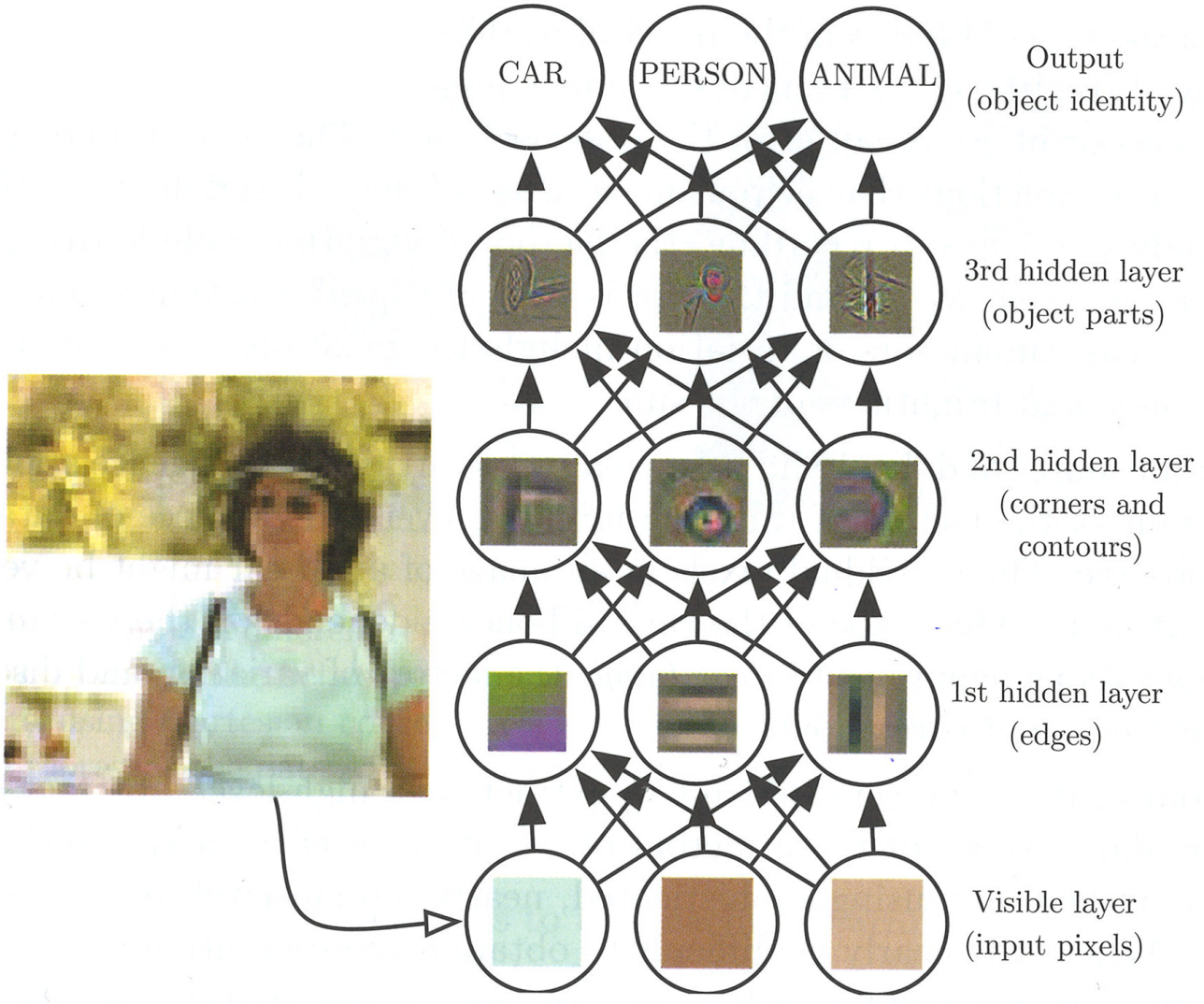
Advanced course in machine learning
582744
Lecture 12

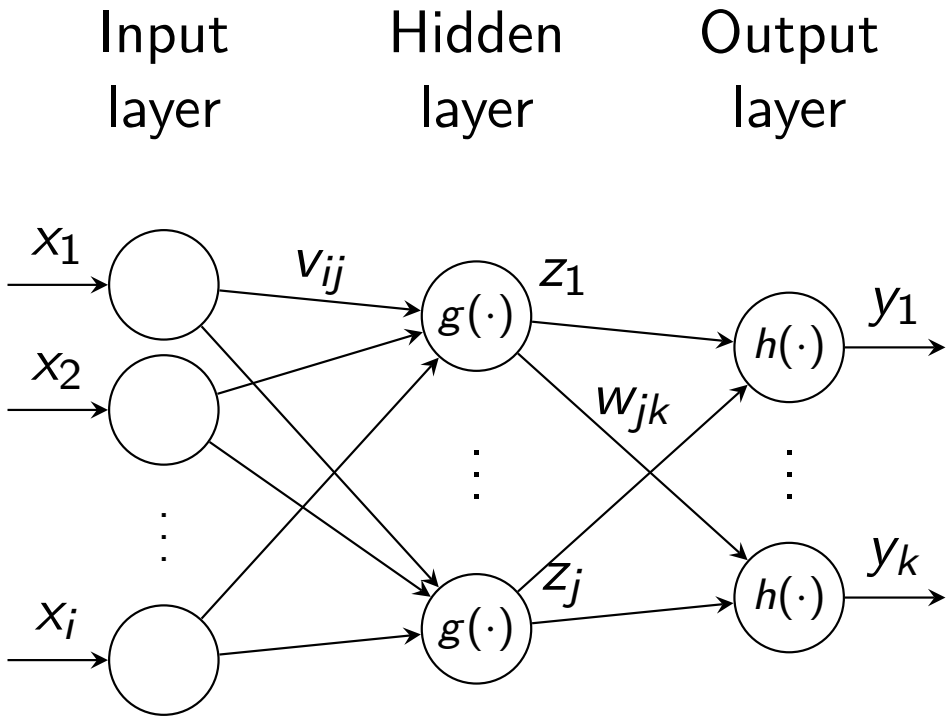
Mats Sjöberg

Previous lectures: deep learning



http://www.deeplearningbook.org/

Previous lectures: multilayer perceptron (MLP)



$$\mathbf{z} = g(\mathbf{V}\mathbf{x}) \qquad \mathbf{y} = h(\mathbf{W}\mathbf{z})$$

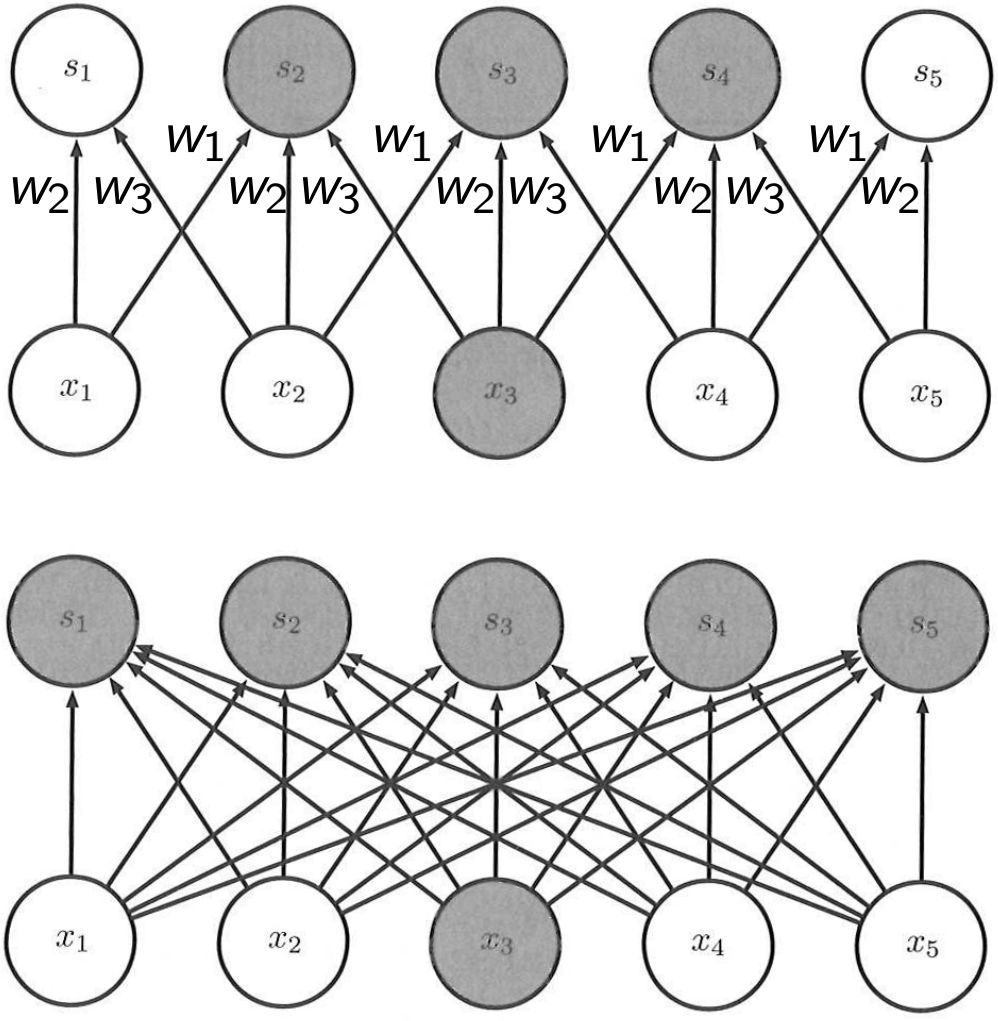
$$\mathbf{y} = h(\mathbf{W}g(\mathbf{V}\mathbf{x}))$$

Trained by backpropagating errors (gradients):

$\frac{\partial L}{\partial W_{kj}} = \delta_k z_j$, $\frac{\partial L}{\partial V_{ji}} = \delta_j^{(z)} x_i$, where $\delta_j^{(z)} = \sum_k \delta_k W_{kj} g'(a_j)$

Previous lectures: convolutional neural networks

Encode knowledge of the domain: 2-D images $\rightarrow$ convolution $\rightarrow$ sparse connectivity, parameter sharing



Previous lectures: convolutional neural networks

Typical architecture: convolutional layers, pooling (subsampling), ReLU (non-linearity)

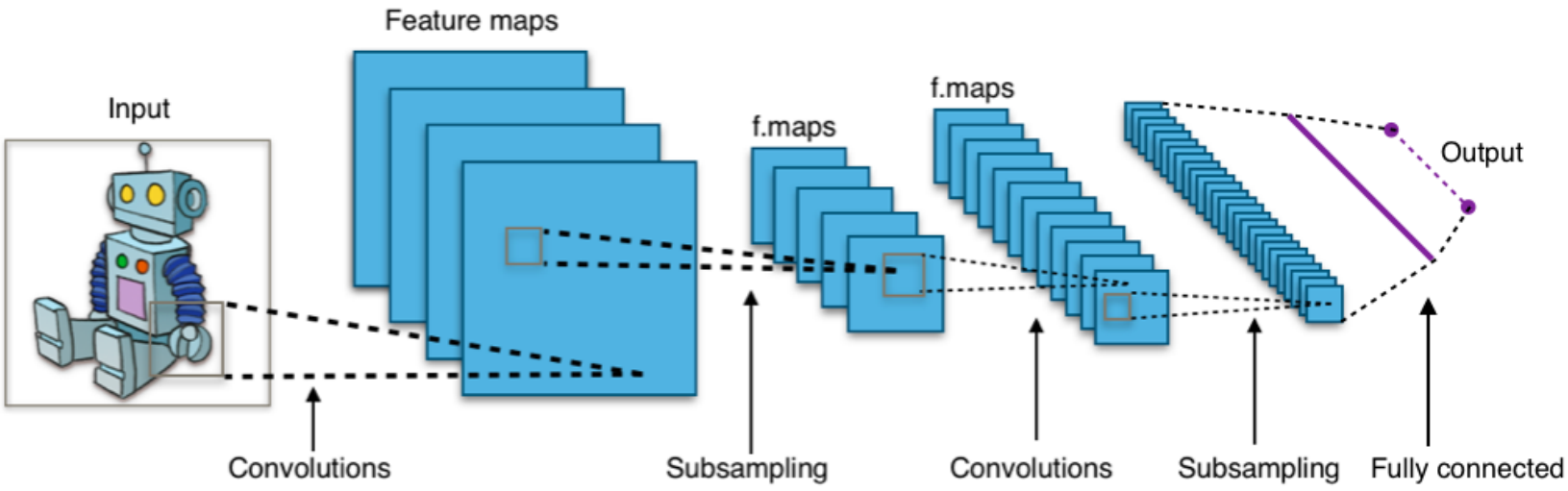


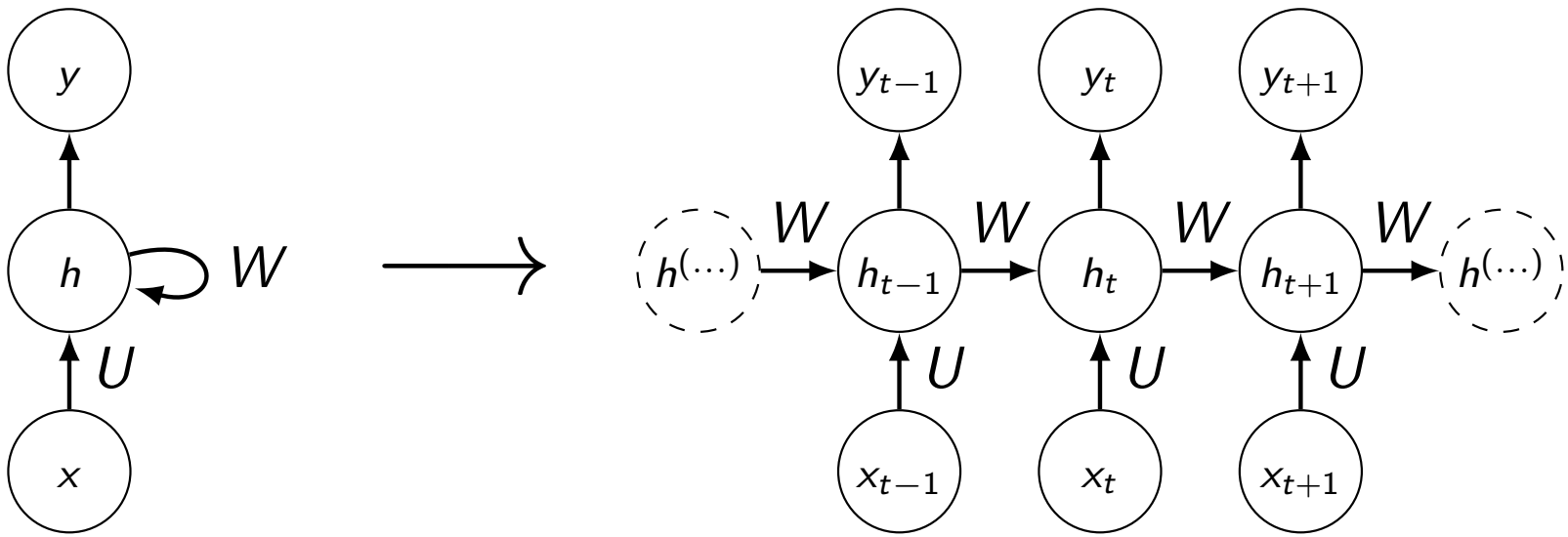
Image by Aphex34 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=45679374

Recurrent neural networks

So far we have considered only feedforward networks

You can also add *recurrent* connections which links the output of a node back from the previous time step

Can be thought of as infinitely deep feedforward network, unrolled as copies of itself



$$\mathbf{h}_t = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t)$$

Recurrent neural networks

Models $P(\mathbf{y}_t|\mathbf{x}_1, \ldots, \mathbf{x}_t)$ (by adding e.g. softmax), i.e., is perfect for processing sequential data

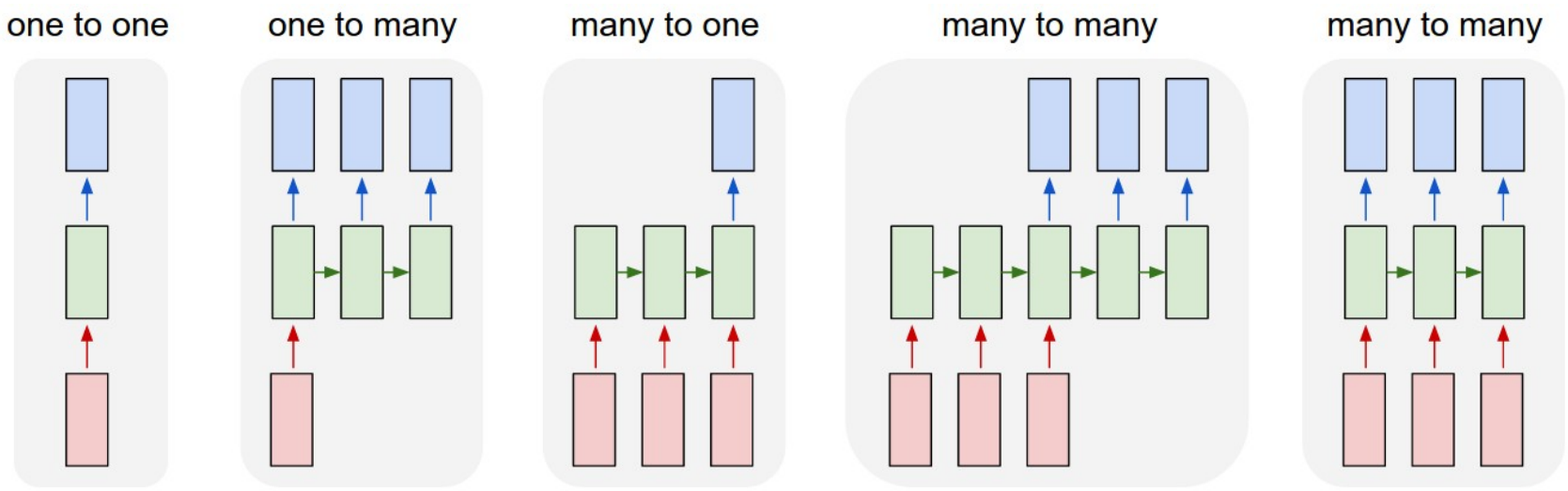
Example: predict the next word in a sentence given the previous t words

The network typically learns h_t as a summary of the inputs seen so far in a way relevant to the task (i.e., a kind of memory)

Training by *backpropagation through time* (BPTT), which is simply backpropagation for the unrolled network

Recurrent neural networks

RNNs allow us not only to process sequences, but also produce sequences as output



one to many e.g. image captioning

many to one e.g. sentiment analysis

many to many e.g. machine translation, or
in synchronisation, e.g. video classification

Image from
`http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

Recurrent neural networks

Input is essentially of variable size (compare to MLP with fixed size vector input)

Also the number of computational steps is essentially unlimited $\rightarrow$ powerful model

In fact RNNs are **Turing complete**

Also non-sequential data, such as images, can be processed in sequence by an attention mechanism (looking at different parts in sequence)

RNNs are already deep in the sense that if unrolled they have (in theory) infinite number of layers (although in practice the time depth is limited)

It is also common to stack RNN layers to get a deeper network

Vanishing gradient problem

As mentioned RNNs are trained using backpropagation through time (BPTT), which is simply backpropagation for the unrolled network

In RNNs the chain of backpropagation processing becomes quite long as you go back in time

Typically it is the same weights applied (multiplied) over and over again for the recurrent connection . . .

Hence gradients propagated over many layers tend to either vanish or explode

Makes it hard for an RNN to learn long-term dependencies

Vanishing gradient problem

The recurrent connection, ignoring the non-linearity, can be written as:

$$\mathbf{h}_t = \mathbf{W}\mathbf{h}_{t-1}$$

Thus recursively:

$$\mathbf{h}_t = \mathbf{W}(\mathbf{W}\mathbf{h}_{t-2}) = \ldots = \mathbf{W}^t\mathbf{h}_0$$

Using eigendecomposition $\mathbf{W} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$:

$$\mathbf{h}_t = \mathbf{Q}\mathbf{\Lambda}^t\mathbf{Q}^T\mathbf{h}_0$$

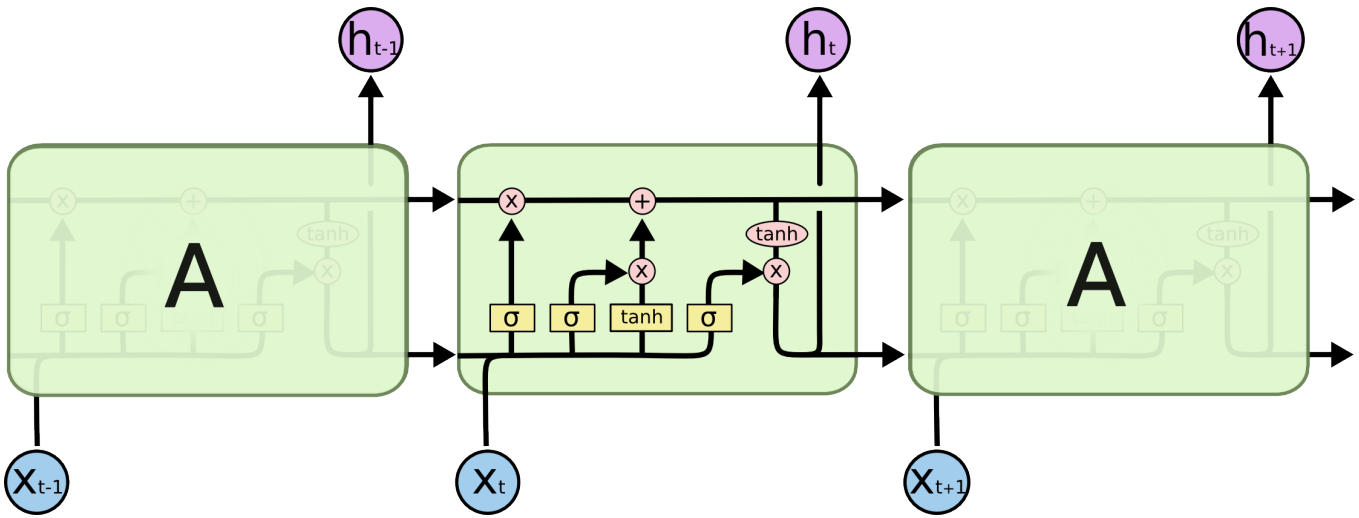
Eigenvalues < 1 will tend to zero (vanish) as t grows,
if > 1 they will become larger and larger (explode)

Long Short-Term Memory (LSTM)

LSTM (Hochreiter & Schmidhuber 1997), is currently the most popular RNN model that solves the vanishing gradient problem

LSTM adds a special **cell state** that self-loops without any direct changes (i.e., no weights that could vanish)

Weights instead control various gates that manipulate the cell state



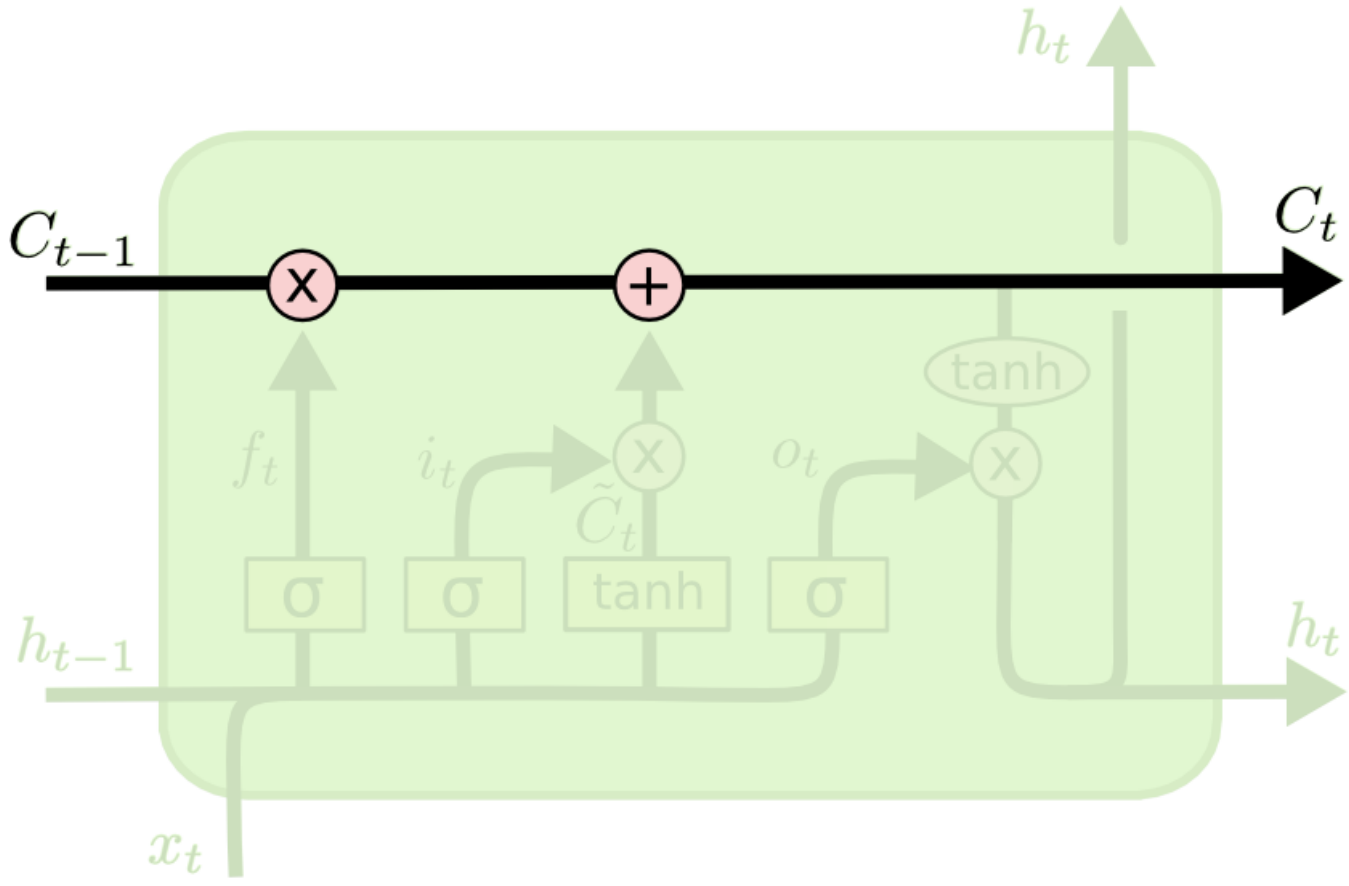
Pictures and explanation from

`http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

LSTM – cell state

C_t is the cell state – that runs through across "time"

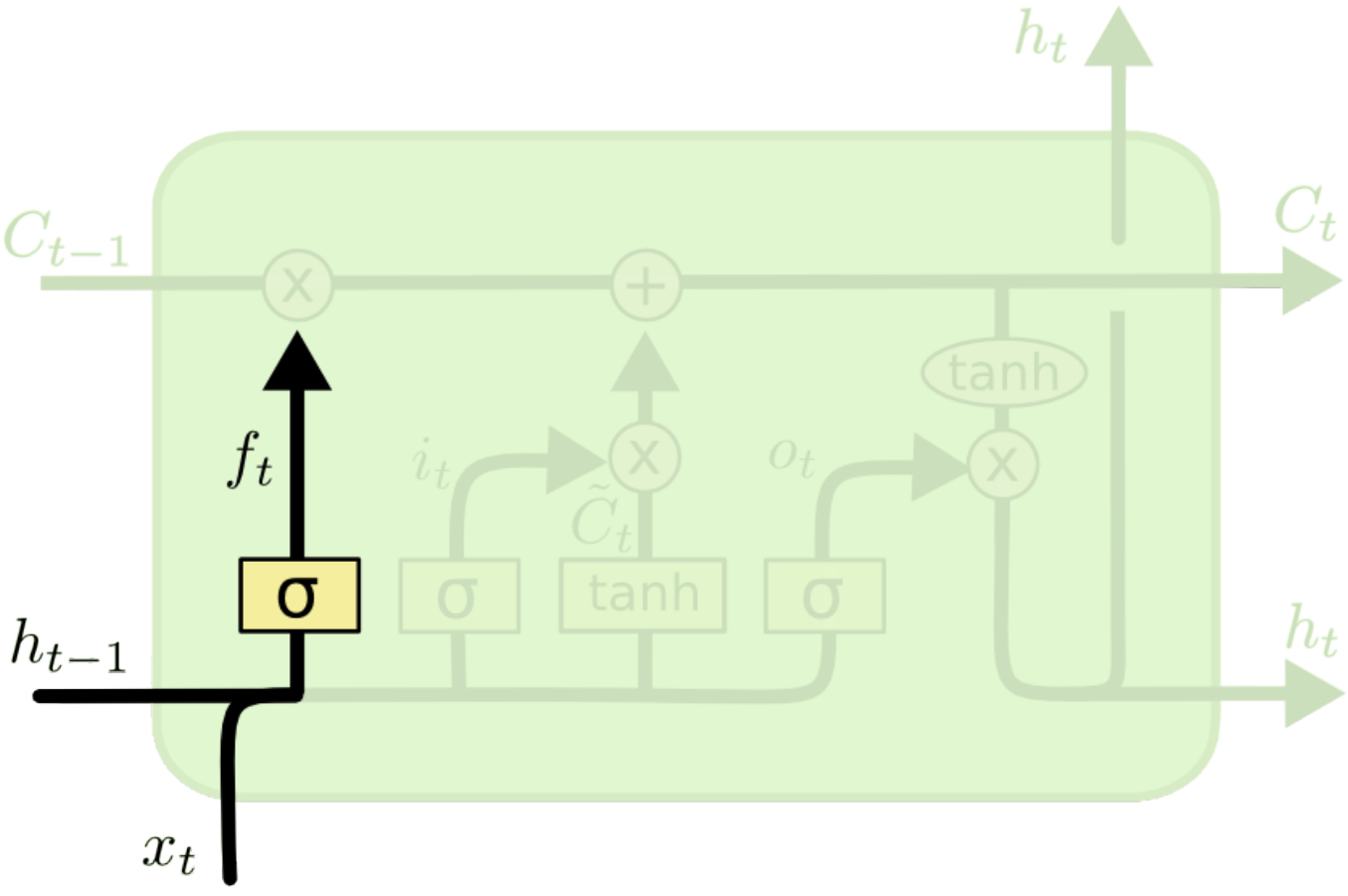
Any changes are controlled by special gates



LSTM – forget gate

$f_t = \sigma(U_f x_t + W_f h_{t-1})$ is the **forget gate**

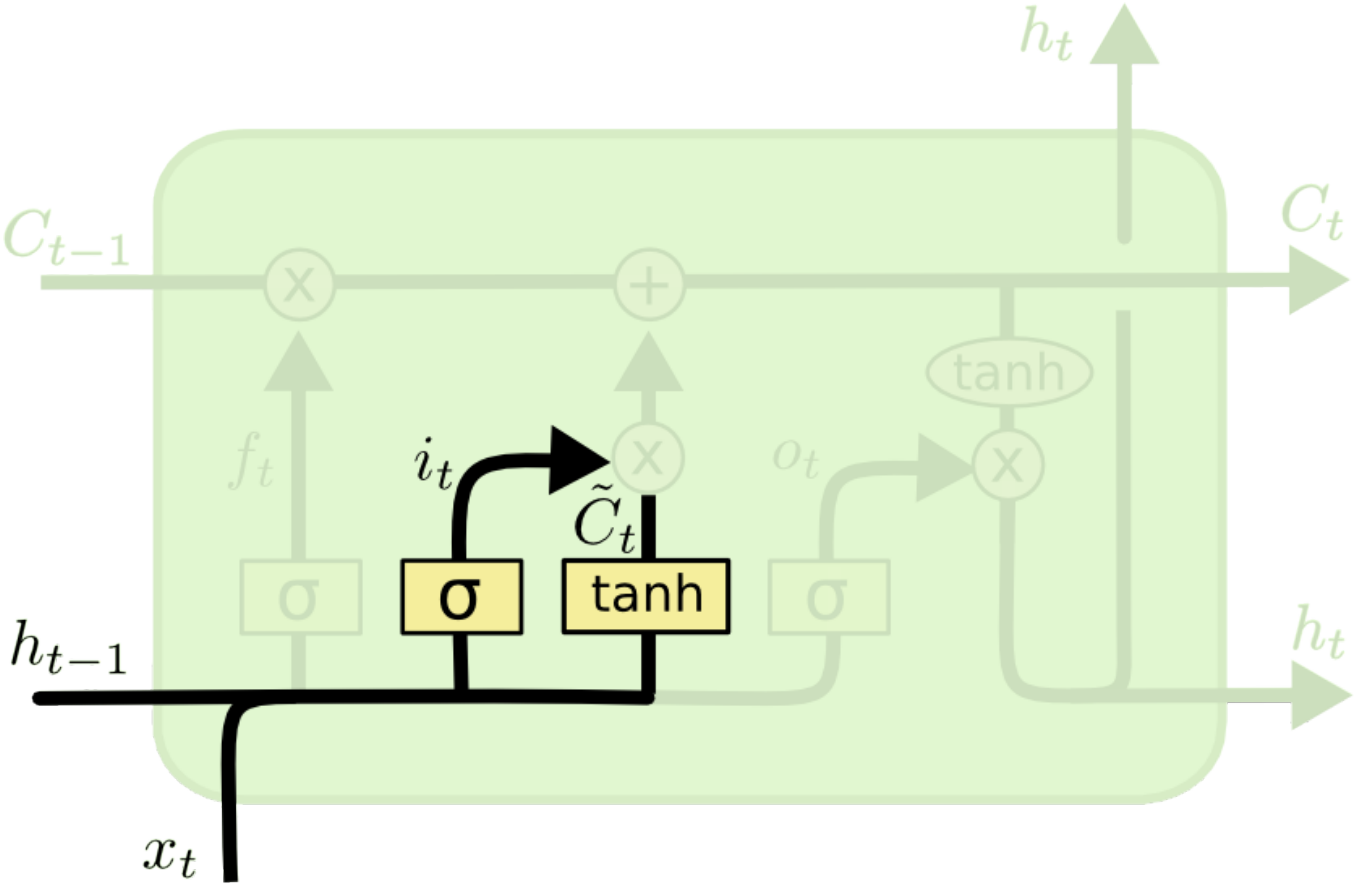
If a component of f_t is 0 it means "forget this", if it is 1 it means "keep this" in the cell state



LSTM – input gate

$i_t = \sigma(U_i x_t + W_i h_{t-1})$ is the **input gate** that decides which components to update

$\tilde{C}_t = \tanh(U_C x_t + W_C h_{t-1})$ is the candidate to update into C_t

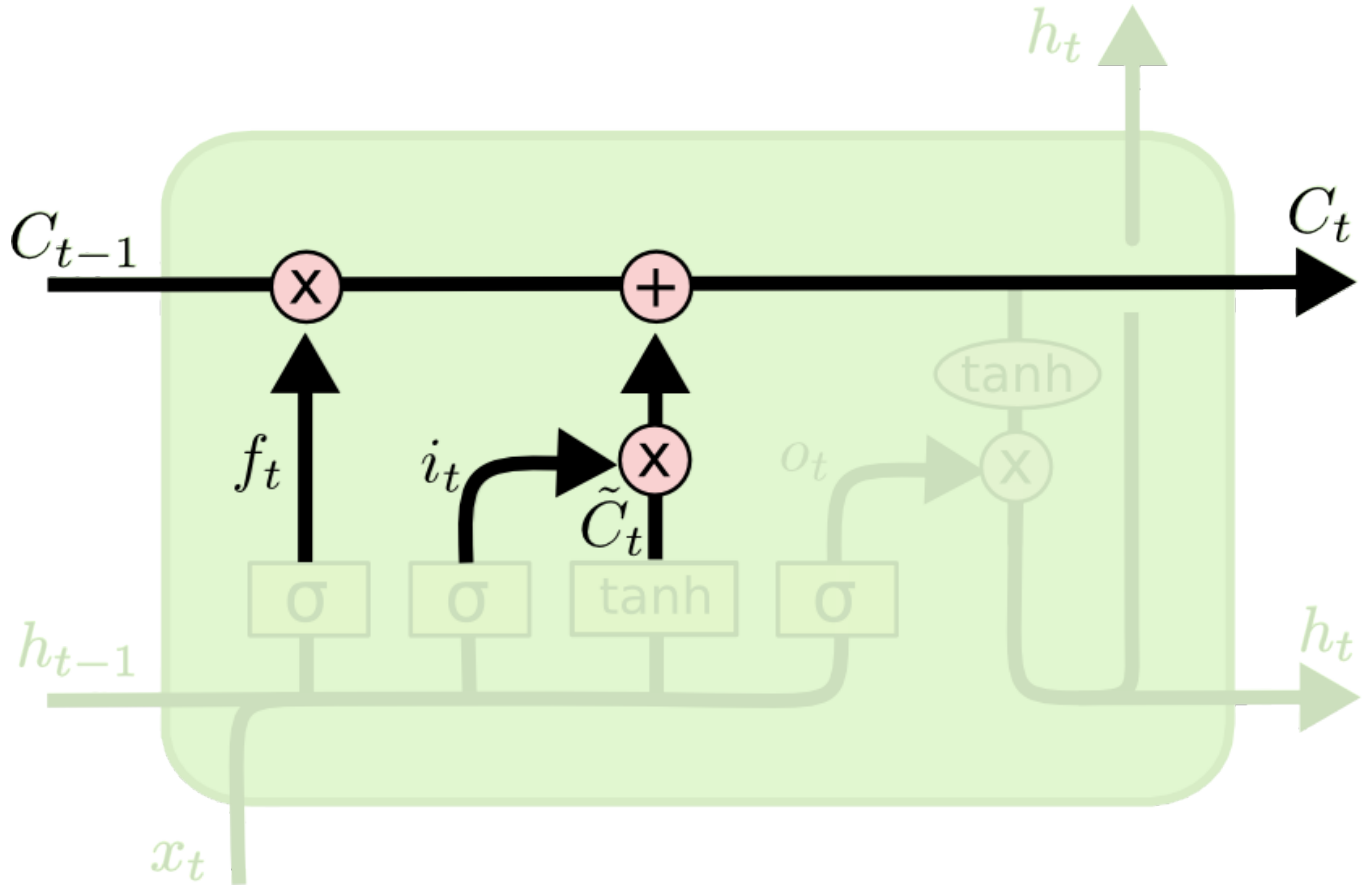


LSTM – updating the cell state

Here we actually apply the gates:

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

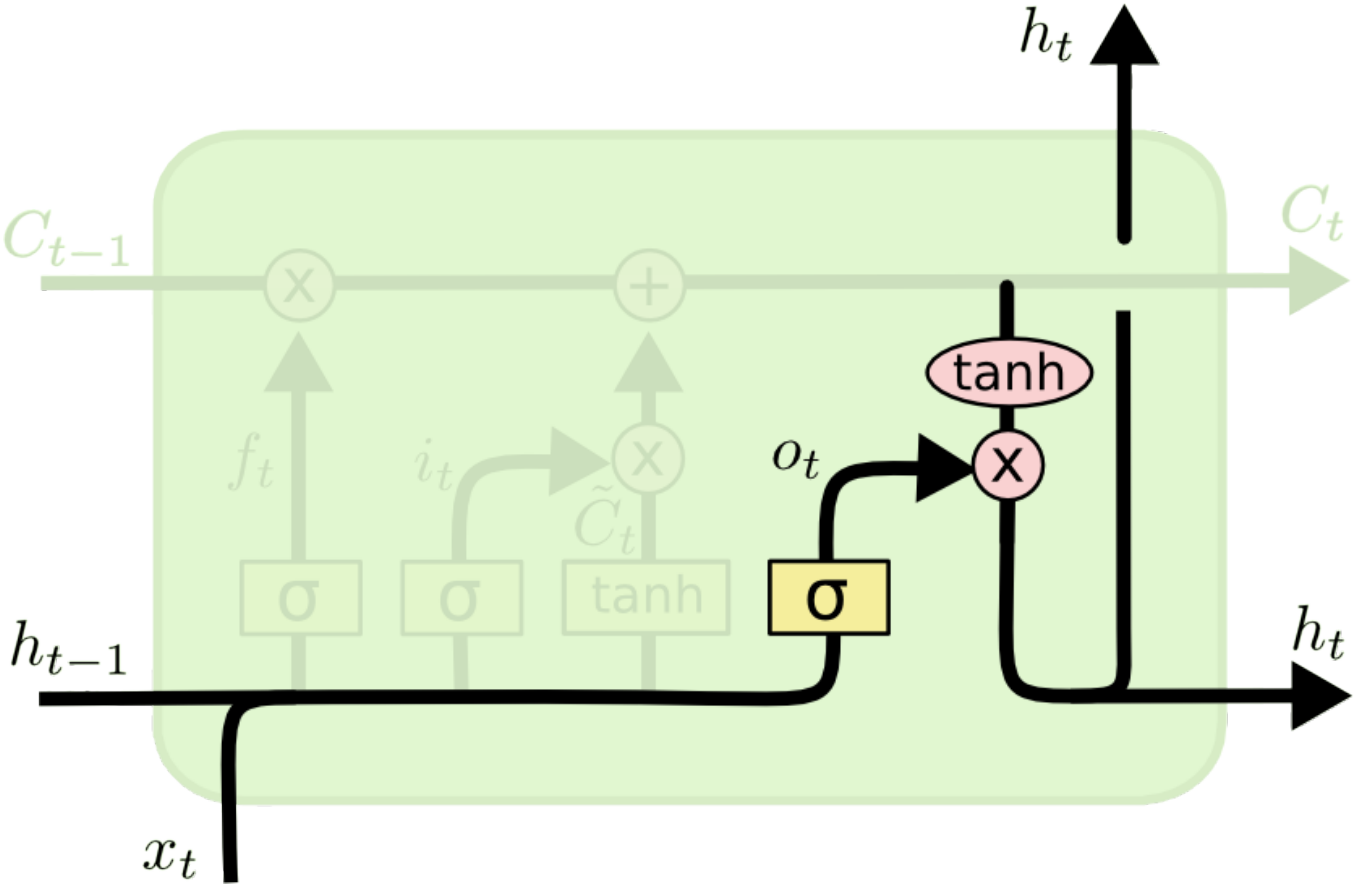
f_t controls what we keep or forget from the old state, i_t controls what new things we learn from the candidate state in $\tilde{C}_t$



LSTM – output

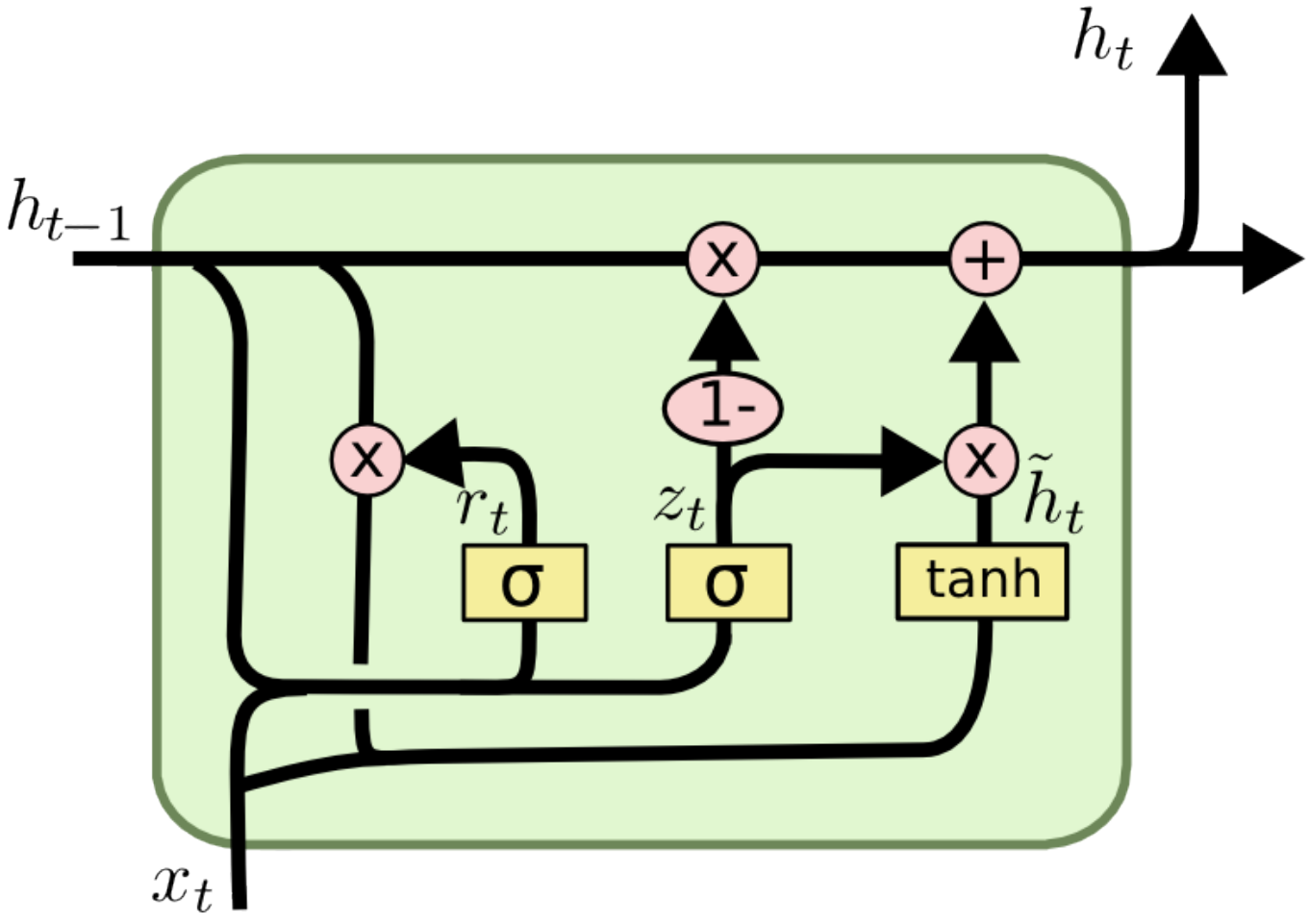
The output of the LSTM h_t is the cell state, but filtered by $o_t = \sigma(U_o x_t + W_o h_{t-1})$:

$$h_t = o_t \tanh(C_t)$$



Another variant: Gated Recurrent Unit

Gated Recurrent Unit (GRU) (Cho et al 2014)



