Elements of neural network architecture

Columbia Water Center - Neural Network Working Group

Luc Bonnafous James Doss-Gollin

September 26^{th}, 2017

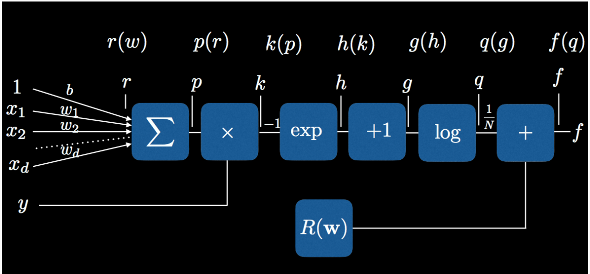
Columbia University

Table of Contents

Quick recap from last week

Chain Rule

Given a training sample **x** let's compute the gradient of f with respect to the weights θ.



$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial k}\frac{\partial k}{\partial p}\frac{\partial p}{\partial r}\frac{\partial r}{\partial w} + \frac{\partial R}{\partial w}$$

Figure 1: Chain rule for logistic regression example

Gradient Descent

(There are more complicated ways to do this). Take our initial weights, choose a learning rate $\eta = 0.1$ say:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} f(\theta_t) \quad (1)$$

Feedforward Network

We just need to generalize what we already learned.

- Let $y = f^*(\mathbf{x})$ be some function we don't know but want to approximate given data – could be a classifier boundary!
- Try to learn parameters θ to approximate this

A feedforward network has *no* feedback – layers are composed of functions ("layers") so that (e.g.)

$$f(\mathbf{x}) = l^3(l^2(l^1(\mathbf{x}))) \tag{2}$$

Over-fitting

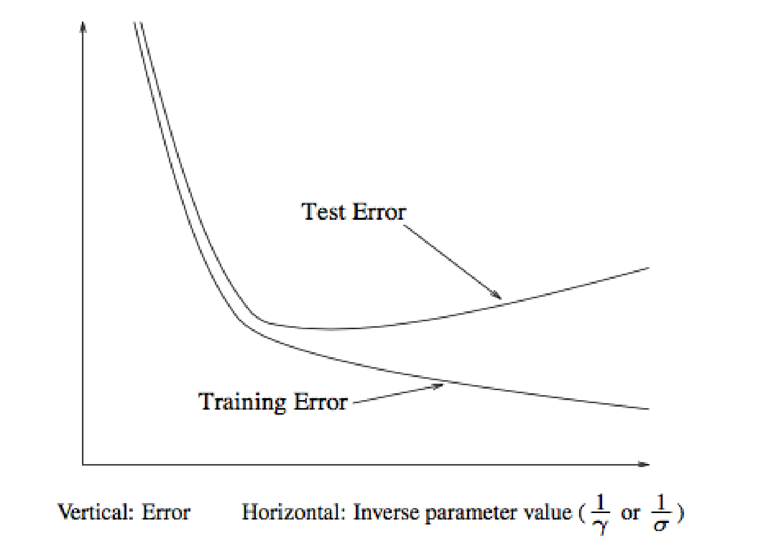


Figure 2: Over-fitting can be a huge challenge!

Ways to think of neural networks

- Sequence of layers mapping inputs into different "feature" spaces to learn meaningful feature combinations and draw conclusions
- Function approximators
- Ways to perform tasks such as classification, dimension reduction, and regression by learning from data

A little more on backpropagation

Basic questions and framing

- How to learn multiple layers of features?
- We need to find a way to update each weight based on an error metric
- This method needs to be general enough to be applied to a variety of multi-layer networks with non-linear activation functions (we thus need an iterative method)
- This method needs to be relatively efficient (better than difference-based methods based on weight perturbation)
- The idea is to use the error derivatives with respect to the activities of hidden units to evaluate the gradient the error function before applying some version of gradient descent

General set-up

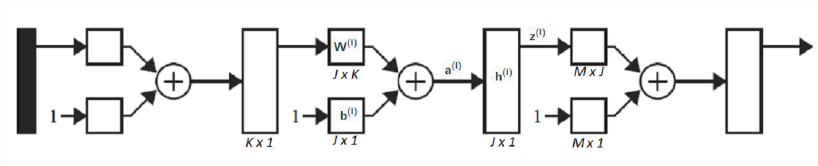


Figure 3: Adapted from Hagan et al., 2002

Insisting on the different steps involved

- Training algorithms generally use a two step procedure to minimize the chosen error function, with adjustments made to the weights at each step
- The first steps evaluates the gradient of the error function relative to the weights: this is where backpropagation comes in (propagation of errors backwards through the network)
- Then adjustments to be made to the weights are computed using the derivatives obtained in the previous stage, often using a version of gradient descent

General set-up

- Let there be a network with L layers in a given state, N training examples. We write an input vector X_n, where $n \in [\![1, N]\!]$, and the corresponding error E_n
- At the level of layer $l \in [\![1, L]\!]$, we write J the number of neurons, K the number of inputs, $Z^{(l-1)}$ the vector of inputs, $Z^{(l)}$, the vector of outputs, $\mathbf{W^{(l)}} \in \mathbb{R}^{JxK}$, the weight matrix, $w_{jk}^{(l)}$ the weight corresponding to input k and neuron j, $a_j^{(l)} = \sum_k w_{jk}^{(l)} z_k^{(l-1)}$, and $h_j^{(l)}$ the activation function of neuron j in layer l (thus $h_j^{(l)}(a_j^{(l)}) = z_j^{(l)}$)
- We are looking for

$$\frac{\partial E_n}{\partial w_{jk}^{(l)}} \tag{3}$$

Derivative computation

- For a given training example X_n, the error E_n only depends on $w_{jk}^{(l)}$ through $a_j^{(l)}$, thus, using the chain rule: $\frac{\partial E_n}{\partial w_{jk}^{(l)}} = \frac{\partial E_n}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{jk}^{(l)}}$
- We write $\delta_j^{(l)} = \frac{\partial E_n}{\partial a_j^{(l)}}$ the "error" for unit j
- We know that $\frac{\partial a_j^{(l)}}{\partial w_{jk}^{(l)}} = z_k^{(l-1)}$, thus $\frac{\partial E_n}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} z_k^{(l-1)}$
- $z_k^{(l-1)}$ is known from forward propagation, and, if we write M the number of neurons in the following layer,

$$\delta_j^{(l)} = \sum_m \frac{\partial E_n}{\partial a_m^{(l+1)}} \frac{\partial a_m^{(l+1)}}{\partial a_j^{(l)}} = h_j^{(l)\prime} \sum_m w_{mj}^{(l+1)} \delta_m^{(l+1)} \tag{4}$$

where the δ_m are known from the previous steps of backpropagation (this is where the error is propagated backwards; for the top layer, $\delta_m = E_n$)

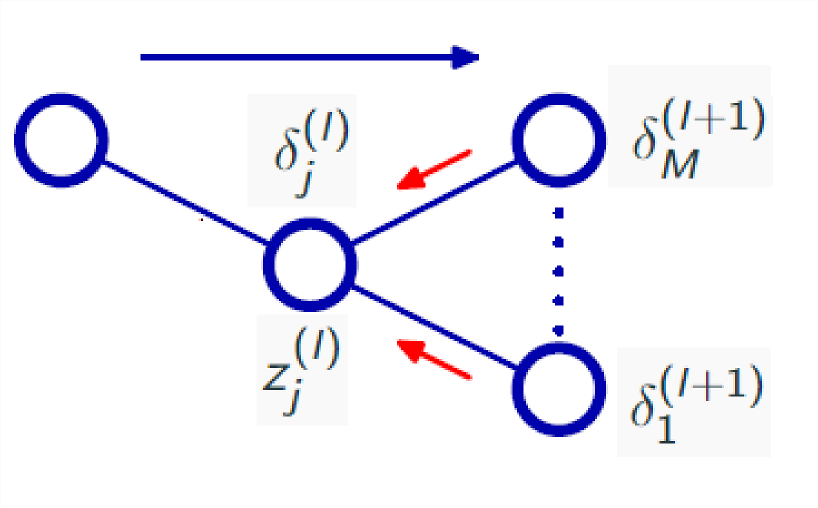


Figure 4: Adapted from Bishop, 2006

Derivative computation

- In the case a batch is used at each time step, the gradient used will be $\frac{\partial E}{\partial w_{jk}^{(l)}} = \sum_n \frac{\partial E_n}{\partial w_{jk}^{(l)}}$

Summary of what happens in practice

- A training example or a batch of training examples goes through the network, and the activations of all the units are recorded
- The errors of the output units are computed
- The δ are backpropagated using (4)
- The derivatives are computed

A couple more points

- Backpropagation can be used to compute more stuff, such as the Hessian matrix that can be used for more efficient optimization procedures, pruning, re-training etc.
- What was presented above did not include regularization, early stopping, dropout which are used to prevent overfitting
- For Recurrent Neural Networks, the algorithm needs to be slightly modified (but not really)
- The question of no-derivable activation function has not been touched on either (ReLU layer) but is not a real issue in practice

Activation functions

Canonical link functions

Name	Input/Output Relation	Icon	MATLAB Function
Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$		hardlim
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$		hardlims
Linear	$a = n$		purelin
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$		satlin
Symmetric Saturating Linear	$a = -1 \quad n < -1$ $a = n \quad -1 \leq n \leq 1$ $a = 1 \quad n > 1$		satlins
Log-Sigmoid	$a = \frac{1}{1 + e^{-n}}$		logsig
Hyperbolic Tangent Sigmoid	$a = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}}$		tansig
Positive Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n$		poslin
Competitive	$a = 1$ neuron with max n $a = 0$ all other neurons	C	compet

Figure 5: Source: Hagan et al., 2002

Output layer

There is generally an obvious choice

- Regression: identity
- Output strictly positive: softplus $y_k(a_k^{(K)}) = ln(1 + e^{(-a_k^{(K)})})$
- Binary classification: logistic sigmoid
- Mutli-class classification: softmax: $y_k(a_k^{(K)}) = \frac{e^{(-a_k^{(K)})}}{\sum_q (1+e_q^{(-a^{(K)}})}$

This choice often goes hand in hand with an error function:

- Regression: sum of squared errors
- Binary classification: cross-entropy
- Mutli-class classification: cross-entropy

Hidden layers and link to backpropagation

- *tanh* is preferred to the logistic sigmoid as it is zero-centered
- Recall $\delta_j^{(l)} = h_j^{(l)'} \sum_m w_{mj}^{(l+1)} \delta_m$
  For activation values outside a relatively small interval centered on zero, the gradient becomes very small, and thus almost no information on the error flows backward: the neuron won't get updated despite the existence of errors in the end, and the learning will become very slow or won't progress: vanishing gradient problem (note: exploding gradient problem)

ReLU

Nowadays' standard: ReLU (Rectified linear unit):
$y_k(a_j^{(l)}) = max(0, a_j^{(l)})$: it does not have the vanishing gradient problem, introduces non linearities, and its derivatives are easy to compute (more so that softplus, which is a smooth version of it)

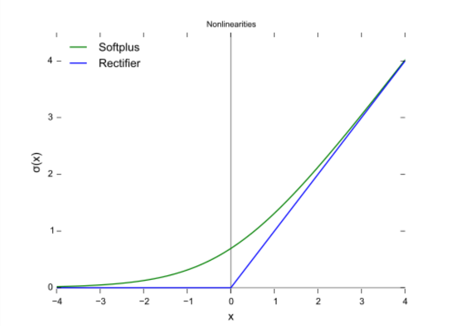


Figure 6: ReLU (blue) and softplus(green) - Source: Wikipedia

ReLU

It still has some issues: it is not differentiable at zero, it is not zero-centered, and if the learning rate is high, it can reach states when it never gets to activate again (the neuron is dead).

Other version: Leaky ReLU

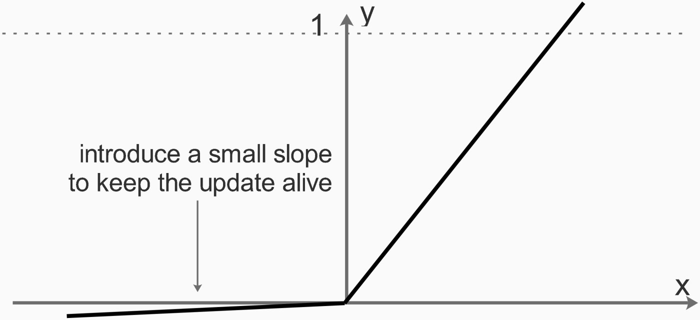


Figure 7: Leaky ReLU - Source: Wangxin's Blog

Out of context cell and layer presentations

Basic Layer

- This is your basic $\mathbf{z} = \mathbf{h}(\mathbf{WX} + \mathbf{b})$

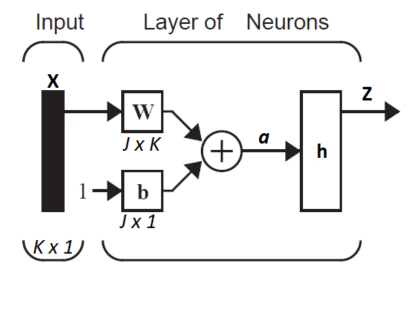


Figure 8: Basic layer schematic - adapted from Hagan et al. 2002

- Keras keras.layers.core.Dense, keras.layers.core.Activation, keras.layers.core.Lambda
- Tensorflow tf.layers.dense, tf.matmul, tf.tanh, etc.

Dropout layer

- This type of cell drops some units of the incoming layer given a certain probability

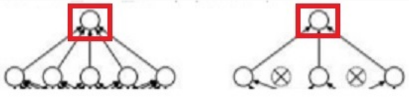


Figure 9: Dropout layer schematic
Source: Caffe framework

- Keras
  keras.layers.core.Dropout
- Tensorflow
  tf.layers.dropout

- This is used to prevent overfitting

Convolutional layer

- Performs cross-correlations between filters and inputs

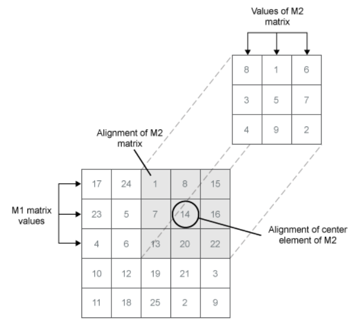


Figure 10: Cross-correlation between M1 and M2, - Source: Mathworks

- Keras
  keras.layers.convolutional
  .Conv2D
- Tensorflow
  tf.layers.conv2d

Convolutional layer

- The goal is to detect features of given shapes by mapping the input sequence into a sequence of feature spaces to detect feature combinations

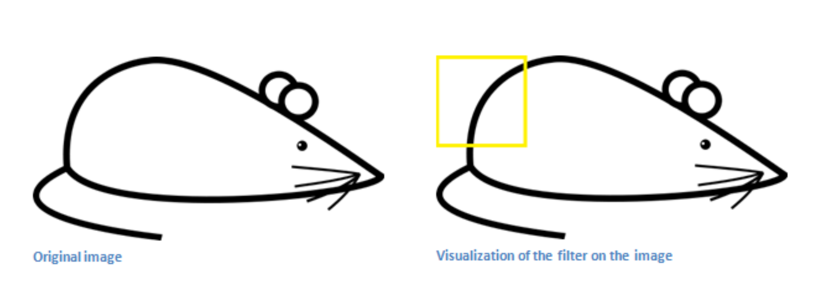


Figure 11: Source: Adit Deshpande

Convolutional layer

- The goal is to detect features of given shapes by mapping the input sequence into a sequence of feature spaces to detect feature combinations

Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

Figure 12: Source: Adit Deshpande

Convolutional layer

- Stride controls the way the filter convolves around the input structure

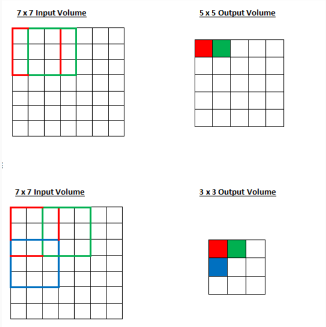


Figure 13: Source: Adit Deshpande

Convolutional layer

- Padding is here to adjust the output volume

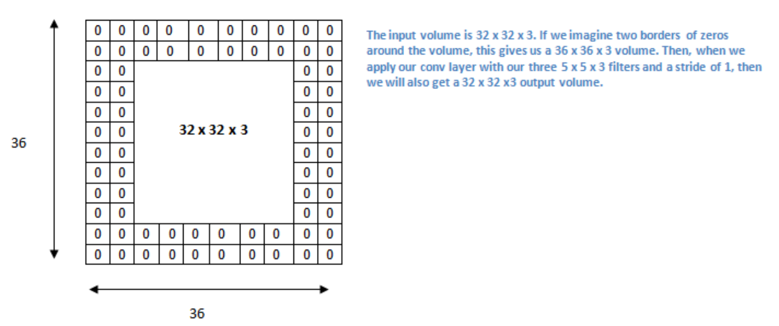


Figure 14: Source: Adit Deshpande

Pooling layers

- Take a statistics from the output of a filter

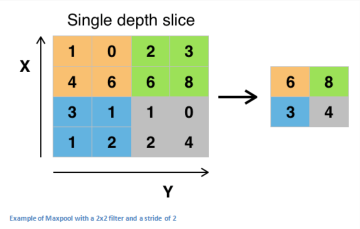


Figure 15: Max-pooling layer schematic - Source: Adit Deshpande

- Keras
  keras.layers.pooling.MaxPooling2
  keras.layers.pooling.AveragePooli
  etc.
- Tensorflow
  tf.layers.max_pooling2d, etc.

- Reduce the size of the info flowing if we don't really care about its exact location

Locally-connected layer

- Similar to convolution but with weights changing the filter weights for each position of the input
- Keras Keras.layers.local.LocallyConnected1D, Keras.layers.local.LocallyConnected2D
- Keep more spatial information

"Deconvolution" layers

- Go from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution

"Deconvolution": unpooling

- Uses both pooled maps and switches as inputs

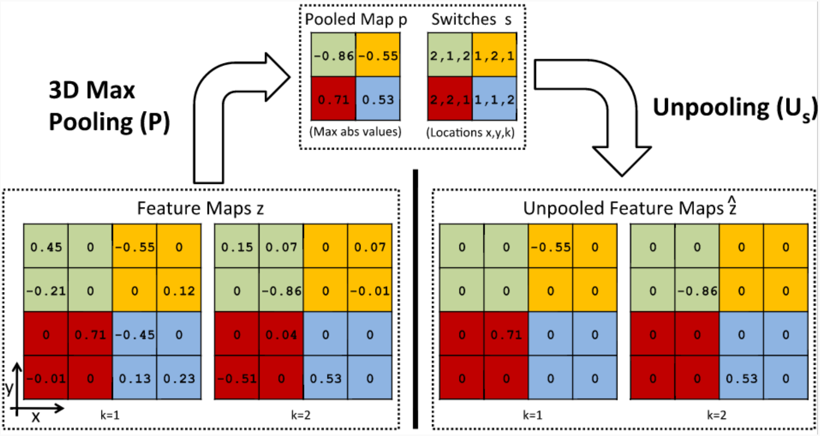


Figure 16: Unpooling - Source: Zeiler

"Deconvolution": transpose convolution

- Here the operation can be learned

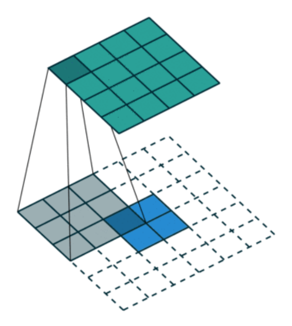


Figure 17: Transpose convolution - Source: Laboratoire d'Informatique des Systèmes

