1 Introduction

Among a range of other tasks, image recognition has been a domain dominated by feed forward Convolutional Neural Network architectures. Strong evidence suggests, that the human brains ability to rapidly recognize objects under appearance variation is solved in the brain via a succession of largely feedforward computations. [DZR12, 2012] Recent success of feedforward networks has only supported this hypothesis [KSH12]. However, evidence also suggests, that after the initial recognition process further processing takes place [SYUK99]. This delayed processing might indicate recurrent computations which I want to investigate in this work. Using a range of convolutional architectures, I try to systematically measure the impact of recursion on image recognition tasks.

2 Background

2.1 Supervised Learning

The common ground for all further aspects in this work will be supervised learning, which is most generally speaking simply the task of deriving a function from data. This function should be able to map given inputs or data points to some sort of output. In the supervised setting, data is given with an associative label such, that the inferred function is able to map said input data to the correct labels, without explicitly looking at the given labels. The learning algorithm, when deriving said function, therefore has to use patterns and structure among the data points to learn something about the quality of a given data point. Outputs are generally either of a quantitative type e.g. temperature measurements or changes in stock price or of a qualitative type like distinctive types of flowers. Tasks involving the former kinds of outputs are generally referred to as regression problems whereas the tasks involving the latter are generally called classification problems.
We can roughly articulate the supervised learning problem as follows: for a given input X, an associated label Y and a function $\mathbb{F}(X) = \hat{Y}$, $\mathbb{F}$ makes a good prediction, when $\hat{Y}$ is close to Y. Proximity, in this case, must be defined and can vary depending on the type of data which is present. For a regression problem of e.g. predicting tomorrows temperature, the prediction is better, the more similar the predicted value $\hat{Y}$ and tomorrows true temperature are, measured by the absolute distance of $|Y - \hat{Y}|$. X in this case could be a vector of meteorologic measurements which are regarded highly predictive of short term weather prognosis.

2.1.1 Linear Models

Linear Regression has been a staple in statistics and machine learning and remains a very important and widely used tool [HTF01]. As it serves as the basis for a wide range

of more complex machine learning algorithms like logistic regression and in some sense even Deep Neural Networks, it serves as a good introduction. The linear model

$$\mathbb{F}(X) = \beta + \sum_{i=1}^{|X|} w_i x_i = \hat{y}$$

predicts Y by using a linear combination of all input variables x_i using weights w_i. β is the intercept of the linear decision boundary, often referred to as the bias. Since in this case we are modeling a scalar value, $\hat{y}$ is a single value, but can also be N-vector if we're predicting values of higher dimensionality. To get an idea how good our model is performing, we first need to come up with a way of measuring its prediction quality. The sum of squares is a widely used method and sums each squared difference between any predicted value $\hat{y}_i$ and its corresponding true value y_i. We define the sum of squared errors as E, a function of our parameters $\mathbb{W}$ which we then can minimize. Since it is a quadratic function, its minimum always exists but does not have to be unique.

$$E(\mathbb{W}) = \frac{1}{2}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 = \frac{1}{2}\sum_{i=1}^{N}(\sum_{j=1}^{|X|}(w_i x_i + \beta) - y_i)^2$$

Minimizing E($\mathbb{W}$) with respect to $\mathbb{W}$ yields:

$$\frac{\delta E}{\delta \mathbb{W}} = \left(\frac{\delta E}{\delta w_1}, ..., \frac{\delta E}{\delta w_n}\right) = \left(\sum_{i=1}^{N}(\hat{y}_i - y_i)x_1, ..., \sum_{i=1}^{N}(\hat{y}_i - y_i)x_n\right)$$

2.2 Neuronale Netze

Neuronale Netze sind biologisch inspirierte Programmierparadigma, welche es erlauben, iterativ vom Betrachten von Daten zu lernen. Dazu benötigen sie meist kein domänenspezifisches Wissen, sondern "lernen" Strukturen in den betrachteten Daten selbst zu erkennen. Die Grundeinheit solcher Netzwerke sind dabei künstliche Neuronen, die in Schichten angeordnet sind. Ein Neuron erhält eine Eingabe und propagiert sie über eine Verbindung (Synapse) an andere Neuronen weiter. Typischerweise senden Neuronen ihre Eingabe entlang eines Pfades von der Eingabe zur Ausgabe und haben Gewichte, die bestimmen, wie viel des erhaltenen Signals an folgende Neuronen weitergegeben wird.

Die simpelste Form eines Neuronalen Netzes ist ein einfaches Perzeptron, welches 1958 von Frank Rosenblatt in [Ros58] vorgestellt wurde. Dieses basiert auf einer McCulloch-Pitts-Zelle, einem rudimentären Modell einer Nervenzelle nach [MP43], enthält mehrere Eingaben und produziert eine einzige Ausgabe. Perzeptrons können prinzipiell aus mehreren Schichten solcher Zellen bestehen, die in Reihe geschaltet sind. Dabei gibt eine Zelle ihren berechneten Output an ein Neuron der nächsten Schichten weiter. Solche

Netzwerke heiSSen Multi-Layer-Perzeptron, im weiteren beschäftigen wir uns jedoch mit einer einzigen Zelle. Das einfache, in Abbildung 1 dargestellte Perzeptron erhält n

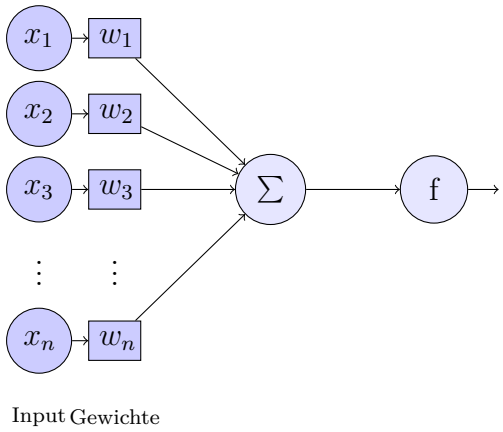


Abbildung 1: Ein einfaches Perzeptron mit n Eingängen

Eingänge $x_1, ..., x_n \in \mathbb{R}$, die jeweils mit korrespondierenden Gewichten $w_1, ..., w_n \in \mathbb{R}$ multipliziert werden. Die Summer der Gewichteten Eingänge wird dann als Argument der Aktivierungsfunktion verwendet. Die Ausgabe des Perzeptron lässt sich als eine Funktion seiner Eingaben wie folgt beschreiben:

$$\mathbb{P}(\boldsymbol{x}, \boldsymbol{w}) = f(\sum_{i=1}^{n} w_i x_i)$$

mit der Aktivierungsfunktion

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

womit die Ausgabe als Zugehörigkeit zu einer klasse, entweder 0 oder 1 zu interpretieren ist.

2.3 Convolutional Neural Networks

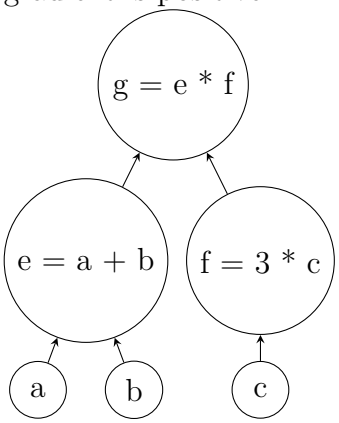
2.4 Recurrent Neural Networks

2.5 Unrolling Recurrent Netowrks

2.6 Backpropagation

Backpropagation is a widely used technique to train neural networks. It is, in fact, the algorithm that made training deep neural networks tractable in the first place. While being originally introduced in the 1970's it has not been adapted until David Rumelheart, Geoffrey Hinton and Ronald Williams drew a lot of attention to it in their

famous 1986 Paper [RHW+88]. At the core of Backpropagation stand partial derivatives like $\frac{\delta C}{\delta w}$ of a cost function C with respect to a weight w. This gradient expresses, at what rate C changes if we tune w. Knowing how the error of the network behaves when changing a parameter can be very helpful, since we then can adjust it in a way, such that our total error decreases. In order to minimize our cost function we therefore have to compute the partial derivatives of the networks cost function with regard to every variable in the network i.e. every weight and bias. We then use those gradients to move "downwards" on our cost function i.e. adding a small positive value to our weight or bias, if the gradient is negative and vice versa adding a small negative value, if the gradient is positive.



