# Deep Networks

Andrea Passerini
passerini@disi.unitn.it

Machine Learning

# Need for Deep Networks

## Perceptron

- Can only model *linear* functions

## Kernel Machines

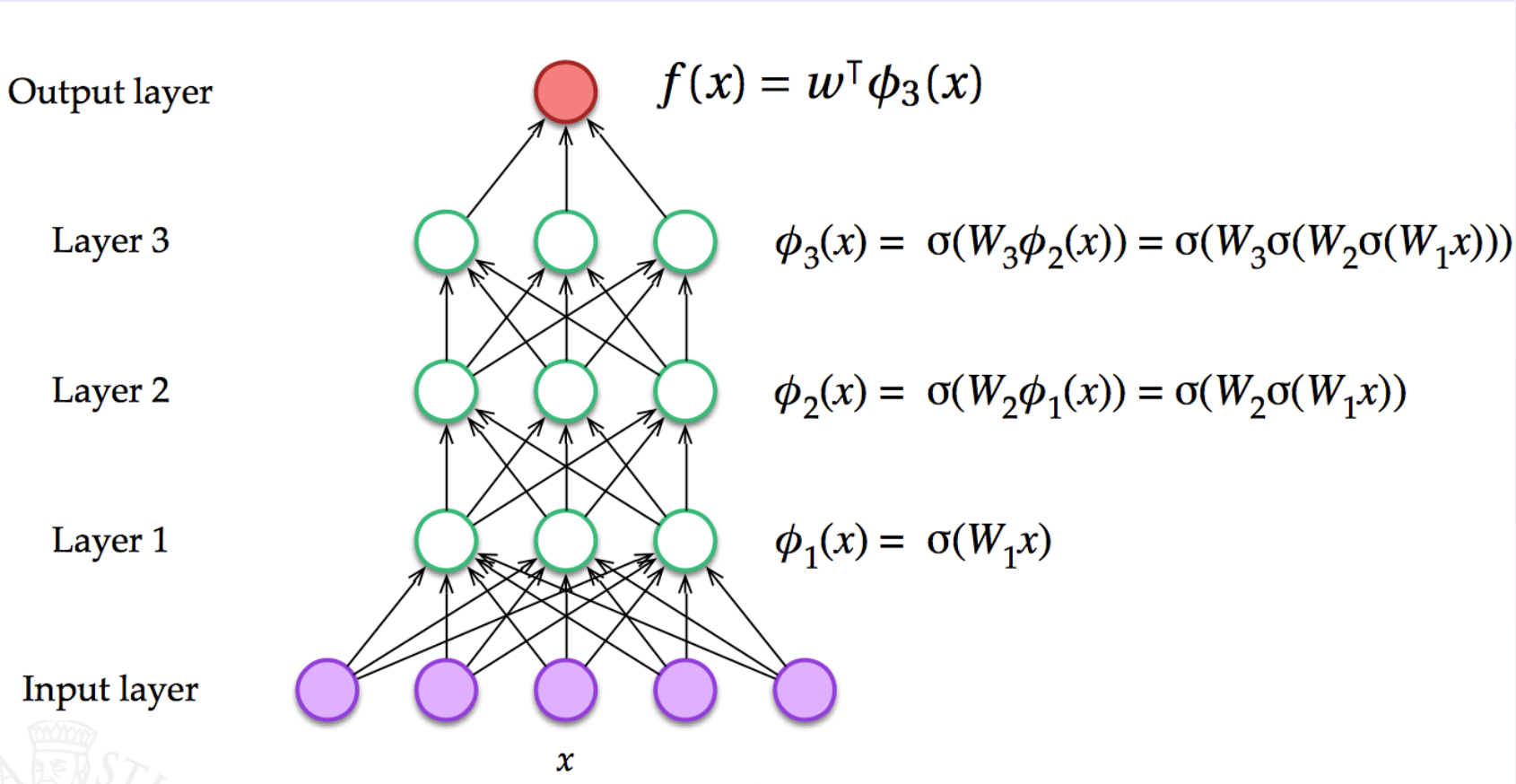
- Non-linearity provided by kernels
- Need to *design* appropriate kernels (possibly selecting from a set, i.e. kernel learning)
- Solution is *linear combination of kernels*

# Need for Deep Networks

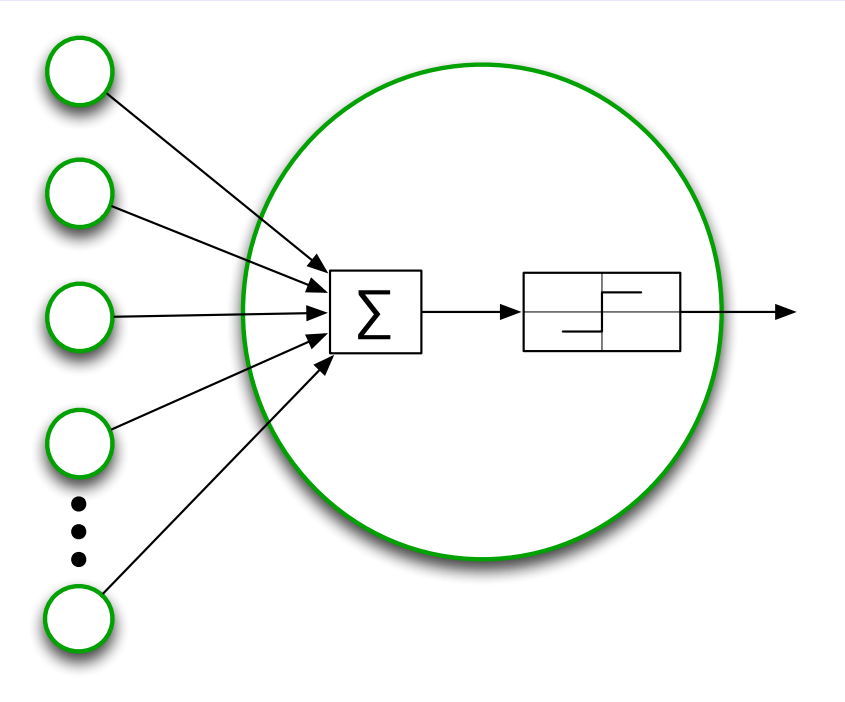
## Multilayer Perceptron (MLP)

- Network of interconnected neurons
- *layered* architecture: neurons from one layer send outputs to the following layer
- Input layer at the bottom (input features)
- One or more hidden layers in the middle (learned features)
- Output layer on top (predictions)

# Multilayer Perceptron (MLP)



# Activation Function

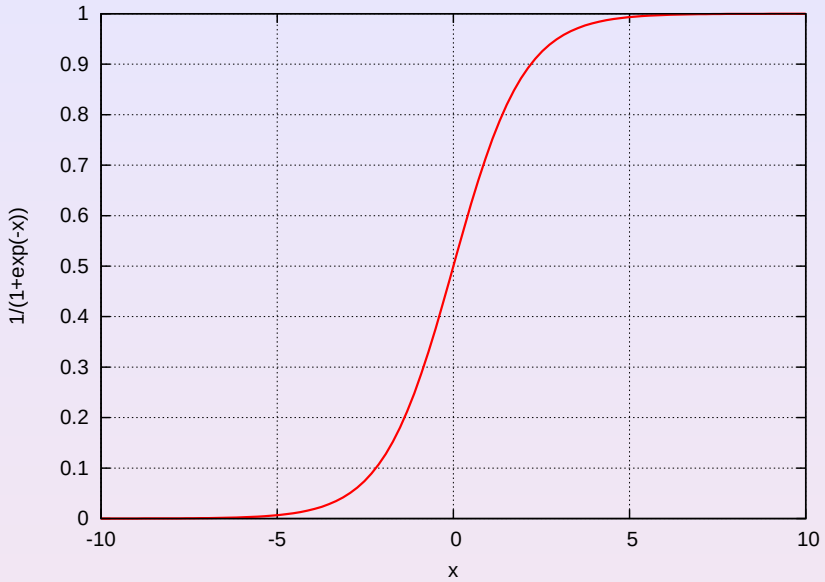



## Perceptron: threshold activation

$$f(x) = sign(\boldsymbol{w}^T \boldsymbol{x})$$

- Derivative is zero everywhere apart from zero (where it's not differentiable)
- Impossible to run gradient-based optimization

# Activation Function




## Sigmoid

$$f(\boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{x}) = \frac{1}{1 + \exp(-\boldsymbol{w}^T \boldsymbol{x})}$$

- Smooth version of threshold
- approximately linear around zero
- saturates for large and small values

# Representational power of MLP

## Representable functions

**boolean functions** any boolean function can be represented by some network with two layers of units

**continuous functions** every bounded continuous function can be approximated with arbitrary small error by a network with two layers of units (sigmoid hidden activation, linear output activation)

**arbitrary functions** any function can be approximated to arbitrary accuracy by a network with three layers of units (sigmoid hidden activation, linear output activation)

# Shallow vs deep architectures: Boolean functions

## Conjunctive normal form (CNF)

- One neuron for each clause (OR gate), with negative weights for negated literals
- One neuron at the top (AND gate)

## PB: number of gates

- Some functions require an exponential number of gates!! (e.g. parity function)
- Can be expressed with linear number of gates with a *deep network* (e.g. combination of XOR gates)

# Training MLP

## Stochastic gradient descent

- Training error for example $(x, y)$ (e.g. regression):

$$E(W) = \frac{1}{2}(y - f(x))^2$$

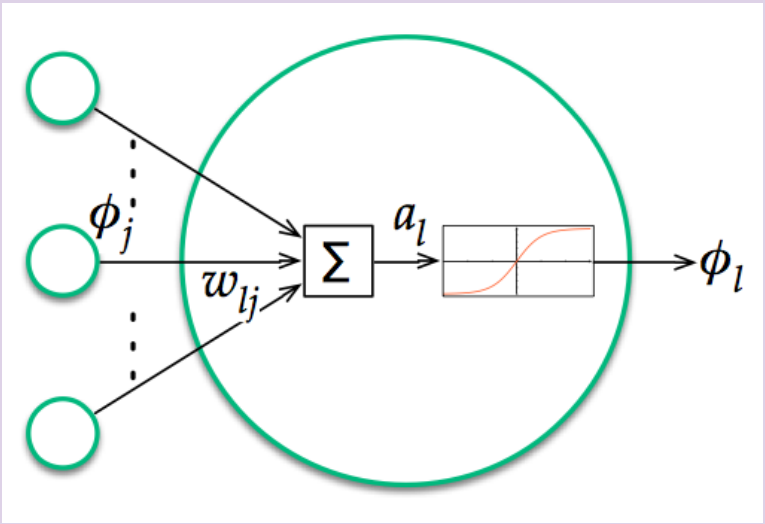
- Gradient update ($\eta$ learning rate)

$$w_{lj} = w_{lj} - \eta \frac{\partial E(W)}{\partial w_{lj}}$$

## Backpropagation

Use chain rule for derivation:

$$\frac{\partial E(W)}{\partial w_{lj}} = \underbrace{\frac{\partial E(W)}{\partial a_l}}_{\delta_l} \frac{\partial a_l}{\partial w_{lj}} = \delta_l \phi_j$$



# Training MLP

## Output units

- Delta is easy to compute on output units.
- E.g. for regression with sigmoid outputs:

$$\begin{aligned}\delta_o &= \frac{\partial E(W)}{\partial a_o} = \frac{\partial \frac{1}{2}(y - f(x))^2}{\partial a_o} \\ &= \frac{\partial \frac{1}{2}(y - \sigma(a_o))^2}{\partial a_o} = -(y - \sigma(a_o))\frac{\partial \sigma(a_o)}{\partial a_o} \\ &= -(y - \sigma(a_o))\sigma(a_o)(1 - \sigma(a_o))\end{aligned}$$

## Derivative of sigmoid

$$
\begin{aligned}
\frac{\partial \sigma(x)}{\partial x} &= \frac{\partial}{\partial x}\frac{1}{1+\exp(-x)} \\
&= -(1+\exp(-x))^{-2}\frac{\partial}{\partial x}(1+\exp(-x)) \\
&= -(1+\exp(-x))^{-2} - \exp(-2x)\frac{\partial \exp(x)}{\partial x} \\
&= (1+\exp(-x))^{-2}\exp(-2x)\exp(x) \\
&= \frac{1}{1+\exp(-x)}\frac{\exp(-x)}{1+\exp(-x)} \\
&= \frac{1}{1+\exp(-x)}(1-\frac{1}{1+\exp(-x)}) \\
&= \sigma(x)(1-\sigma(x))
\end{aligned}
$$

