UPPSALA UNIVERSITET

Natural Computation Methods in Machine Learning (NCML)

Lecture 6: More extensions and variants. Data representation. Exploting prior knowledge.

Automatic sizing

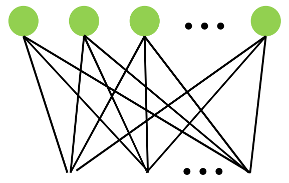
- We want to minimize the number of hidden nodes
  - ➢ less parameters
  - ➢ less risk of overfitting (i.e. better generalization)
  - ➢ faster recall (fewer terms in weighted sums)
  - ➢ faster training (fewer parameters to update)
- Two approaches
  - – Start with a large network and prune it
  - – Start with a small network and let it grow
  - – (or a combination of both, of course)
  - – Examples in the book (chapter 7)

Weight decay
Pruning technique

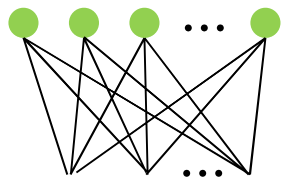
- Let each weight strive for 0
- Simple implementation
  - After each weight update, update again by:
    $$w^{new} := (1 - \epsilon)w^{old}$$
  - ε is a forgetfulness constant, $0 \leq \varepsilon < 1$
- Effect: Unnecessary weights end up close to 0
  - Remove them and retrain with the new network
  - Repeat until all (remaining) weights seem to be necessary
- Not only used for pruning! (actually, usually not)
  - Works as a regularizer also if weights are not removed!
  - Restricts the network
  - Keeps weights small → less numerical issues

Upstart
Growing technique

- Self-sizing method for classification (only)
- Not used much, but illustrates the idea of more useful variants
- Consider a single layer binary perceptron



- No hidden nodes ➔ only linearly separable problems can be solved
- Train it anyway, and note for which input vectors each output node makes mistakes



Upstart

Growing technique

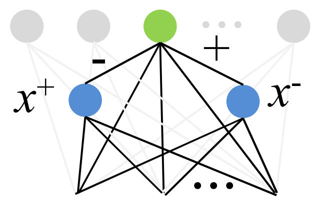
- An output node can make two kinds of mistakes:

  E^+: $y=1$, should be 0 (weighted sum, S, too large)

  E^-: $y=0$, should be 1 (weighted sum, S, too small)

- For each output, create two children, x^+ and x^-
- Train them, separately, to recognize the cases for which the parent made mistakes:
  - This is another classification problem (smaller)
  - x^+ is trained to recognize the input vectors for which the parent made E^+ mistakes
  - x^- is trained to recognize the input vectors for which the parent made E^- mistakes

UPPSALA UNIVERSITET



Upstart
Growing technique

- x^+ will output a 1 if the parent's weighted sum was too large
  - Connect it to the parent through a large negative weight
- x^- will output a 1 if the parent's weighted sum was too small
  - Connect it to the parent through a large positive weight
- If the children can't solve <u>their</u> problem, create new children for them ...
- Result: A finite "tree" of neurons, all connected to the same inputs

What happens if we do this for XOR?

XOR with Upstart

XOR

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

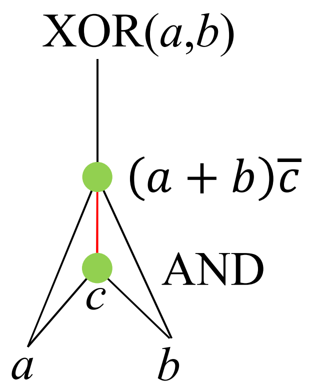
$\Rightarrow$

OR

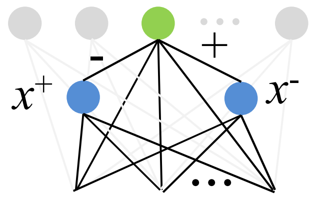
a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

E^+

$\Rightarrow$



There are no E^- examples and therefore no x^- node



Upstart
Some notes

- Upstart is divide-and-conquer
  - Each new subproblem is smaller than the parent's
- Could have solved Minsky & Papert's credit assignment problem (1969)
  - without replacing the step function
- Severe risk of overfitting, though
  - and limited to classification
- More general variant: Fahlman's Cascade Correlation Algorithm
  - Can be described as a generalized version of Upstart

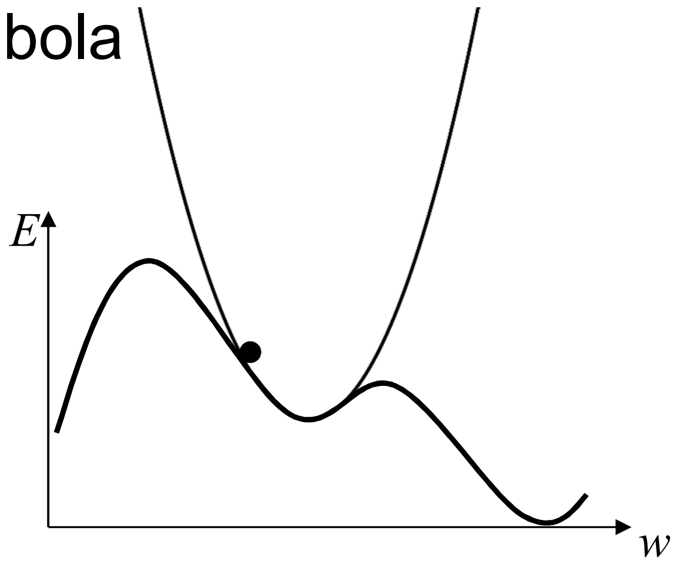
2nd order methods

- Faster training by considering 2nd order info
  - i.e. derivatives of derivatives (Hessians)
  - i.e. how slopes change over time
    - (RPROP is 1st order – it checks **if** slopes change, not how)
- Classic example: Newton-Raphson's method
- In effect – try to guess where the minimum is and jump directly there
- 2nd order neural network training algorithms have existed for a long time, but are not used much. