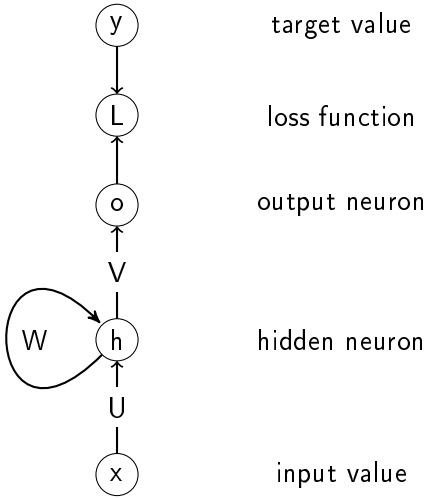
Technische Universität Berlin

Recurrent Neural Networks

Ansgar Rössig, Arne Schmidt

Agenda

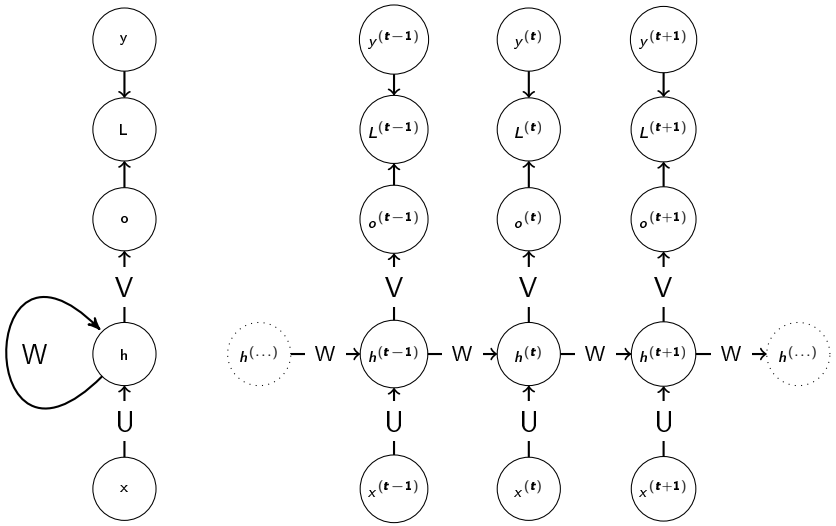
Recurrent Neural Network Example



Recurrent Neural Network Example

- RNNs are good to work with input sequences of unknown / variable length
- parameters U, V, W are shared accross multiple time steps
- theoretically, the number of time steps is not bounded
- in practice, a fixed number τ of time steps is used for all computations

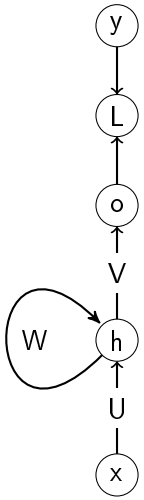
Recurrent Neural Network Example



Notes

- the unfolded computational graph shows how forward and back-propagation can be applied
- it is assumed that the same parameters are relevant at all time steps

Forward Propagation



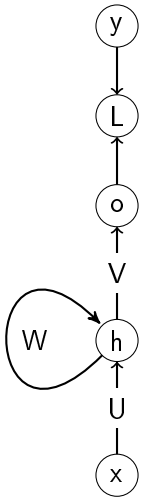
- start with initial state $h^{(0)}$
- then for $t = 1, \dots, \tau$:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$
$$h^{(t)} = \sigma(a^{(t)})$$
$$o^{(t)} = c + Vh^{(t)}$$
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

Notes

- for forward propagation an initial state $h^{(0)}$ is fixed
- then the inputs can be propagated forward through the unfolded computational graph as in the case of feedforward networks
- this can be continued for arbitrarily many time steps, e.g. until the end of the input sequence is reached

Back-propagation and loss function

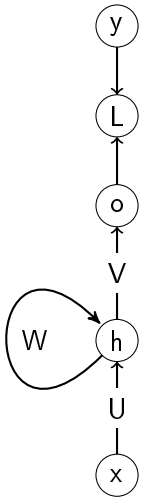


- let $L^{(t)}$ be the negative log-likelihood of $y^{(t)}$ given $x^{(1)}, \ldots, x^{(t)}$
- sum the loss over all time steps

$$
\begin{aligned}
&L(\{x^{(1)}, \ldots, x^{(\tau)}\}, \{y^{(1)}, \ldots, y^{(\tau)}\}) \\
&= \sum_t L^{(t)} \\
&= -\sum_t \log p_{model}(y^{(t)} \mid \{x^{(1)}, \ldots, x^{(t)}\}) \\
&= -\sum_t \log \hat{y}^{(t)}
\end{aligned}
$$

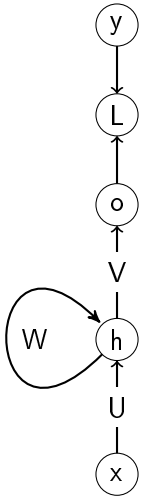
- use back-propagation through time (BPTT)

Back-propagation and loss function



- back-propagation through time (BPTT) is simply the normal back-propagation algorithm applied to the unfolded computational graph
- BPTT is used to minimize the sum of the loss values at all time steps

Back-propagation and loss function



- parameters are shared across all time steps
- therefore, the gradient is the sum over all time steps
- we need to introduce copies $W^{(t)}$ of W for each time step t

$$\nabla_W L = \sum_t \sum_i \left(\frac{\delta L}{\delta h_i^{(t)}} \right) \nabla_{W^{(t)}} h_i^{(t)}$$

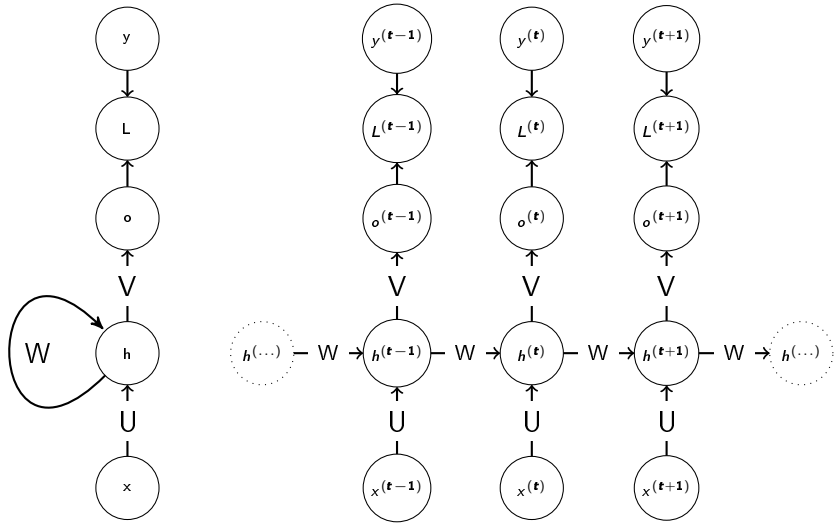
Reminder

$$
\begin{aligned}
a^{(t)} &= b + W h^{(t-1)} + U x^{(t)} \\
h^{(t)} &= \sigma(a^{(t)}) \\
o^{(t)} &= c + V h^{(t)} \\
\hat{y}^{(t)} &= \text{softmax}(o^{(t)})
\end{aligned}
$$

Notes

- there is only one parameter matrix W for all time steps
- though the computation of the gradient of $W^{(t)}$ depends on $W^{(t+1)}, W^{(t+2)}, \ldots$
- therefore the copies of W are introduced and the final gradient of W is computed as the sum over all $W^{(t)}$ gradients

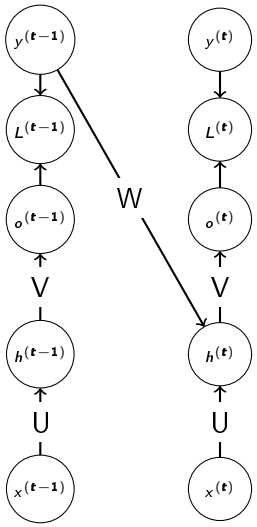
Back-propagation



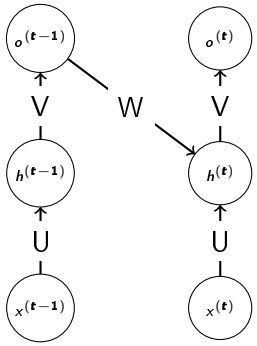
Notes

- back-propagation is applied to this graph
- due to the connections from $h^{(t)}$ to $h^{(t+1)}$ for all t, the gradients have to be computed for one time step after the other, starting from the last one
- this inhibits parallelization

Teacher forcing



Training Time vs. Test Time



Notes

- If connections exist only between the output neuron at one time step and the hidden neuron of the next time step, parallelization is possible.
- During training the value of $y^{(t-1)}$ can be used instead of $o^{(t-1)}$ and thus the gradients for each time step can be computed in parallel, this is called teacher forcing.
- During test/application the output value $o^{(t-1)}$ is used. However, if training of the network is continued during the application, the values of $o^{(t)}$ might become quite different than the ones of $y^{(t)}$. One possibility to take care of this is to train partly with $y^{(t)}$ and partly with $o^{(t)}$ values.
- Such networks are less powerful than RNNs with connections between hidden units but easier to train.

Computational Power

Computational Power of RNN

Siegelmann and Sontag (1995) [?]

A universal Turing Machine can be simulated by a RNN of at most 886 neurons.

Church–Turing thesis

Turing machine

- infinite tape
- read / write head
- finite state register
- finite table of instructions (program)

Church-Turing thesis

The class of all Turing computable functions is the same as the class of all intuitively computable functions.

Church–Turing thesis

p-stack Turing machine

- p stacks
- p read / write heads
- push / pop / don't change stack
- state register
- table of instructions
- input and output: binary sequence on stack 1

Church-Turing thesis

The class of all Turing computable functions is the same as the class of all intuitively computable functions.

Key idea

- Three neurons per stack:
  - stack enconding
  - top element
  - stack.isEmpty()
- some other neurons which execute the "Turing machine"

Notes

- p-stack and "normal" Turing machines are equivalent up to polynomially bounded differences in computation time
- the number of further neurons depends on the number of states and stacks
- due to the Church-Turing thesis RNN can compute all intuitively computable functions (e.g. from $\mathbb{N}$ to $\mathbb{N}$)

Computational Power of RNN

$$\sigma(x) := \begin{cases} 0, & \text{if } 0 > x \\ x, & \text{if } 0 \leq x \leq 1 \\ 1, & \text{if } 1 < x \end{cases}$$

Lemma: Siegelmann and Sontag (1993) [?]

The computational power of a RNN which performs an update on neurons $x \in \mathbb{R}^n$ and inputs $u \in \mathbb{R}^m$ using a function $f = \psi \circ \phi$ is equivalent to a RNN which uses the activation function σ (up to polynomial differences in time).
$\phi : \mathbb{R}^{n+m} \to \mathbb{R}^n$ is a polynomial in $n + m$ variables
$\psi : \mathbb{R}^n \to \mathbb{R}^n$ has bounded range and is locally Lipschitz

Notes

- This lemma implies that RNNs with activation functions as sigmoid or ReLU are not computationally stronger than RNNs with the activation function σ defined here.
- σ will be used throughout the proof ideas which we present
- We show some key ideas, the entire construction is a bit more complicated and can be found in the paper.

Stack enconding

Represent a stack as binary string $\omega = \omega_1\omega_2 \dots \omega_n$. Encode this as

$$c(\omega) = \sum_{i=1}^{n} \frac{2\omega_i + 1}{4^i} \qquad \in [0, 1[.$$

- For the empty stack, set $c(\omega) = 0$.
- $\omega_1 = 1 \;\Leftrightarrow\; c(\omega) \geq \frac{3}{4}$
- $\omega_1 = 0 \;\Leftrightarrow\; c(\omega) \in [\frac{1}{4}, \frac{1}{2}[$
- e.g. for $\omega = 1011$:

$$c(1011) = 0.3133_4 = \frac{223}{256} \geq \frac{192}{256} = \frac{3}{4}$$

Stack operations

Let $q = c(\omega)$ be a stack enconding.

$$4q - 2 \begin{cases} \geq 1, & \text{if } q \geq \frac{3}{4} \\ \leq 0, & \text{if } q \leq \frac{1}{2} \end{cases}$$

Reading value of the top element:

$$\mathrm{top}(q) = \sigma(4q - 2)$$

Stack operations

- Pushing 0 to stack $\omega = 1011$ gives $\tilde{\omega} = 01011$.
- $q = c(\omega) = 0.3133_4$
- $\tilde{q} = c(\tilde{\omega}) = 0.13133_4$

$$\tilde{q} = \frac{q}{4} + \frac{1}{4} = \sigma(\frac{q}{4} + \frac{1}{4})$$

- Pushing 1 to stack $\omega = 1011$ gives $\tilde{\omega} = 11011$.
- $q = c(\omega) = 0.3133_4$
- $\tilde{q} = c(\tilde{\omega}) = 0.33133_4$

$$\tilde{q} = \frac{q}{4} + \frac{3}{4} = \sigma(\frac{q}{4} + \frac{3}{4})$$

Notes

- The definition of $c(\omega)$ ensures, that there is a difference of at least $\frac{1}{4}$ in the stack enconding between stacks with a 1 on top and stacks with a 0 on top.
- 0.3133_4 etc. are numbers in the 4-adic system
- σ is as defined earlier
- $\frac{q}{4} + \frac{1}{4} \in [0, 1]$, $\frac{q}{4} + \frac{3}{4} \in [0, 1]$ and therefore σ applied to these values is the identity function

Stack operations

- Popping from stack $\omega = 1011$ gives $\tilde{\omega} = 011$.
- $q = c(\omega) = 0.3133_4$
- $\tilde{q} = c(\tilde{\omega}) = 0.133_4$

$$\tilde{q} = 4q - (2 \cdot \text{top}(q) + 1)$$

Stack operations

- Stack is empty $\Leftrightarrow q = 0$
- Stack is non-empty $\Leftrightarrow q \geq 0.1_4 = \frac{1}{4}$
- use

$$\text{empty}(q) = \sigma(4q) = \begin{cases} 0, & \text{if } q = 0 \\ 1, & \text{if } q \geq 0.1_4 \end{cases}$$

Universal Turing machine simulation

- create three neurons per stack to hold
  - q
  - top(q)
  - empty(q)
- some more neurons for states and computation
- can be used to simulate universal turing machine with at most 886 neurons
- in 2015, Carmantini *et al.* [?] constructed a RNN simulating a universal Turing machine with only 259 neurons

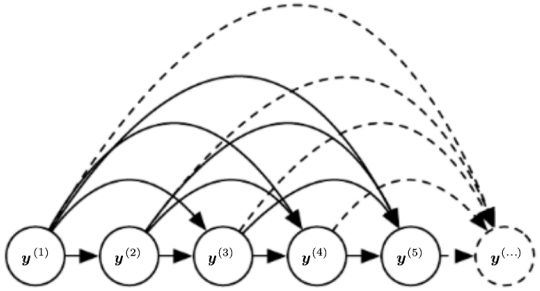
Notes

- One can construct a universal Turing machine with one tape, four letters (values on the tape) and seven states. With this an upper bound on the number of neurons that are necessary can be derived.
- The approach of Carmantini *et al.* [?] uses more complicated concepts to proof the result, therefore we present the ideas of Siegelmann and Sontag which give a better intuition.
- Hence, assuming an unbounded number of possible time steps τ, a RNN can compute any Turing computable function from $\mathbb{N}$ to $\mathbb{N}$.

Variants of RNNs

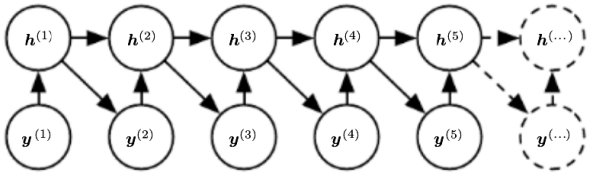
Directed graphical models

- directed acyclic graph
- nodes represent random variables
- edges show dependencies of variables



Directed graphical models with RNN

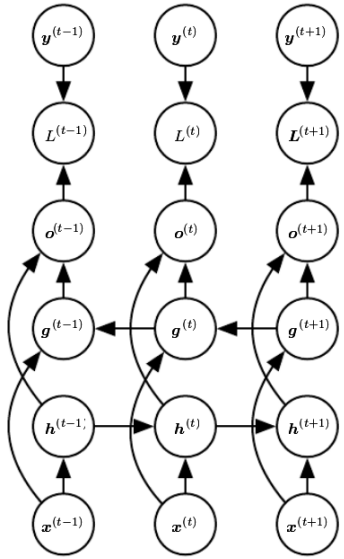
- RNN can represent the joint probability
- introduction of hidden units provides efficient parametrization
- number of parameters is independent of sequence length
- sparse representation, compared e.g. to table



Notes

- Direceted graphical models represent random variables and their influences/independence, similar to Markov chains (or Markov Random Fields).
- A RNN can be used to give an efficient parametrization of these.
- No input is used in this case, the hidden neurons $h^{(t)}$ are random variables which are deterministic given the values of their parents.

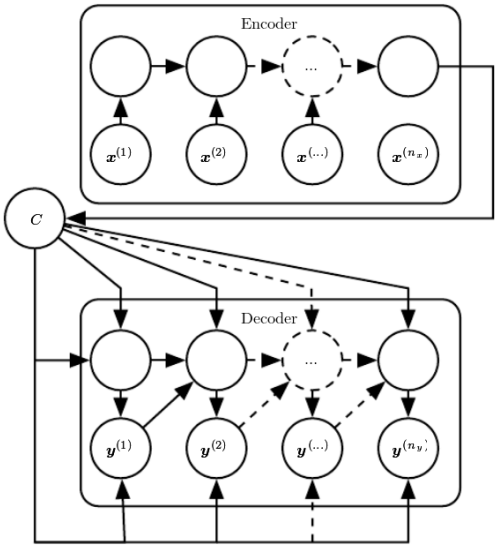
Bidirectional RNNs



Notes

- In many applications an output depends not only on earlier but also on later inputs. E.g. in speech recognition the interpretation of the current sound may depend on sounds before and after this.
- The example architecture shows a RNN whose outputs depend on prior and later inputs.
- Such RNNs have similarities to convolutional neural networks, only that here a "window" of flexible rather than fixed size appears around a time step.
- Forward and back-propagation can be performed as usual, only for the parameters associated with $g^{(t)}$ the gradient computation must start from the first time step rather than the last one.

