:

$$L_k = T_k \max(0, m^+ - \|v_k\|^2 \quad + \lambda (1 - T_k) \max(0, \|v_k\| - m^-)^2. \quad (153)$$
with $T_k = 1$ if and only if digit of class $k$ is present and $m^+ = 0.9$, $m^- = 0.1$ and $\lambda = 0.5$.
If digit $k$ is on the input image and corresponding digit capsule $v_k$ ends up having longer length than $m^+$, then we have no margin loss. If digit $k$ is not on the input image and corresponding digit capsule $v_k$ ends up being shorter than $m^-$, then we have no margin loss. With $\lambda$ we can regularize the loss for falsely classified digits.

### 3.2.7. RECONSTRUCTION AS A REGULARIZATION METHOD

As (Hinton et al., 2017) proposes, we can add a reconstruction loss to encourage the digit capsules to construct the instantiation parameters of the input image. During training, all but the true activity vector are blocked out and this activity vector is then used to reconstruct the input image. This reconstruction system is called decoder (Hinton et al., 2017) and consist of 3 fully-connected layers attached to DigiCaps, see figure (14). The true activity vector is fed into the decoder and the final layer consists of 28 * 28 units representing the pixel values of the reconstructed image. We can add the minimization of the squared difference between the input image pixel values and the reconstructed image pixel values to the sum of margin losses, and receive a total loss function. The reconstruction loss is down-scaled by 0.0005 (Hinton et al., 2017), so that the margin loss is the dominating factor. We use backpropagation and a gradient descent optimizer (adam) to minimize the sum of the margin losses $\sum_k L_k$ plus the squared error of the decoder (Hinton et al., 2017).

![Decoder structure to reconstruct image from a digit capsule](https://example.com/decoder.png)

**Figure 14.** Decoder structure to reconstruct image from a digit capsule (Hinton et al., 2017)

### 3.3. CapsNet Training

An image is fed into the Input Layer. Applying a convolutional layer leads to various conv filters that represent certain low level shapes and angles in the image. These filters are stacked to capsule layers and squashed, in which each capsule represents a low level entity of the image.

The activity vectors of each capsule store the instantiation parameters of the entity, the length shows the probability of it existing. The capsules associated to the digit in the image have a striking longer length than the others. In what manner each of entity is part of a whole class i.e. in what spatial relationship each entity is connected to the whole object, is defined by the weight matrix $W_{ij}$. Each capsule is then multiplied with a weight matrix $W_{ij}$. The result is $\hat{u}_{j|i}$, again in which the well predicting capsules have long length. Then routing by agreement sets the weights $c_{ij}$ according to how much $\hat{u}_{j|i}$ agrees with the mean squash($\sum_i c_{ij} \hat{u}_{j|i}$) i.e. the well predicting capsules receive a higher weight and contribute greater than others. The algorithm returns a Digit Caps layer, namely $10 \times 16$ squashed capsules $v_0$, $\ldots$, $v_9$. These high level capsules store high level features of more complex form, concretely, the entire digit. The result of an input image is that the corresponding capsule has long length and the other capsules have short length. The length characterizes the probability of a digit in the image and yields a prediction.

As (Hinton et al., 2017) pointed out, one can analyze individual dimensions of a capsule to learn of what instantiation it is represented by. There are programs, such as (Hsu, November, 2017), that perturb the dimensions of a capsule, making visible which instantiation is associated with the stroke thickness, skew, width etc. This might help improve transparency and accountability, since each capsules instantiation in every capsule layer can be exposed (Shahroudnejad et al., February, 2018).

A tensorflow implementation of CapsNet for MNIST can be found at [https://github.com/MJimitater/CapsNet/blob/master/CapsNet.ipynb](https://github.com/MJimitater/CapsNet/blob/master/CapsNet.ipynb).

### 3.4. Performance and Evaluation

The number of trainable parameters (including bias units) in CapsNet presented for the MNIST dataset in (Hinton et al., 2017), figure (9), are:

| Layer       | Number of Parameters |
|-------------|----------------------|
| Input       | 784                  |
| FC          | 512                  |
| FC          | 1024                 |
| FC          | 784                  |
| Output      | 10                   |

**Figure 15.** (Hinton et al., 2017) Reconstruction examples of MNIST test dataset with 3 routing iterations. $(l,p,r)$ stands for the label, the prediction of the CapsNet and the reconstruction respectively. The two far right columns represent two failure examples. The other columns with correct classification show that the reconstructions perpetuate many details and cancel out noise.
- Convolutional layer:
  \((9 \times 9 + 1) \times 256 = 20,992\)

- PrimaryCaps layer:
  \((32 \times 8) \times ((9 \times 9 + 1) \times 256) = 5,373,952\)

- DigitCaps layer:
  \((1152 \times (16 \times 8) + 1152 \times 2) \times 10 = 1,497,600\)
  - sum (without decoder): 6,892,544

- Decoder: 1\(^{st}\) fully-connected layer:
  \((512 + 1) \times 1024 = 5,373,952\)

- Decoder: 2\(^{nd}\) fully-connected layer:
  \((1024 + 1) \times 784 = 1,497,600\)
  - sum (with decoder): 8,221,456

Other than a slight pixel shift, no data augmentation is performed in Hinton’s paper (Hinton et al., 2017). The baseline to compare is a three convolutional layer CNN with 256, 256, 128 kernels of size 5 × 5 and stride of 1. The last layer is followed by two fully-connected layers of size 328, 192, and finally the output class layer of 10 units, softmax activation, dropout, and adam optimizer (Hinton et al., 2017). The baseline sports 35.4 million trainable parameters, however the paper (Hinton et al., 2017) is not entirely clear on how this number is calculated. The major property is to design the baseline to have best possible performance on MNIST while keeping computation cost on a similar level to CapsNet.

![Figure 16. MNIST average error rate and standard deviation achieved by (Hinton et al., 2017). CapsNet is able to get a lower error rate than the baseline.](image)

The CapsNet’s architecture is the same as the one discussed in (9). As figure (16) of (Hinton et al., 2017) shows, CapsNet scores lower error rates than the baseline. The presented results also show the importance of the reconstruction decoder. It enforces the object’s instantiation to be "encapsulated". The number of routing iterations also has an effect: a major positive effect with decoder, a minor negative effect without decoder.

Eventhough (Hinton et al., 2017) mentions notable performance of CapsNet on the MultiMNIST dataset and on the CIFAR10 dataset, concretely that the error rates match the ones of when standard CNNs first were applied to these datasets. CapsNet is a new artificial neural network architecture and is subject to further improvements to come.

(Mukhometzianov & Carrillo, May, 2018) gives insights to more extensive performance testing of CapsNet. The CapsNet as in figure (9), excluding the decoder subnetworks, was applied to several large datasets, such as Yale Face Database B (University, 2018) with 38 classes, BelgiumTS traffic signs dataset (Timofte et al., 2011) with 62 classes and CIFAR-100 (Krizhevsky, 2009) with 100 classes, among others.

As baseline, different CNNs were used, including modified LeNet (LeCun et al., 1998) and Resnet 50 (He et al., December, 2015).

On these large datasets with many more classes than MNIST, the baseline CNNs outperform CapsNet in both accuracy and average training time using GPU computing power (Mukhometzianov & Carrillo, May, 2018).

![Figure 17. (Mukhometzianov & Carrillo, May, 2018) The baseline CNNs train a lot faster and score higher accuracy than CapsNet almost every time.](image)

Regarding computational cost, CapsNet requires a lot more than CNNs. This is due to the fact that capsules output higher dimensional activity vectors with instantiation parameters than just scalar products, causing greater GPU memory usage.

There are three drawbacks presented by (Mukhometzianov & Carrillo, May, 2018):

First of all: for complex images, even higher dimensional capsules are needed to store all instantiation parameters, 32-D or even 64-D capsules. This would need further powerful GPUs with more memory to fit the capsule sizes.

Second of all: CapsNets achieve good accuracy with small datasets of simple complexity, but fail to hold up with CNNs at large complex datasets. It turns out that on complex datasets CapsNet also need many training images, this equalizes the hope for needing less training images. This hope only holds for simple and less complex images, as researched by (Mobiny & Nguyen, June, 2018). CNNs do possess undesirable properties e.g. lack of equivariance, however, on complex datasets, training with transformed
images can solve this (Wikipedia, 2018c).

Third of all: increasing the size of the input images drastically increases computational cost. Downsizing is only a limited option, since it implies loss of information.

As investigated by (Afshar et al., March, 2018), modified CapsNets perform decently better than baseline CNNs in MRI brain tumor classification, as capsules handle small datasets well, despite the complexity in the images. Equivariance helps CapsNet to classify brain tumors more accurately than CNNs.

We can adhere that the success of CapsNets depend on the specific image application. It is yet to be researched in how far the image complexity matters, as well as the number of classes, the number of capsule layers and other hyperparameters.

As (Mukhometzianov & Carrillo, May, 2018) points out, it becomes clear that CapsNets are at an early stage of development and lack scalability, as CNNs were too when they first were applied (Wikipedia, 2018c). Further improvements are needed to boost accuracy and ameliorate computational cost, like done by (Mobiny & Nguyen, June, 2018). Several modifications have already been introduced by Hinton in an updated paper (Hinton et al., May, 2018) e.g. matrix capsules which group capsules to form part-whole relationships, rather than having weight matrices for this. Further testing and modification of hyperparameters is needed for CapsNet to utilize its full potential.

---

**References**

Afshar, Parnian, Mohammadi, Arash, and Plataniotis, Konstantinos N. Brain tumor type classification via capsule networks. March, 2018. URL https://arxiv.org/pdf/1802.10200.pdf.

Avrutskiy, V.I. Backpropagation generalized for output derivatives. December, 2017. URL https://arxiv.org/pdf/1712.04185.pdf.

Bourdakos, Nick. Understanding capsule networks - ai’s alluring new architecture. February, 2018. URL https://medium.freecodecamp.org/understanding-capsule-networks-ais-alluring-new-architecture-bdb228173ddc.

Brownlee, Jason. How to improve deep learning performance. September, 2016. URL https://machinelearningmastery.com/improve-deep-learning-performance/.

Chevalyre, Yann. Cours data mining / machine learning by yann chevalyre, paris dauphine université. 2017.

Dauphin, Yann N., Pascanu, Razvan, Gulcehre, Caglar, Cho, Kyunghyun, Ganguli, Surya, and Bengio, Yoshua. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. June, 2014. URL https://arxiv.org/pdf/1406.2572.pdf.

Fischetti, Matteo and Jo, Jason. Deep Neural Networks as 0-1 Mixed Integer Linear Programs: A Feasibility Study. December, 2017. URL https://arxiv.org/pdf/1712.06174.pdf.

Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron. Deep Learning. MIT Press, 2016. http://www.deeplearningbook.org.

Géron, Aurélien. Capsule networks (capsnets) – tutorial. 2017. URL https://www.youtube.com/watch?v=pPN8d0E3900&t=434s.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. December, 2015. URL https://arxiv.org/pdf/1512.03385.pdf.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. February, 2015. URL https://arxiv.org/pdf/1502.01852.pdf.

Hinton, Geoffrey, Sabour, Sara, and Frosst, Nicholas. Matrix capsules with em routing. May, 2018. URL https://openreview.net/pdf?id=HJWlfGWRb.
Hinton, Geoffrey E., Sabour, Sara, and Frosst, Nicholas. Dynamic routing between capsules. 2017.

Hsu, Jeremy. A partial implementation of the capsule network from dynamic routing between capsules. November, 2017. URL https://github.com/JeremyCCHsu/CapsNet-tf.

Hui, Jonathan. Understanding dynamic routing between capsules (capsule networks). 2017.

Janocha, Katarzyna and Czarnecki, Wojciech Marian. On loss functions for deep neural networks in classification. February, 2017. URL https://arxiv.org/pdf/1702.05659.pdf.