3 Materials and Methods

3.1 Generatives Modell für Stimuli

Um die Effekte von Rekursion auf das klassifizieren von okkludierten Objekten zu untersuchen, wurde ein möglichst einfaches Grundproblem betrachtet. Die Klassifizierung von Ziffern ist ein gut untersuchtes Machine Learning Problem, in dem übermenschliche Performance erreicht werden kann. Allgemein gilt das sehr ähnliche MNIST-Datenset gemeinhin als das "Hello world!" des Machine Learning. Durch die Einfachheit der Aufgabe, können die Auswirkungen von Rekursion isoliert von anderen Herausforderungen betrachtet werden.

3.2 Models

Um die Wirkungskraft von Rekursion zu untersuchen, wurden verschiedene CNNs mit variierenden Graden an Rekursion benutzt und systematisch miteinander verglichen. Die Nomenklatur richtet sich am Paper von Spoerer und Kriegeskorte aus. Als Grundlage dient eine Standard feed-forward Architektur (B) mit reinen 'bottom-up' Verbindungen. Da diese jedoch in der Anzahl der Parameter und der Anzahl der durchgeführten Konvolutionen gegenüber seinen Rekursiven Varianten unterlegen ist, wurden

zum Vergleich zusätzlich Modelle entwickelt, die in den entsprechenden Domänen angepasst wurden. Einerseits wurde die GröSSe der Konvolutionskernel angepasst und somit die GröSSe der erlernbaren Features. Andererseits wurde die Anzahl der Konvolutionen erhöht, die Gewichte in den zusätzlichen Konvolutionen jedoch mit den anderen Konvolutionen geteilt. Somit kann die Anzahl der Faltungsoperationen vergleichbar gemacht werden, ohne die freien Parameter zu erhöhen. Die Archtitekturen werden im Folgenden BK respektive BKC genannt.

3.2.1 Implementation

Apart from BKC, all models consist of two convolution layers. All bottom up convolutions are implemented as standard convolutions with 1x1 stride and zero padding, which leads to the output pictures being the same size as the input. The output of the convolution is then fed into a parameterized version of the Rectified Linear Unit activation function (PReLU). PReLU works as a generalized form of the ReLU activation function as it controls the output for negative values with a slope which can be learned.

$$f(x_i) = \begin{cases} x_i, & if\ y_i > 0 \\ a_i y_i, & if\ y_i \leq 0 \end{cases}$$

If the slope parameter is zero, PReLU results in standard ReLU. If the parameter is a small positive number, PReLU equals Leaky ReLU. The output of the PReLU activation is then normalized with local response normalization (LRN), which tries to account for lateral inhibition which was found in brains. LRN therefore dampens responses which are uniformly large in any given local neighborhood and strengthens activations which are relatively larger than their surroundings and so imposes sparse activations. After normalization, the image is fed into a max-pooling layer with 2x2 stride and a pooling window of size 2x2, thus reducing the image size by half in each dimension. This whole process is repeated for each of the convolutions. After the second pooling, the image therefore is 8x8 in size and is flattened to a 1x64 vector before being fed into a readout layer, which maps the input to 1x10 vector. The sigmoid function is then applied to the final output, yielding values from 0 to 1 which can be interpreted as the probability that each of the responding targets is present in the given input picture.

3.2.2 Recurrent Convolution Layers

The heart of this work are recurrent convolution layers (RCL) whose effect on occluded image recognition is to be invetigated. We denote the input of layer l at timestep t with