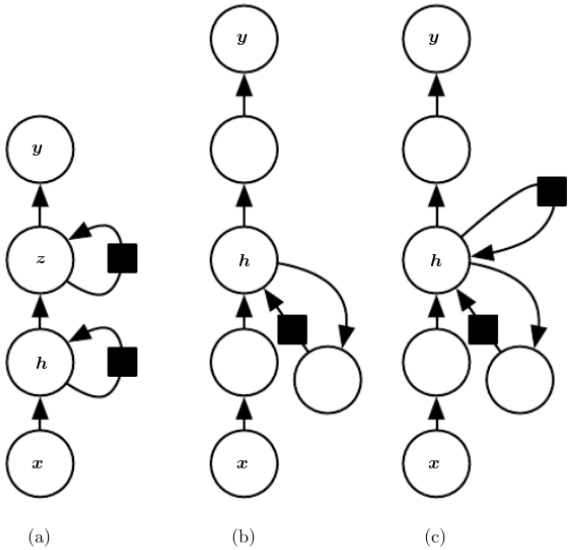
Encoder-Decoder Architectures



Notes

- Encoder-Decoder architectures allow to produce an output sequence whose length is independent of the length of the input sequence. Machine translation is a taks which requires this setting.
- The vector c contains all the information of the input sequence. This requires a high-dimension or input information may be lost. Some authors propose to use a variable length sequence at this point instead.

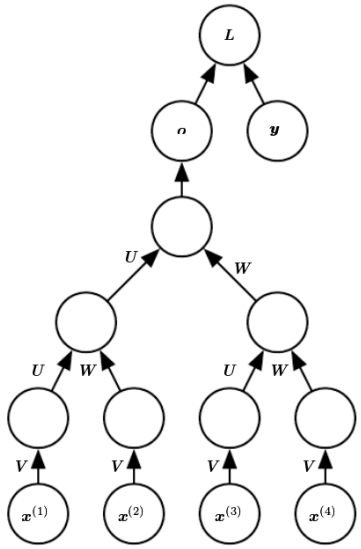
Deep Recurrent Networks



Notes

- Black boxes in the figure symbolize that the information is passed forward/backward in the next time step.
- Various configurations of recurrent layers are possible. Skip connections as in (c) may be introduced to shorten the paths for the gradient backward pass, which can help to facilitate the training of the RNN.

Recursive Neural Networks



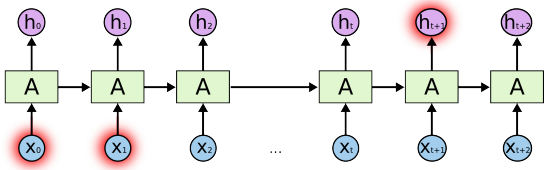
Notes

- RNN can also be generalized into recursive neural networks. The structure can then have the form of a tree which gives various possibilities to process the input information of variable length.

Long Term Dependencies

Problem of Long Term Dependencies

- in RNN's gradients are propagated over many stages, which can lead to vanishing or exploding gradients
- the influence of an output vanishes over time
- challenge: How can we make information flow over long distances?



Techniques for Long Term Dependencies

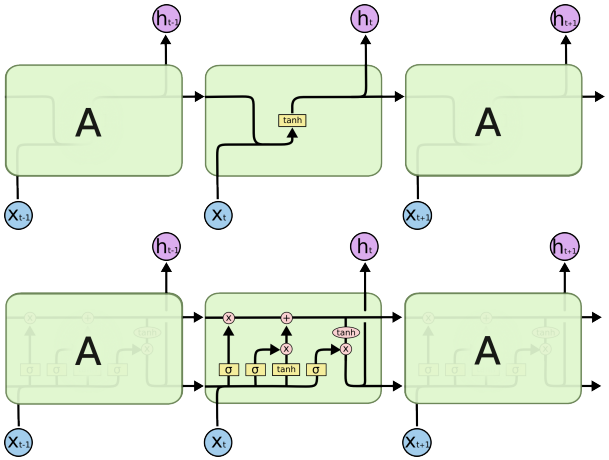
- Echo state networks
- Gated RNN's - LSTM

LSTM - Long Term Short Memory

Idea:

- have one state that flows over time and can be manipulated
- gates control the information that is passed to the state
- units are not connected through fixed weights but with dynamical connections

Difference between normal RNN and LSTM

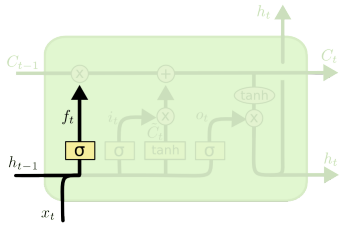


Source: (Great Blog!)
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

There are 3 gates at a standard LSTM:

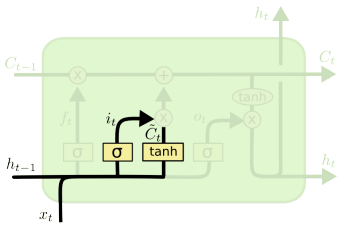
- Forget-Gate: gate that can delete information of the state
- Input-Gate: decides, which values of the state are going to be updated with which information
- Output-Gate: creates output from the state and the input

The Forget-Gate



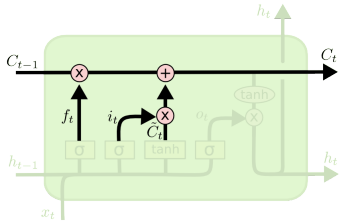
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

The Input-Gate



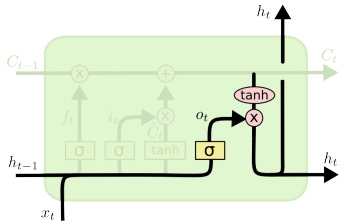
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

The State-Update



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

The Output-Gate



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Programming Project

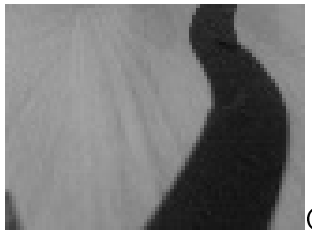
The Real World Problem

Semantic Segmentation

Source: University of Cambridge

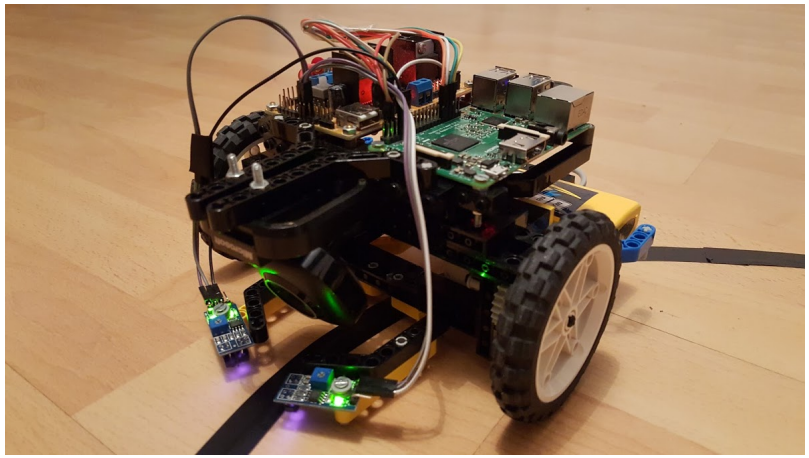
Each pixel from a camera image should be annotated with a class (tree",car",ßtreet",..). This technique is neccessary if it is not only important WHAT objects you can see in an image but also WHERE they are, for example in the case of autonomous cars.

My Toy-Example


Greyscale Image

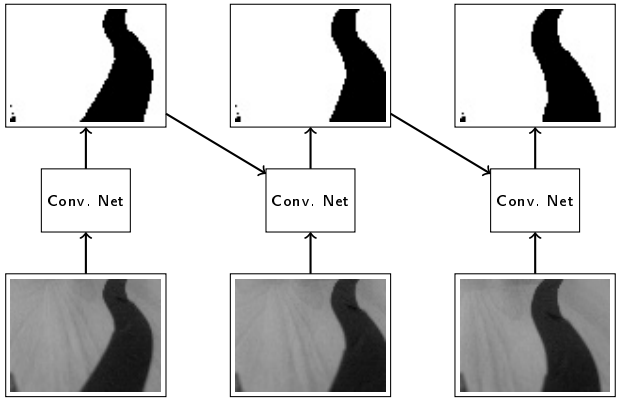

Segmentation: Road- Not Road

The robot car

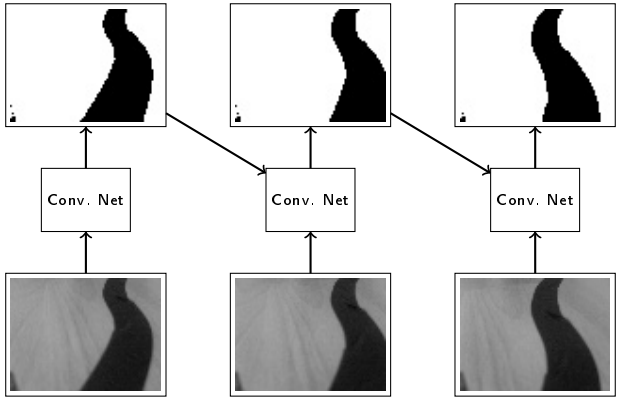


Light-sensors to follow the black line, webcam to take pictures.

Idea



Idea

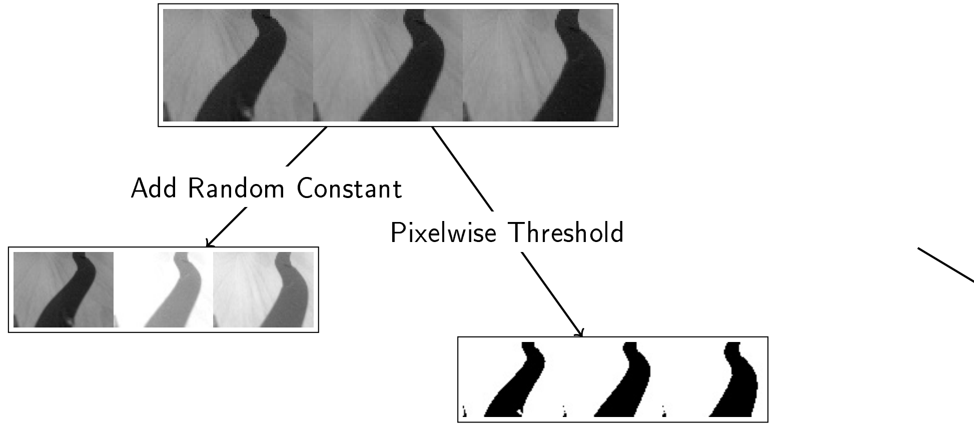


The aim is to construct an RNN which obtains the previous segmentation and moving direction as well as the actual image as an input. This network is than able to interpret an image in the context of a sequence of images.

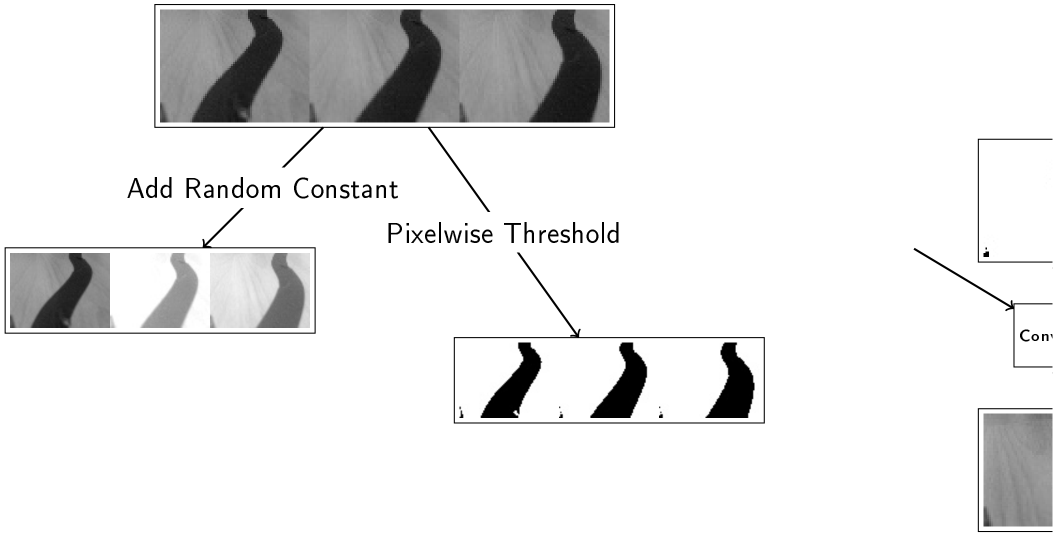
Challenges

- get a lot of training data
- generate the ground trouth automatically
- make the task neither too complicated nor too easy

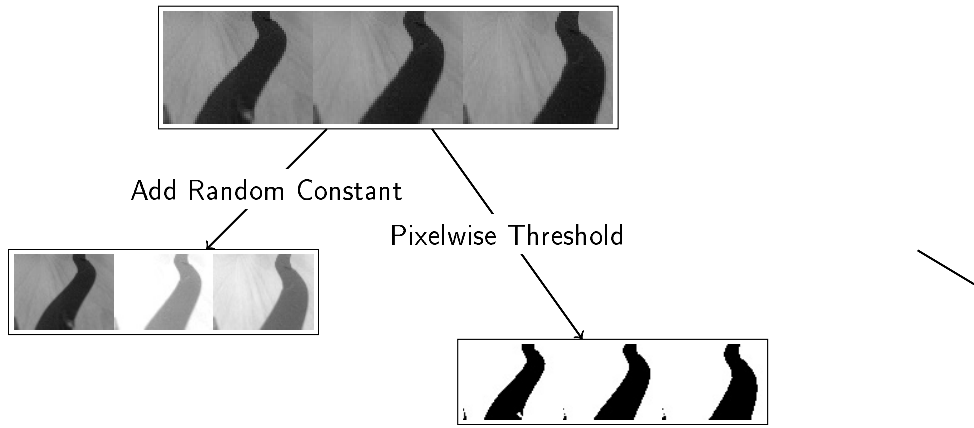
Generation of data and ground truth



Generation of data and ground truth



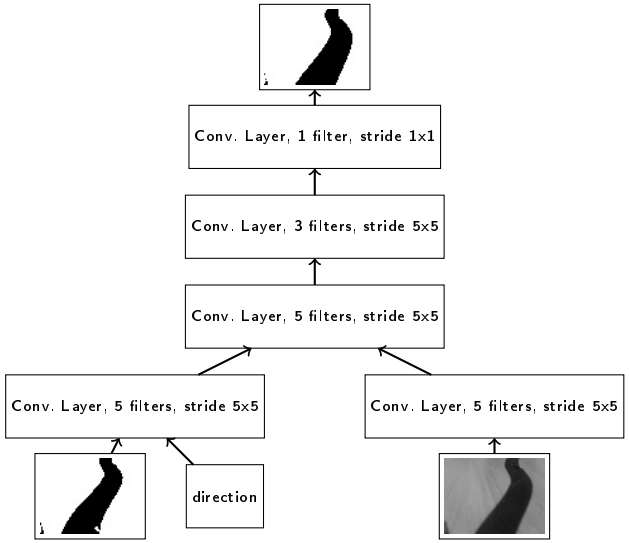
Generation of data and ground truth



From greyscale images the trainingset was generated:

- the input images are the pictures distorted with a random constant value

Architecture of the RNN



Programming: Tensorflow (ML-library for python)

- really easy to use
- computational efficient
- builds 'graph' of the neural network which consists of optimized operations

Code

All available under:
`https://github.com/arneschmidt/RNN_project/`
Feel free to change the RNN and try things out!

Code

All available under:
`https://github.com/arneschmidt/RNN_project/`
Feel free to change the RNN and try things out!

Ansgar Rössig, Arne Schmidt Programming Project TU Berlin

Code

All available under:
`https://github.com/arneschmidt/RNN_project/`
Feel free to change the RNN and try things out!

Code

All available under:
`https://github.com/arneschmidt/RNN_project/`
Feel free to change the RNN and try things out! Ideas to try out:

- different architecture (change number of layers, stride, filters etc.)
- use an LSTM for the prediction of the direction
- use an LSTM for the segmentation

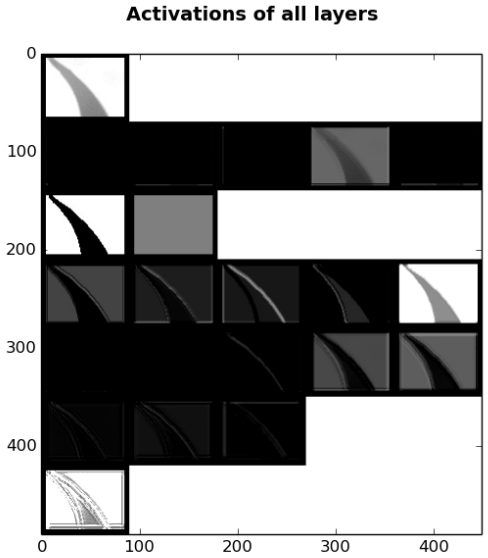
First results

Comparison of RNN with normal CNN:
Pixelwise accuracy:
Image only: 92.45
Image+previous segmentation (RNN): 98.64
(same architecture and 10.000 training steps)

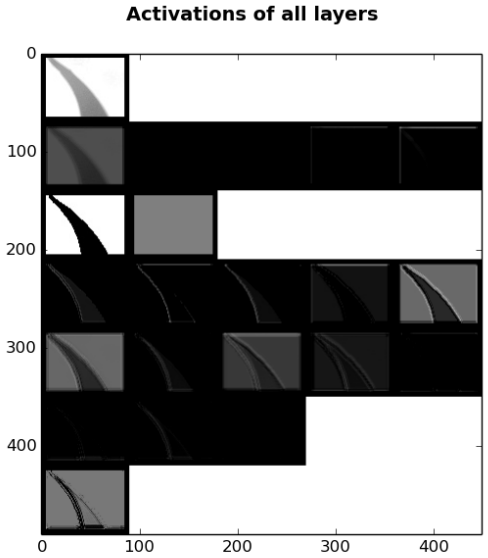
Training procedure

1. 5000 iterations with teacher forcing, batches of 10 images
2. 5000 iterations with sequences of 5 images
3. 5000 iterations with sequences of 10 images