Kaggle. Cats vs. dogs. 2014. URL https://www.kaggle.com/c/dogs-vs-cats.

Karpathy, Andrej and Johnson, Justin. Cs231n convolutional neural networks for visual recognition. Spring, 2018. URL http://cs231n.github.io/convolutional-networks/.

Krizhevsky, Alex. Learning multiple layers of features from tiny images. 2009. URL https://www.cs.toronto.edu/~kriz/cifar.html.

Lecun, Yann, Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., and Jackel, L.D. Handwritten Digit Recognition with a Back-Propagation Network. 1989. URL http://yann.lecun.com/exdb/publis/pdf/lecun-90c.pdf.

LeCun, Yann, Bottou, Leon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. 1998. URL http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

medium. relu operation. 2017. URL https://medium.com/data-science-group-iitr/building-a-convolutional-neural-network-in-python-with-tensorflow-d251c3ca8117.

Mobiny, Aryan and Nguyen, Hien Van. Fast capsnet for lung cancer screening. June, 2018. URL https://arxiv.org/pdf/1806.07416.pdf.

Mukhometzianov, Rinat and Carrillo, Juan. Capsnet comparative performance evaluation for image classification. May, 2018. URL https://arxiv.org/pdf/1805.11195.pdf.

Pechyonkin, Max. Understanding hinton’s capsule networks. part iv: Capsnet architecture. February, 2018. URL https://medium.com/@pechyonkin/part-iv-capsnet-architecture-6a64422f7dce.database#Performance.

Peng, Min, Wang, Chongyang, Chen, Tong, and Liu, Guangyuan. Nirfacenet: A convolutional neural network for near-infrared face identification. 2016. URL http://www.mdpi.com/2078-2489/7/4/61.

Picasso, Pablo. Portrait de femme au col d’hermine. 1923. URL https://www.wikiart.org/en/pablo-picasso/untitled-1937-8.

quora. convolutional operation. 2016. URL https://www.quora.com/How-common-is-it-for-neural-networks-to-be-represented-by-3rd-order-tensors-or-greater.

quora. max pooling. 2017. URL https://www.quora.com/What-is-max-pooling-in-convolutional-neural-networks.

Ross, Mike. A visual representation of capsule network computations. November, 2017. URL https://medium.com/@mike_ross/a-visual-representation-of-capsule-network-computations-83767d79e737.

Ruder, Sebastian. An overview of gradient descent optimization algorithms. June, 2017. URL https://arxiv.org/pdf/1609.04747.pdf.

Shahroudnejad, Atefeh, Mohammad, Arash, and Platanitis, Konstantinos N. Improved explainability of capsule networks: Relevance path by agreement. February, 2018. URL https://arxiv.org/pdf/1802.10204.pdf.

Timofte, Radu, Zimmermann, Karel, and van Gool, Luc. Multi-view traffic sign detection, recognition, and 3D localisation. Journal of Machine Vision and Applications, Springer-Verlag, 2011. https://btsd.ethz.ch/shareddata/.

University, Yale. Yale face database b. 2018. URL http://cvc.cs.yale.edu/cvc/projects/yalefaces/yalefaces.html.

Wikipedia. Receptive field. 2018a. URL https://en.wikipedia.org/wiki/Receptive_field#Visual_system.

Wikipedia. Dot product. 2018b. URL https://en.wikipedia.org/wiki/Dot_product.

Wikipedia. Convolutional neural network. 2018c. URL https://en.wikipedia.org/wiki/Convolutional_neural_network.

Wikipedia. Mnist database. 2018d. URL https://en.wikipedia.org/wiki/MNIST_database#Performance.
Wikipedia. Softmax function. 2018e. URL https://en.wikipedia.org/wiki/Softmax_function.

Zhang, Zhifei. Derivation of backpropagation in convolutional neural network (cnn). 2016. URL https://pdfs.semanticscholar.org/5d79/11c93ddcb34cac088d99bd0cae9124e5dcd1.pdf.