# Training MLP

## Hidden units

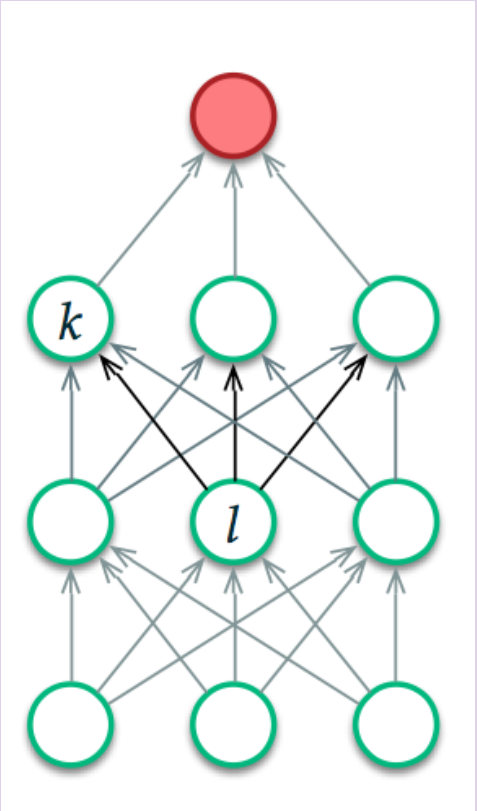
Consider contribution to error through all outer connections (again sigmoid activation):

$$\delta_l = \frac{\partial E(W)}{\partial a_l} = \sum_{k \in \mathrm{ch}[l]} \frac{\partial E(W)}{\partial a_k} \frac{\partial a_k}{\partial a_l}$$

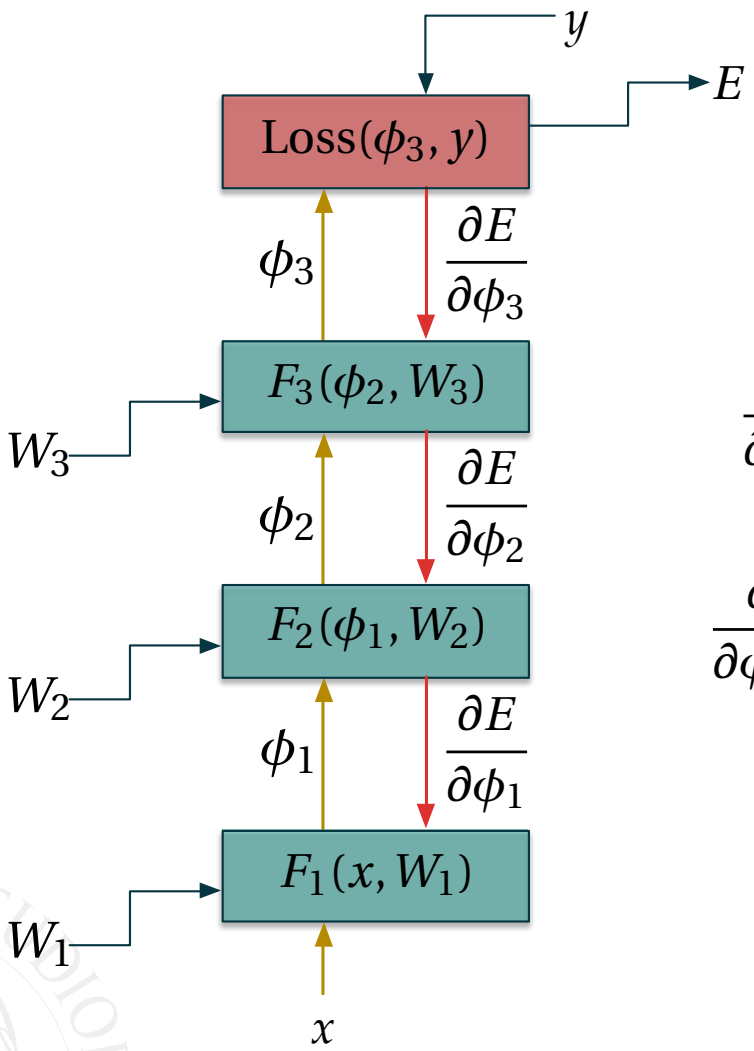
$$= \sum_{k \in \mathrm{ch}[l]} \delta_k \frac{\partial a_k}{\partial \phi_l} \frac{\partial \phi_l}{\partial a_l}$$

$$= \sum_{k \in \mathrm{ch}[l]} \delta_k w_{kl} \frac{\partial \sigma(a_l)}{\partial a_l}$$

$$= \sum_{k \in \mathrm{ch}[l]} \delta_k w_{kl} \sigma(a_l)(1 - \sigma(a_l))$$



# Deep architectures: modular structure




$$\phi_j = F_j(\phi_{j-1}, W_j)$$

$$\frac{\partial E}{\partial W_j} = \frac{\partial E}{\partial \phi_j}\frac{\partial \phi_j}{\partial W_j} = \frac{\partial E}{\partial \phi_j}\frac{\partial F_j(\phi_{j-1}, W_j)}{\partial W_j}$$

$$\frac{\partial E}{\partial \phi_{j-1}} = \frac{\partial E}{\partial \phi_j}\frac{\partial \phi_j}{\partial \phi_{j-1}} = \frac{\partial E}{\partial \phi_j}\frac{\partial F_j(\phi_{j-1}, W_j)}{\partial \phi_{j-1}}$$

# Remarks on backpropagation

## Local minima

- The error surface of a multilayer neural network can contain several minima
- Backpropagation is only guaranteed to converge to a *local* minimum
- Heuristic attempts to address the problem:
  - use stochastic instead of true gradient descent
  - train multiple networks with different random weights and average or choose best
  - many more..

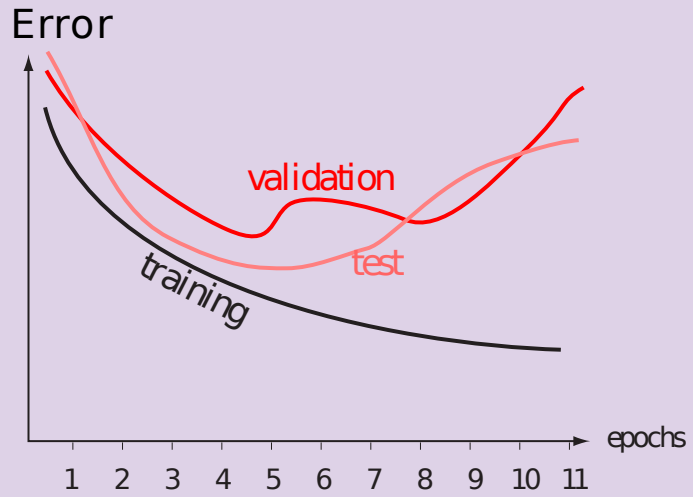
## Note

- Training kernel machines requires solving *quadratic* optimization problems $\rightarrow$ global optimum guaranteed
- Deep networks are more expressive in principle, but harder to train

# Stopping criterion and generalization

## Stopping criterion

- How can we choose the training termination condition?
- Overtraining the network increases possibility of *overfitting* training data
- Network is initialized with small random weights $\Rightarrow$ very simple decision surface
- Overfitting occurs at later iterations, when increasingly complex surfaces are being generated
- Use a separate *validation* set to estimate performance of the network and choose when to stop training



# Training deep architectures

## PB: Vanishing gradient

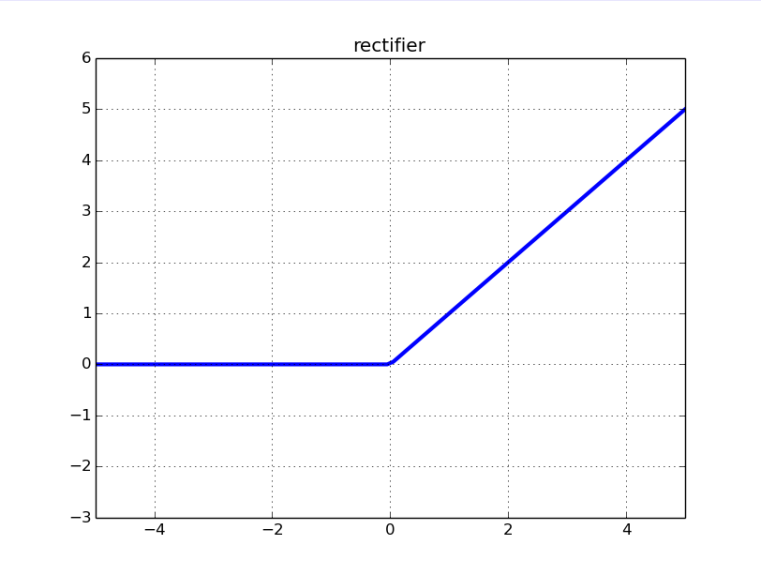
- Error gradient is backpropagated through layers
- At each step gradient is multiplied by derivative of sigmoid: very small for saturated units
- Gradient vanishes in lower layers
- Difficulty of training deep networks!!

# Tricks of the trade

## Few simple suggestions

- Do not initialize weights to zero, but to small random values around zero
- Standardize inputs ($x' = (x - \mu_x)/\sigma_x$) to avoid saturating hidden units
- Randomly shuffle training examples before each training epoch

# Tricks of the trade: activation functions




## Rectifier

$$f(x) = max(0, \boldsymbol{w}^T \boldsymbol{x})$$

- Linearity is nice for learning
- Saturation (as in sigmoid) is bad for learning (gradient vanishes $\rightarrow$ no weight update)
- Neuron employing rectifier activation called rectified linear unit (ReLU)

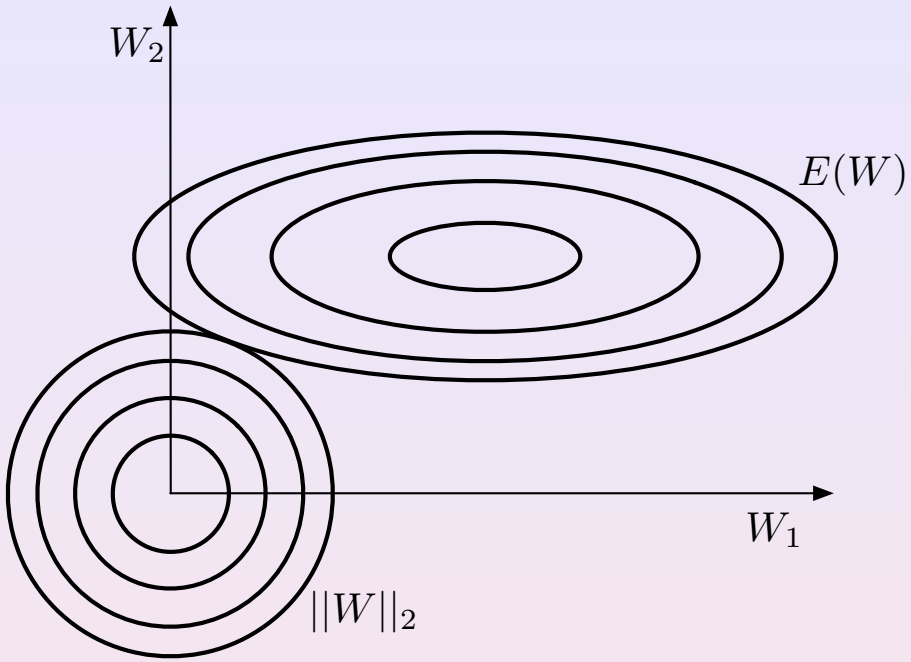
# Tricks of the trade: loss functions

## Cross entropy

$$E(W) = -\sum_{(\boldsymbol{x},y)\in\mathcal{D}} y \log f(x) + (1-y)\log(1-f(x))$$

- Minimize cross entropy of network output wrt targets
- Useful for binary classification tasks
- Model target function as probability that output is one (use sigmoid for output layer)
- Corresponds to maximum likelihood learning
- Log removes saturation effect of sigmoid (helps optimization)
- Can be generalized to multiclass classification (use softmax for output layer)

# Tricks of the trade: regularization

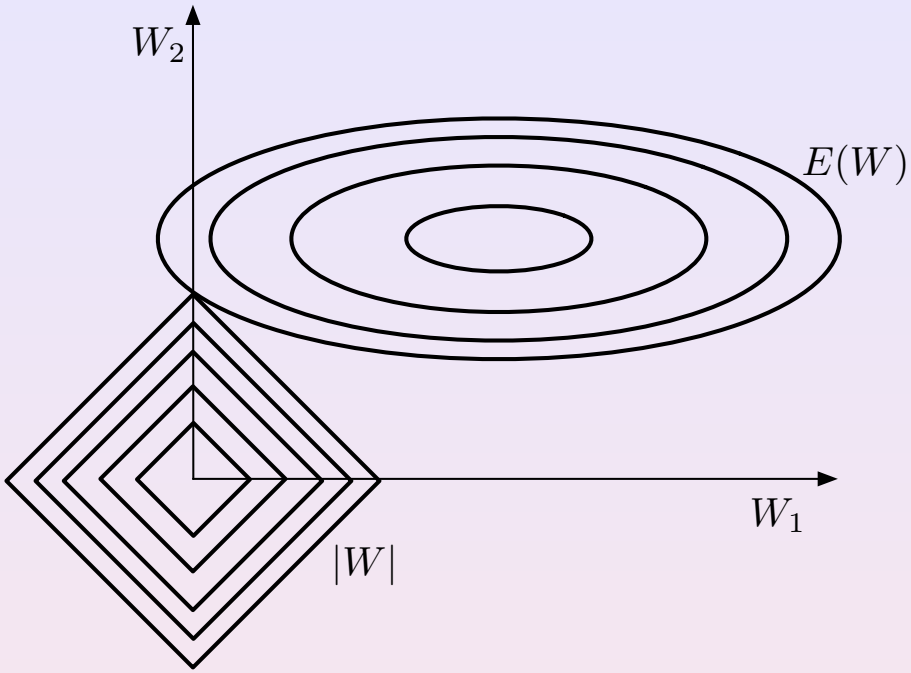



## 2-norm regularization

$$J(W) = E(W) + \lambda||W||_2$$

- Penalizes weights by Euclidean norm
- Weights with less influence on error get smaller values

# Tricks of the trade: regularization




## 1-norm regularization

$$J(W) = E(W) + \lambda |W|$$

- Penalizes weights by sum of absolute values
- Encourages less relevant weights to be exactly zero (sparsity inducing norm)

# Tricks of the trade: initialization

## Suggestions

- Randomly initialize weights (for breaking *symmetries* between neurons)
- Carefully set initialization range (to preserve forward and backward variance)

$$W_{ij} \sim U(-\frac{\sqrt{6}}{\sqrt{n+m}}, \frac{\sqrt{6}}{\sqrt{n+m}})$$

$n$ and $m$ number of inputs and outputs

- *Sparse* initialization: enforce a fraction of weights to be non-zero (encourages *diversity* between neurons)

# Tricks of the trade: gradient descent

## Batch vs Stochastic

- Batch gradient descent updates parameters after seeing all examples $\rightarrow$ too slow for large datasets
- Fully stochastic gradient descent updates parameters after seeing each example $\rightarrow$ objective too different from true one
- *Minibatch* gradient descent: update parameters after seeing a minibach of $m$ examples ($m$ depends on many factors, e.g. size of GPU memory)

# Tricks of the trade: gradient descent

## Momentum

$$
\begin{aligned}
v_{ji} &= \alpha v_{ji} - \eta \frac{\partial E(W)}{\partial w_{lj}} \\
\mathbf{w}_{ji} &= w_{ji} + v_{ji}
\end{aligned}
$$

- $0 \leq \alpha < 1$ is called **momentum**
- Tends to keep updating weights in the same direction
- Think of a ball rolling on an error surface
- Possible effects:
  - roll through small local minima without stopping
  - traverse flat surfaces instead of stopping there
  - increase step size of search in regions of constant gradient

# Tricks of the trade: adaptive gradient

## Decreasing learning rate

$$\eta_t = \begin{cases} (1 - \frac{t}{\tau})\eta_0 + \frac{t}{\tau}\eta_\tau & \text{if } t < \tau \\ \eta_\tau & \text{otherwise} \end{cases}$$

- Larger learning rate at the beginning for faster convergence towards attraction basin
- Smaller learning rate at the end to avoid oscillation close to the minimum

# Tricks of the trade: adaptive gradient

## Adagrad

$$
\begin{aligned}
r_{ji} &= r_{ji} + \left( \frac{\partial E(W)}{\partial w_{lj}} \right)^2 \\
\mathbf{w}_{ji} &= w_{ji} - \frac{\eta}{\sqrt{r_{ji}}} \frac{\partial E(W)}{\partial w_{lj}}
\end{aligned}
$$

- Reduce learning rate in steep directions
- Increase learning rate in gentler directions

## Problem

- Square gradient accumulated over all iterations
- For non-convex problems, learning rate reduction can be excessive/premature

# Tricks of the trade: adaptive gradient

## RMSProp

$$
\begin{aligned}
r_{ji} &= \rho r_{ji} + (1-\rho)\left(\frac{\partial E(W)}{\partial w_{lj}}\right)^2 \\
\mathbf{w}_{ji} &= w_{ji} - \frac{\eta}{\sqrt{r_{ji}}}\frac{\partial E(W)}{\partial w_{lj}}
\end{aligned}
$$

- Exponentially decaying accumulation of squared gradient ($0 < \rho < 1$)
- Avoids premature reduction of Adagrad
- Adagrad-like behaviour when reaching convex bowl

# Tricks of the trade: batch normalization

## Covariate shift problem

- Covariate shift problem is when the input distribution to your model changes over time (and the model does not adapt to the change)
- In (very) deep networks, *internal* covariate shift takes place among layers when they get updated by backpropagation

# Tricks of the trade: batch normalization

## Solution (sketch)

- Normalize each node activation (input to activation function) by its batch statistics

$$\hat{x}_i = \frac{x_i - \mu_B}{\sigma_B}$$

where:

  - $x$ is the activation of an arbitrary node in an arbitrary layer
  - $\mathcal{B} = \{x_1, \ldots, x_m\}$, is a batch of values for that activation
  - $\mu_B, \sigma_B^2$ are batch mean and variance

- Scale and shift each activation with adjustable parameters ($\gamma$ and $\beta$ become part of the network parameters)

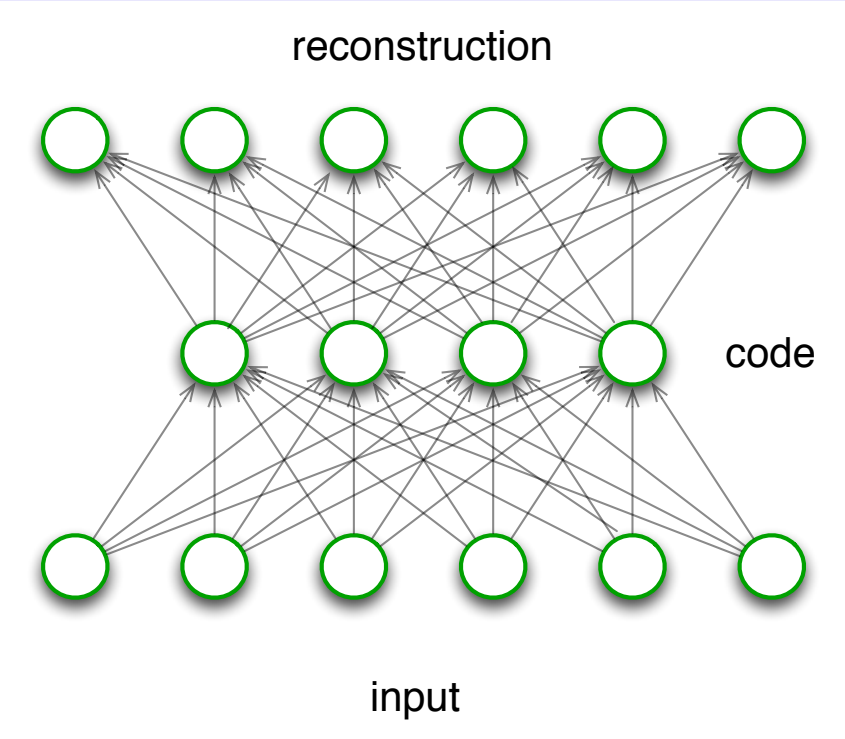
$$y_i = \gamma \hat{x}_i + \beta$$

# Tricks of the trade: batch normalization

## Advantages

- More robustness to parameter initialization
- Allows for faster learning rates without divergence
- Keeps activations in non-saturated region even for saturating activation functions
- Regularizes the model

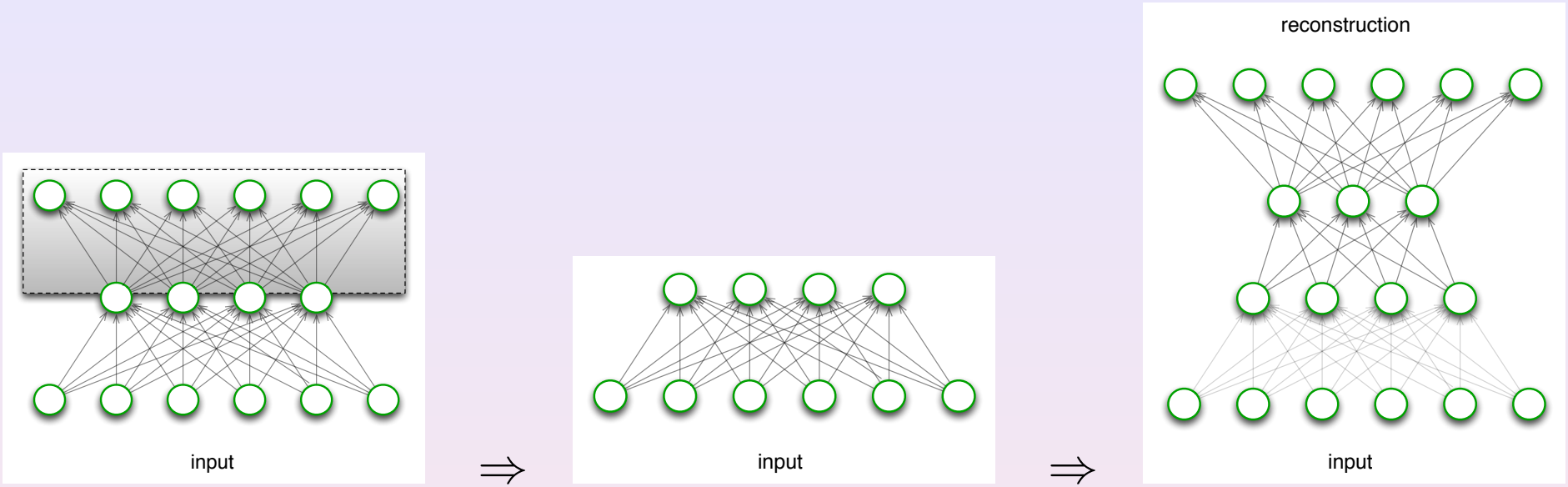
# Tricks of the trade: layerwise pre-training




## Autoencoder

- train shallow network to *reproduce* input in the output
- learns to map inputs into a sensible hidden representation (*representation learning*)
- can be done with unlabelled examples (*unsupervised learning*)