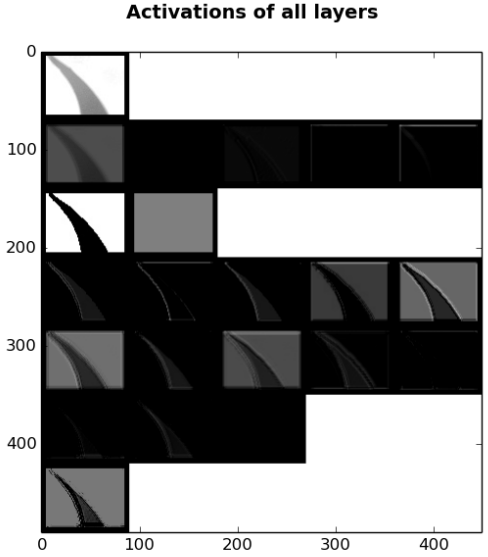
After 0 iterations



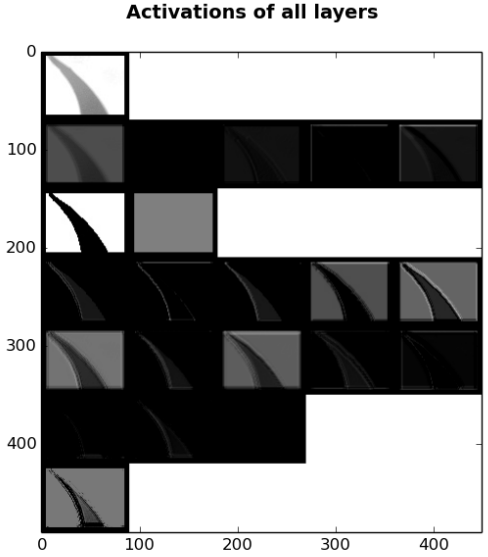
After 100 iterations



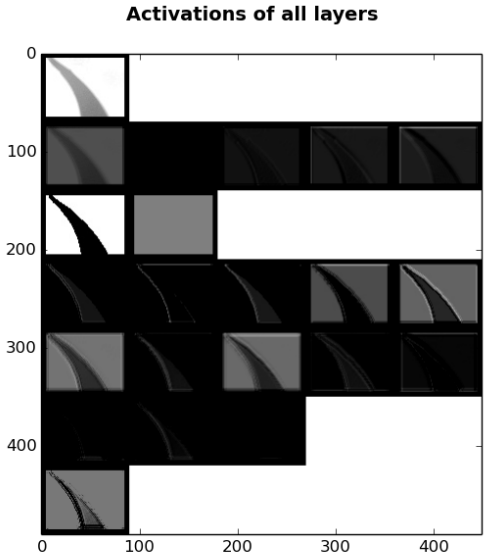
After 200 iterations



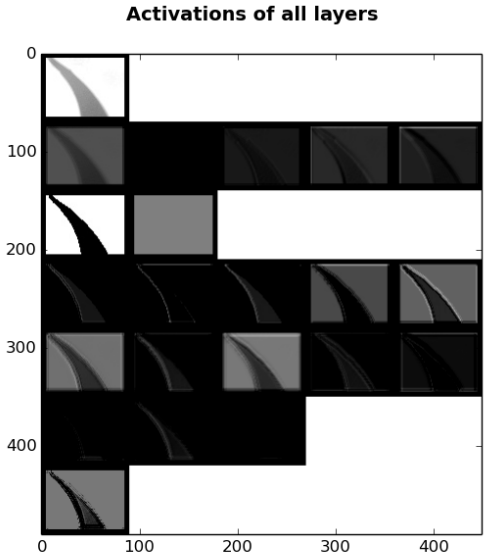
After 300 iterations



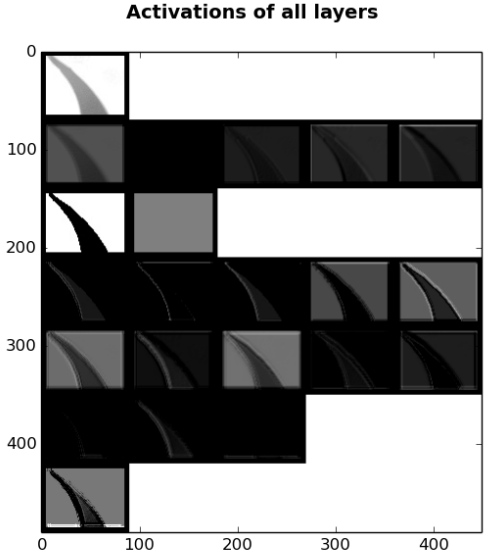
After 400 iterations



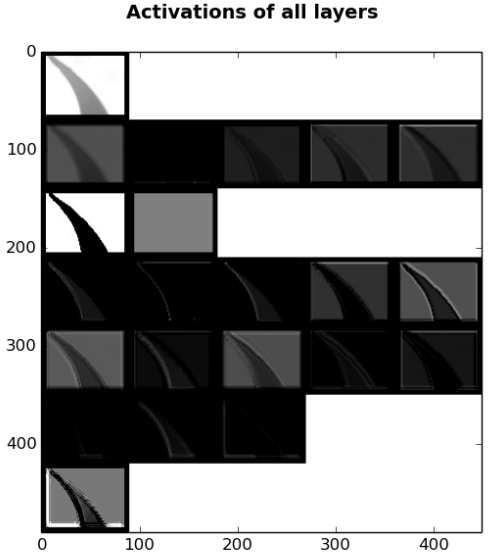
After 500 iterations



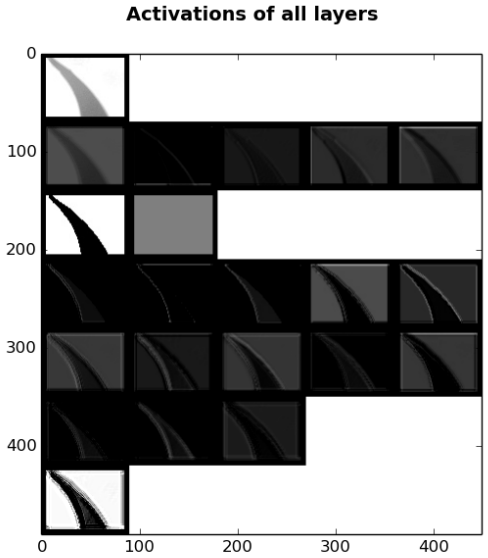
After 600 iterations



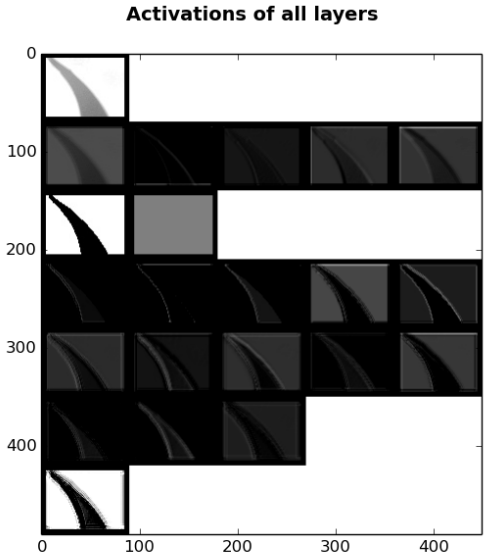
After 700 iterations