- Keras
  conv2d_transpose
- Tensorflow
  tf.layers.conv2d_transpose

"Deconvolution": transpose deconvolution

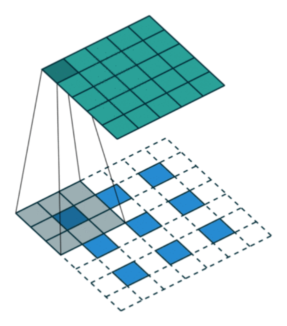


Figure 18: Transpose convolution - Source: Laboratoire d'Informatique des Systèmes d'Apprentissage

- Keras
  conv2d_transpose
- Tensorflow
  tf.layers.conv2d_transpose

Basic recurrent layers

- $\mathbf{z_t} = \mathbf{h}(\mathbf{W}\mathbf{X_t} + \mathbf{U}\mathbf{a_{t-1}} + \mathbf{b})$, $\mathbf{z_t} = \mathbf{h}(\mathbf{W}\mathbf{X_t} + \mathbf{U}\mathbf{y_{t-1}} + \mathbf{b})$

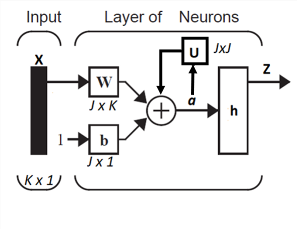


Figure 19: Basic recurrent layer

- Keras keras.layers.recurrent.SimpleRNN
- Tensorflow tf.contrib.rnn.BasicRNNCell

- Keep information from previous steps, thus introducing memory in the system. Useful if inputs are not independent

Basic recurrent layers

- Obviously, if you want to go too far back in time, it will screw up
- One representation
- Another one:

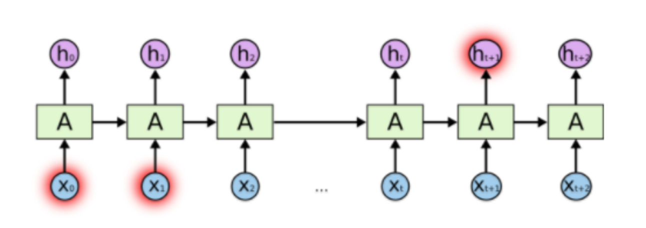


Figure 20: Basic recurrent layer - Source: Christopher Colah

Long short term memory layers

- Choose how much of the previous info we keep

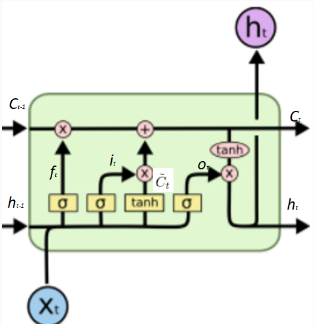


Figure 21: LSTM layer - Source: Christopher Colah

- Keras
  keras.layers.recurrent.LSTM
- Tensorflow
  tf.contrib.rnn.BasicLSTMCell

Long short term memory layers

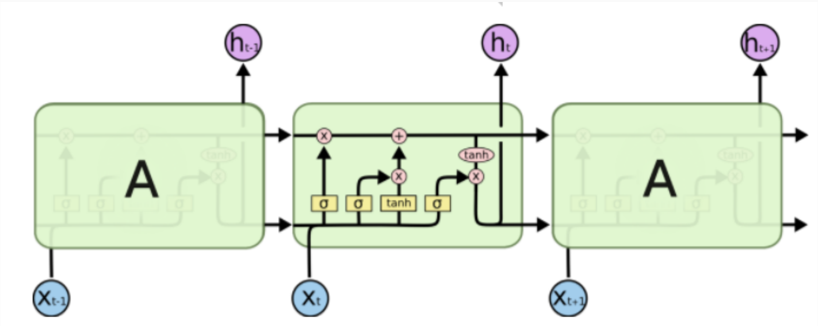


Figure 22: LSTM layer - Source: Christopher Colah

Long short term memory layers

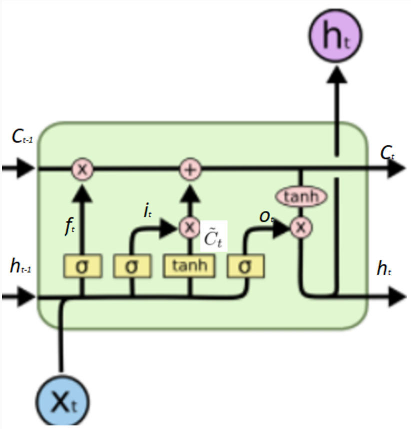


Figure 23: LSTM layer - Source: Christopher Colah

- $f_t = \sigma(W_f X_t + U_f h_{t-1} + b_f)$
- $i_t = \sigma(W_i X_t + U_i h_{t-1} + b_i)$
- $\tilde{C}_t = tanh(W_c X_t + U_c h_{t-1} + b_c)$
- $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
- $o_t = \sigma(W_o X_t + U_o h_{t-1} + b_o)$
- $h_t = o_t * tanh(C_t)$

Long short term memory layers

- Bi-directional RNN
- Peep-hole LSTM
- Convolutional LSTM
- Combining RNN and CNN, etc.

Network examples

CNNs for image recognition

Input -> Conv -> ReLU -> Conv -> ReLU -> Pool -> ReLU -> Conv -> ReLU -> Pool ->Fully Connected

Figure 24: CNN structure - Source: Adit Deshpande

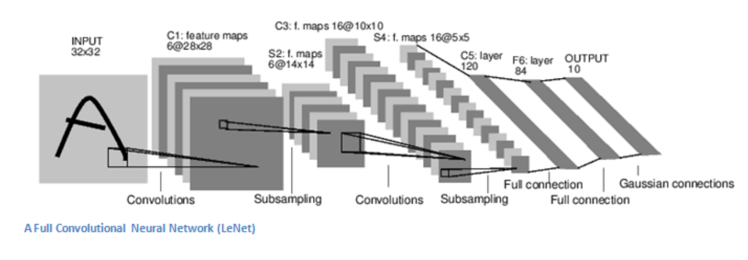


Figure 25: LeNEt structure - Source: LeCun 1998

One more MNIST tutorial

CNNs for image recognition

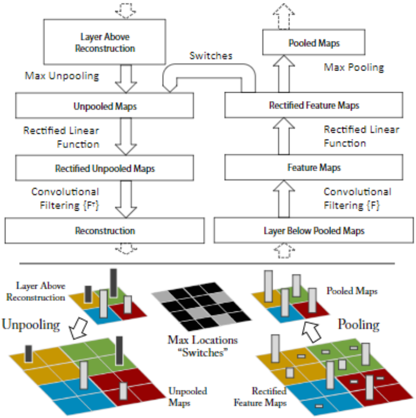


Figure 1. Top: A deconvnet layer (left) attached to a convnet layer (right). The deconvnet will reconstruct an approximate version of the convnet features from the layer beneath. Bottom: An illustration of the unpooling operation in the deconvnet, using *switches* which record the location of the local max in each pooling region (colored zones) during pooling in the convnet.