RNN examples

Great blog post, fun text examples:
`http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

Visual Chatbot:
`http://demo.visualdialog.org/`

RNN examples: "Show and tell"

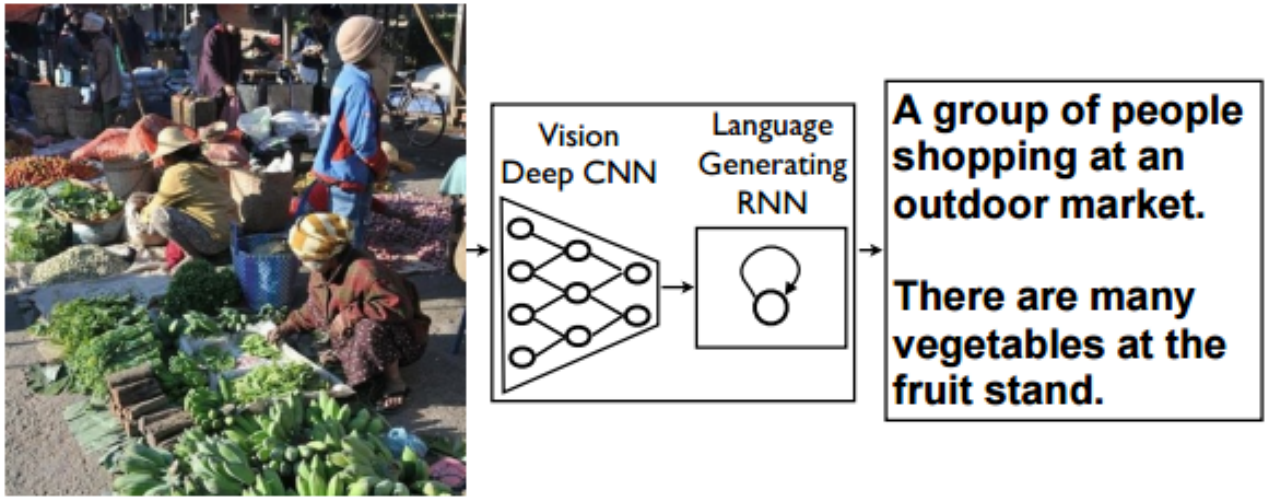


Figure 1. NIC, our model, is based end-to-end on a neural network consisting of a vision CNN followed by a language generating RNN. It generates complete sentences in natural language from an input image, as shown on the example above.

RNN examples: "Show and tell"

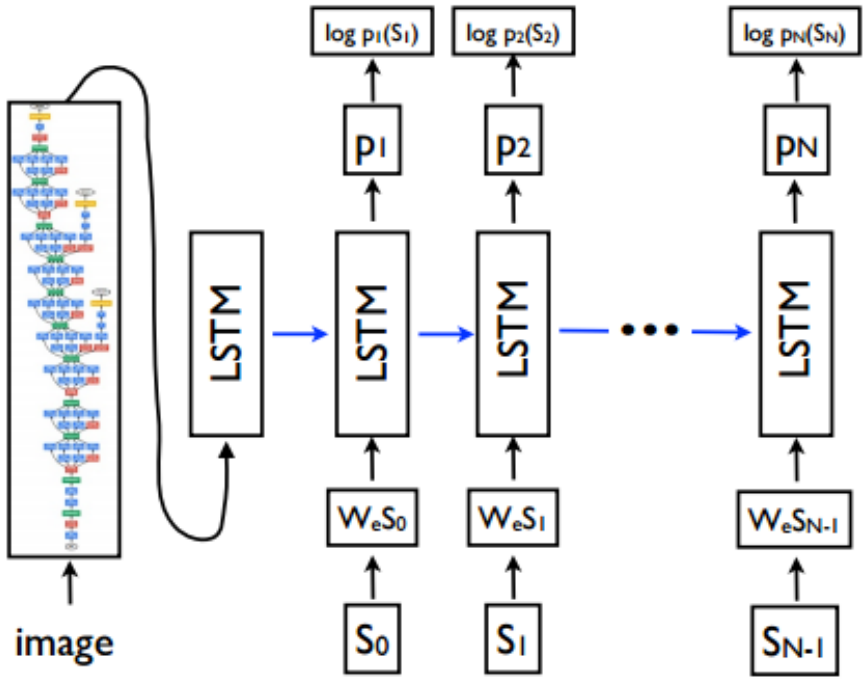
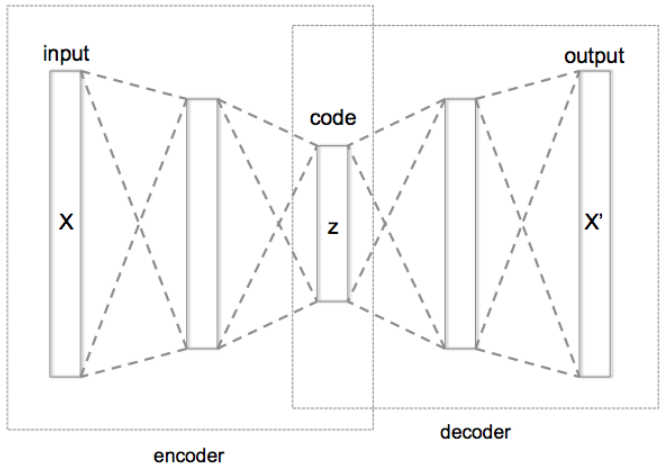


Figure 3. LSTM model combined with a CNN image embedder (as defined in [12]) and word embeddings. The unrolled connections between the LSTM memories are in blue and they correspond to the recurrent connections in Figure 2. All LSTMs share the same parameters.

Autoencoders

Autoencoders are neural networks where the desired output is the same as the input (!)

The idea is that in between there are "narrow" hidden layers which force the information to be compressed into a compact format



The first part of the network that compresses the inputs is called **encoder** and the second part is **decoder**, whereas the narrowest part of the representation is the **code** – this matches the compression terminology

Autoencoders

The encoder transforms the input $\mathbf{x}$ into the code $\mathbf{z}$:

$$\mathbf{z} = f(\mathbf{x})$$

And the decoder from the code back to the reconstructed $\mathbf{x}'$

$$\mathbf{x}' = g(\mathbf{z})$$

We want to minimize the loss $L(\mathbf{x}, \mathbf{x}') = L(\mathbf{x}, g(f(\mathbf{x})))$ which penalizes large differences between the input and the output

For linear encoder/decoder and if L is MSE, we get PCA

Denoising autoencoders

A neat trick is to train autoencoders by feeding in noisy versions of the vectors but still use the originals as outputs (kind of like the modified inputs in supervised learning)

The network learns to de-noise the inputs and often solves the original autoencoding task more accurately as well

With denoising autoencoders we can actually use codes that are larger than the input – the same would hold if we regularize the network enough

Representation learning

Supervised deep networks often learn interesting internal representations for the hidden layers

These are called **distributed representations** (compare to symbolic representation, or cluster-based representations)

We can use those as "unsupervised" feature extraction for other tasks; they are probably good features if they helped in solving the supervised task

If you need vectorial representations for images – for any purpose – you can use the last hidden layer(s) of a CNN trained to classify images

Representation learning

We can even start with the goal of learning the representation and use some artificial task for training the network

For example, we can learn **word embeddings** to represent natural language words as vectors by predicting the word based on the context, or the other way around (Mikolov et al. 2013)

"word2vec": word embeddings allow "semantic" arithmetics:
W("woman") - W("man") + W("aunt") = W("uncle")