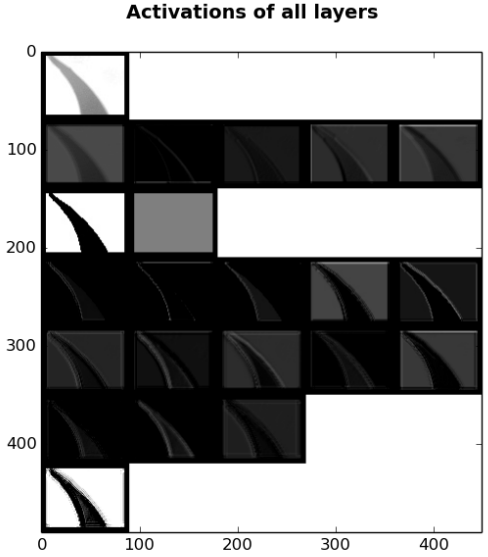
After 800 iterations



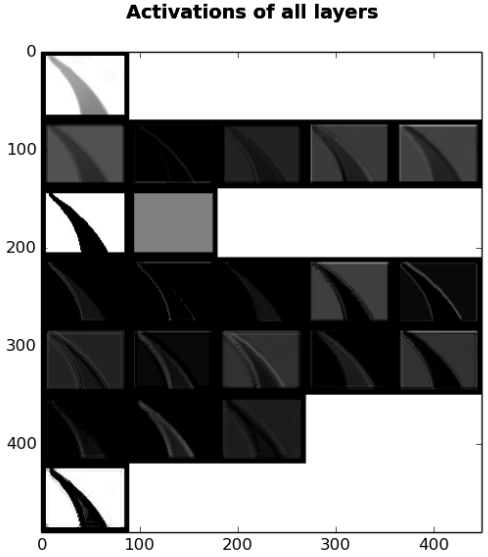
After 900 iterations



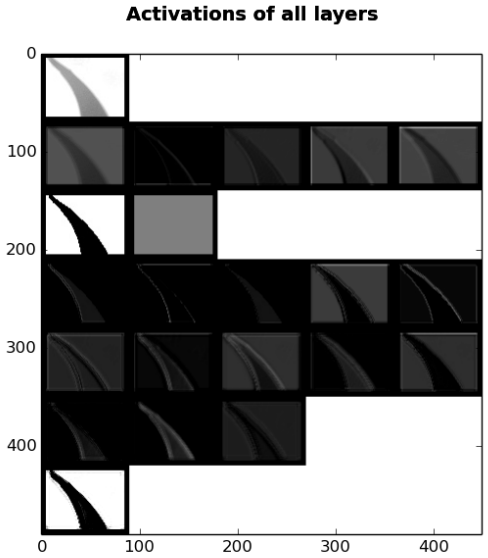
After 1000 iterations



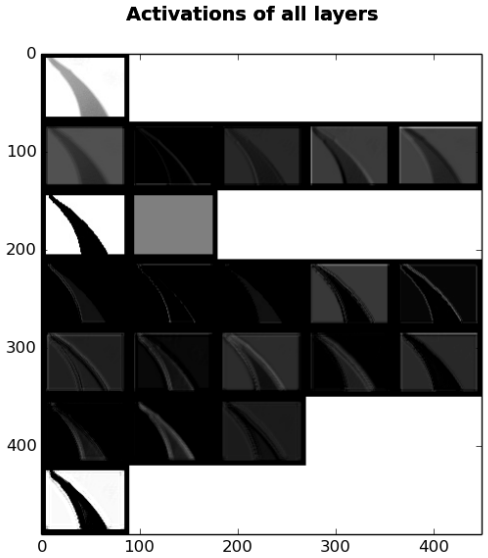
After 2000 iterations



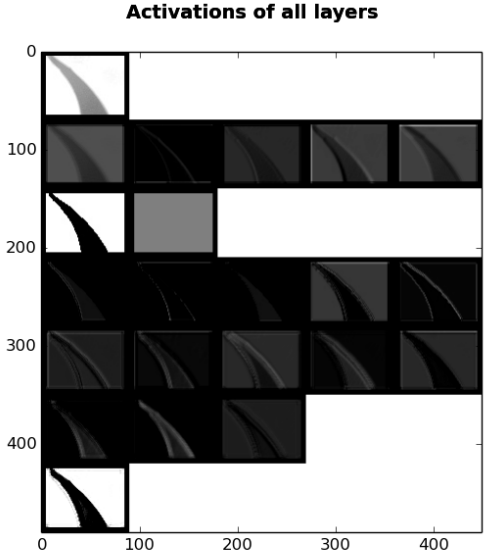
After 3000 iterations



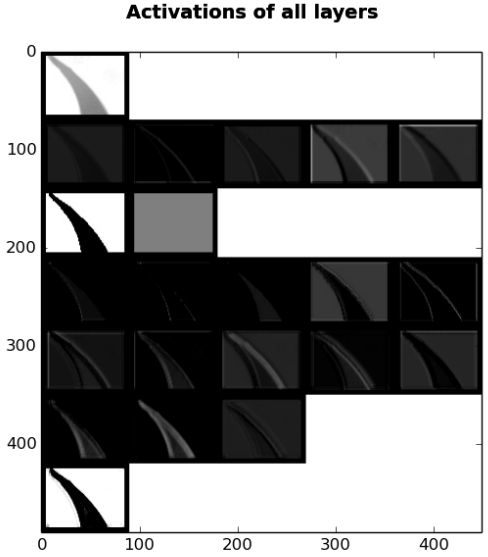
After 4000 iterations



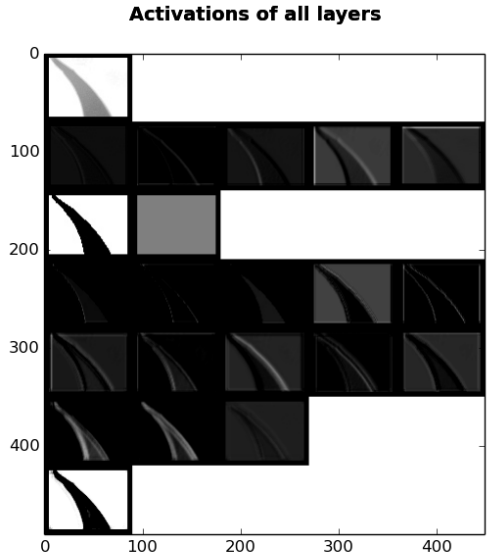
After 5000 iterations



After 10000 iterations



After 15000 iterations



References