RNNs

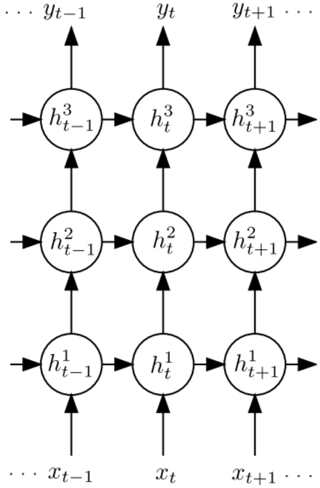


Fig. 3. Deep Recurrent Neural Network

RNNs

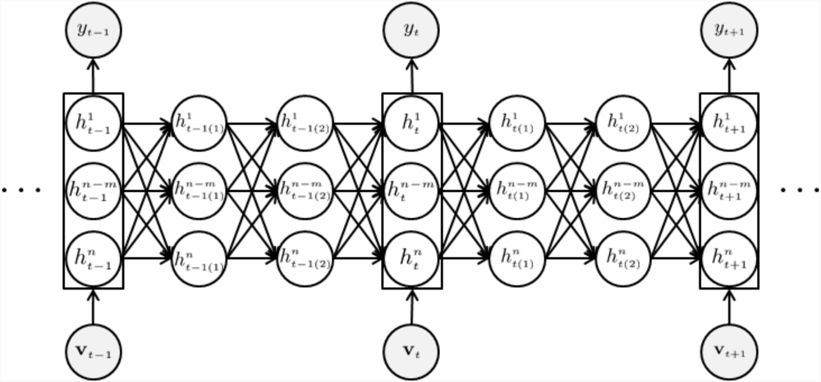


Figure 28: RNN - Source: MLC

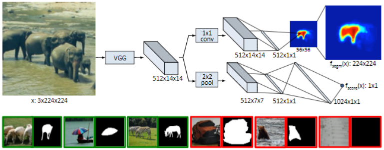


Figure 1: (**Top**) Model architecture: the network is split into two branches after the shared feature extraction layers. The top branch predicts a segmentation mask for the the object located at the center while the bottom branch predicts an object score for the input patch. (**Bottom**) Examples of training triplets: input patch x, mask m and label y. Green patches contain objects that satisfy the specified constraints and therefore are assigned the label $y = 1$. Note that masks for negative examples (shown in red) are not used and are shown for illustrative purposes only.

Figure 29: DeepMask - Source: Pinheiro et al. 2015

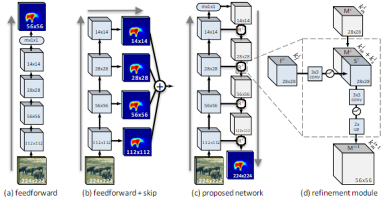
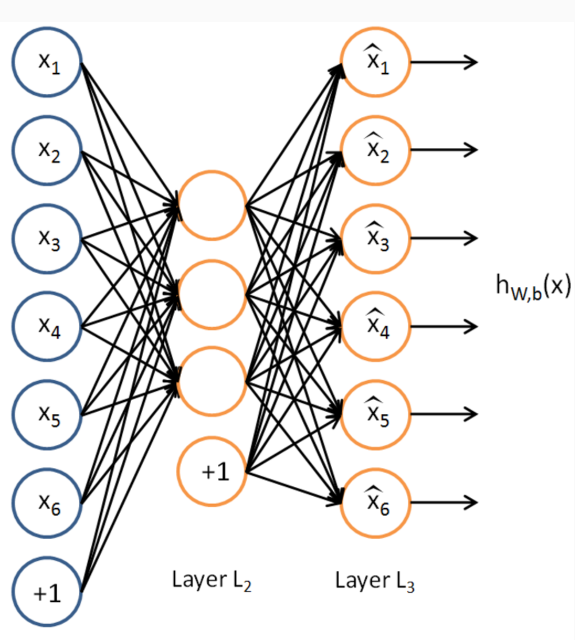


Fig. 1: Architectures for object instance segmentation. (a) Feedforward nets, such as DeepMask [22], predict masks using only upper-layer CNN features, resulting in coarse pixel masks. (b) Common 'skip' architectures are equivalent to making independent predictions from each layer and averaging the results [24, 29, 30], such an approach is not well suited for object instance segmentation. (c,d) In this work we propose to augment feedforward nets with a novel top-down refinement approach. The resulting bottom-up/top-down architecture is capable of efficiently generating high-fidelity object masks.

Figure 30: SharpMask - Source: Pinheiro et al. 2016

Autoencoder



Last words

- Not that many layer types in use, but variations of a couple
- Visualization DeepViz
- BNN

References and resources

Books in the shared drive

- Bishop 2006, Hagan 2002

Backpropagation

- DeepGrid
- wildml
- neuralnetworksanddeeplearning
- Colah's blog

CNNs & RNNs

- Deshpande's blog
- Daniil's blog
- NYU CS