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Deep generative models

If the input is $\mathbf{x}$ and the corresponding labels $\mathbf{y}$, generative models learn $P(\mathbf{x}, \mathbf{y})$, while discriminative models learn $P(\mathbf{y}|\mathbf{x})$

(generative: or $P(\mathbf{x}|\mathbf{y})$ and $P(\mathbf{y})$ together)

From $P(\mathbf{x}, \mathbf{y})$ you can get $P(\mathbf{y}|\mathbf{x})$ (Bayes' rule) or sample new $\mathbf{x}, \mathbf{y}$ pairs

You start with a theory of how the data is generated (e.g. mixture of Gaussians), and then try to fit its parameters to the data

Naturally there are also "deep" generative models :-)

Variational autoencoder

Basically an autoencoder with a probabilistic twist

The hidden layer outputs are distribution parameters, typically mean and variance of a Gaussian distribution

The net is trained to get as close as possible to a unit Gaussian (mean zero, variance one)

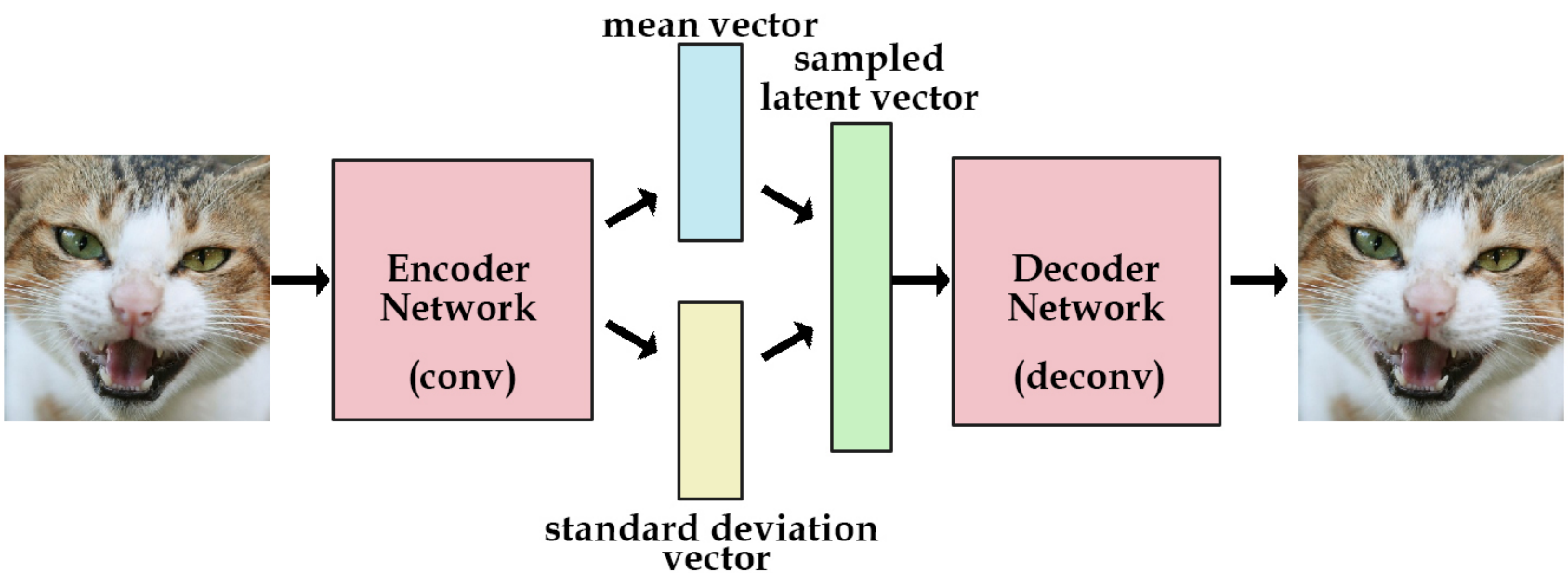


Image from `http://kvfrans.com/variational-autoencoders-explained/`

Generative Adversarial Networks (GAN)

"The coolest idea in machine learning in the last twenty years"
– Yann LeCun

We have two networks: **generator** and **discriminator**

The generator produces samples, while the discriminator tries to distinguish between real data items and the generated samples (by assigning a probability value)

The discriminator tries to learn to classify correctly, while the generator in turn tries to learn to fool the discriminator

The result is typically that the generator produces better and better images, and in the end the discriminator outputs 0.5 every time . . .

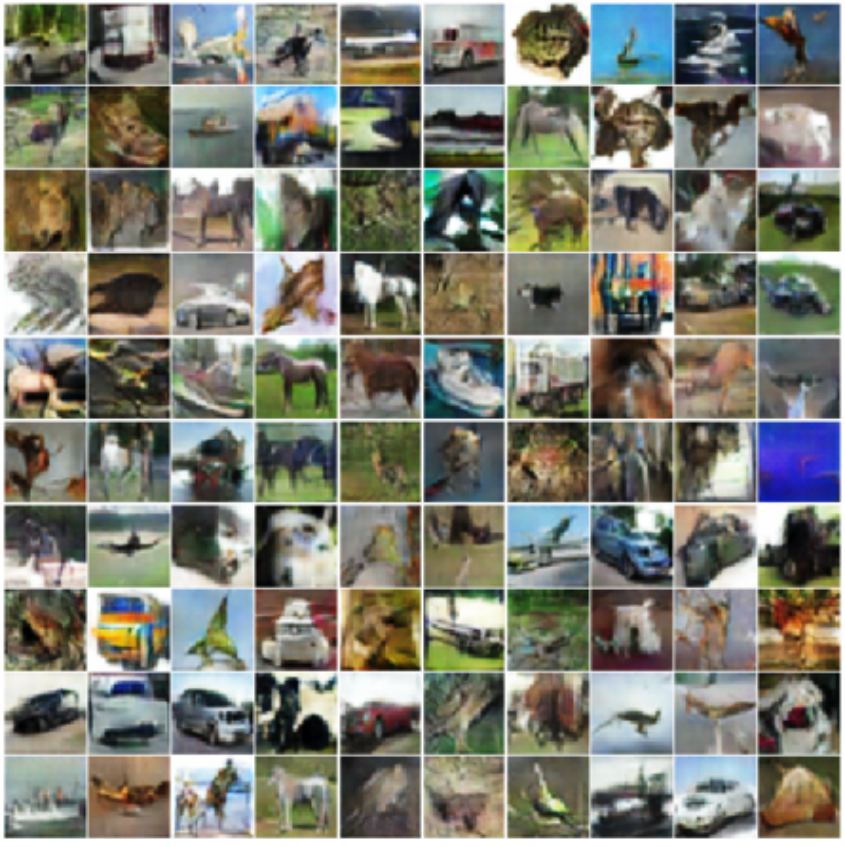
GAN examples

Generated bedrooms

https://arxiv.org/abs/1511.06434v2

GAN samples

Generated CIFAR-10 examples



https://arxiv.org/abs/1606.03498

GAN examples

Generative Adversarial Text to Image Synthesis
`https://arxiv.org/pdf/1605.05396.pdf`

Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks
`https://junyanz.github.io/CycleGAN/`

What's next for the course?

no lecture on Tuesday (May 2)

"recapitulation lecture" on Thursday (May 4) – possible changes will be advertised on Moodle

Two more exercise sessions: tomorrow and next week's Friday (May 5)

Exercise 7 for bonus points (released on Tuesday)

The course exam is May 11 at 16:00 in B123

Advertisements

- 582762: Special Course in Unsupervised Machine Learning: Probabilistic Factor Analysis Methods

  Intensive course, starts Mon 15.5.2017

  `https://courses.helsinki.fi/582762/119045100`

- Seminar course in Deep Learning

  Starts Wed 1.11.2017, no course web page yet ...

  Some lectures, but also strong focus on practical implementation