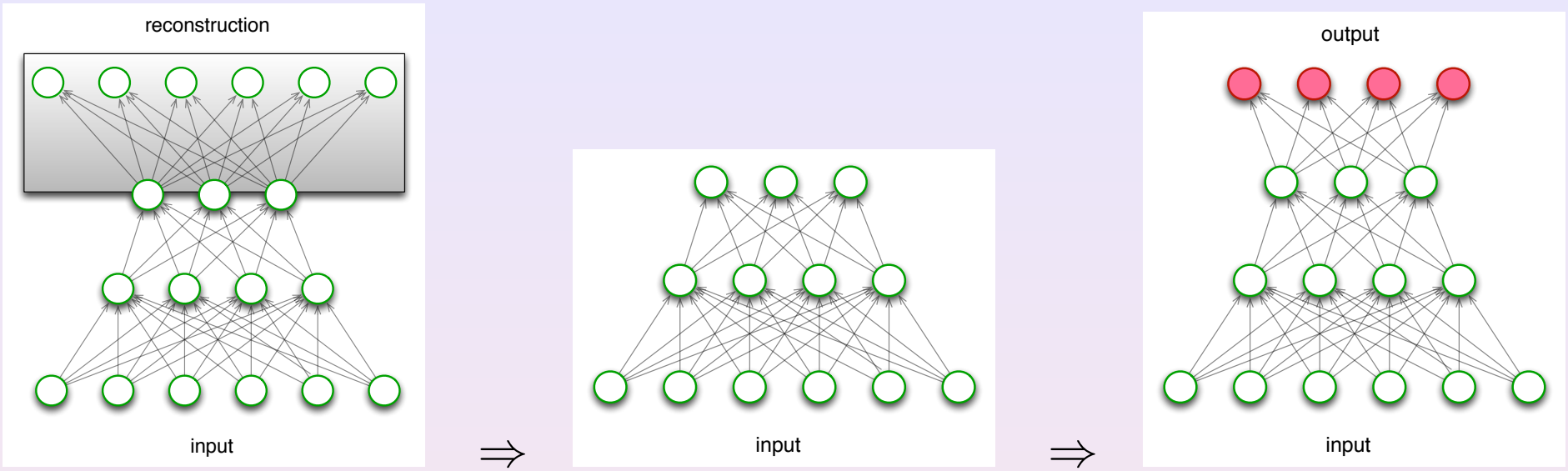
# Tricks of the trade: layerwise pre-training




## Stacked autoencoder

- repeat:
  1. discard output layer
  2. freeze hidden layer weights
  3. add another hidden + output layer
  4. train network to reproduce input

# Tricks of the trade: layerwise pre-training




## global refinement

- discard autoencoder output layer
- add appropriate output layer for supervised task (e.g. one-hot encoding for multiclass classification)
- learn output layer weights + refine all internal weights by backpropagation algorithm

# Tricks of the trade: layerwise pre-training

## Modern pre-training

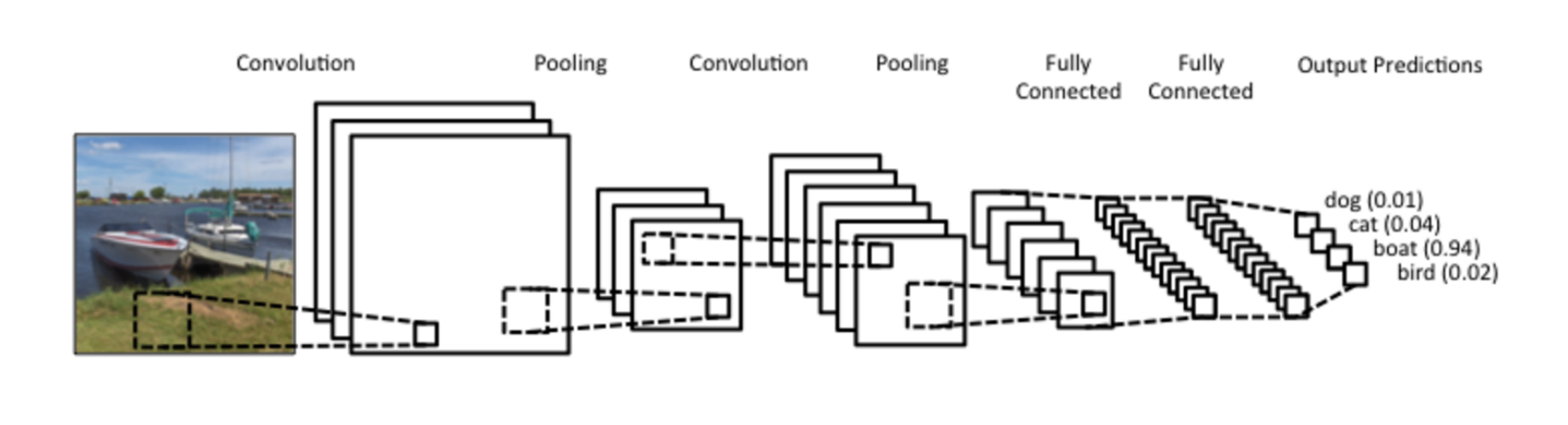
- Supervised pre-training: layerwise training with actual labels
- Transfer learning: train network on similar task, discard last layers and retrain on target task
- Multi-level supervision: auxhiliary output nodes at intermediate layers to speed up learning

# Popular deep architectures

## Many different architectures

- *convolutional networks* for exploiting local correlations (e.g. for images)
- *recurrent* and *recursive* networks for collective predictions (e.g. sequential labelling)
- *deep Boltzmann machines* as *probabilistic generative models* (can also generate new instances of a certain class)
- *generative adversarial networks* to generate new instances as a *game between discriminator and generator*

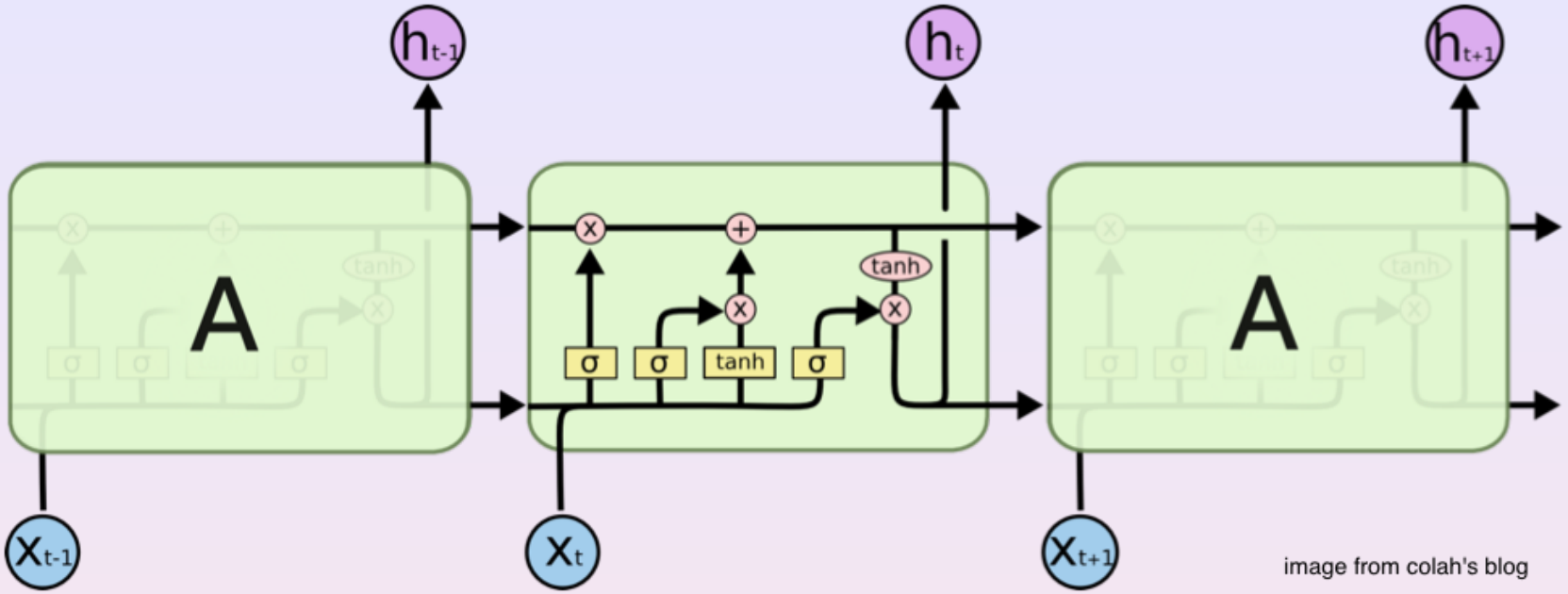
# Convolutional networks (CNN)




## Location invariance + compositionality

- *convolution filters* extracting local features
- *pooling* to provide invariance to local variations
- *hierarchy of filters* to compose complex features from simpler ones (e.g. pixels to edges to shapes)
- *fully connected layers* for final classification

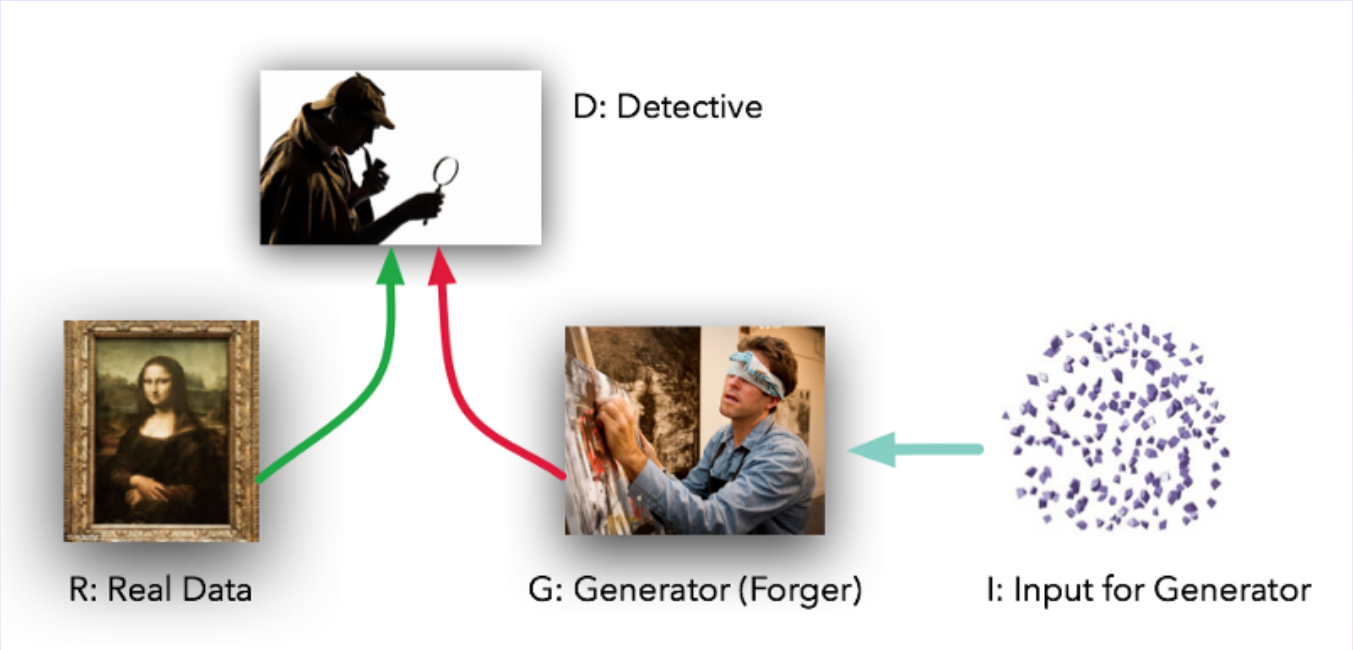
# Long Short-Term Memory Networks (LSMT)




## Recurrent computation with selective memory

- *Cell state* propagated along chain
- *Forget gate* selectively forgets parts of the cell state
- *Input gate* selectively chooses parts of the candidate for cell update
- *Output gate* selectively chooses parts of the cell state for output

# Generative Adversarial Networks (GAN)




## Generative learning as an adversarial game

- A *generator* network learns to generate items (e.g. images) from random noise
- A *discriminator* network learns to distinguish between real items and generated ones
- The two networks are jointly learned (adversarial game)
- No human supervision needed!!

# References

## Libraries

- TensorFlow (https://www.tensorflow.org/)
- Keras (https://keras.io/)
- PyTorch (http://pytorch.org/)
- Caffe (http://caffe.berkeleyvision.org/)

## Literature


- Yoshua Bengio, *Learning Deep Architectures for AI*, Foundations & Trends in Machine Learning, 2009.
- Ian Goodfellow, Yoshua Bengio and Aaron Courville, *Deep Learning*, Book in preparation for MIT Press, 2016 (`http://www.deeplearningbook.org/`)
- Christopher Olah, *Understanding LSTM Networks* (`http://colah.github.io/posts/2015-08-Understanding-LSTMs/`)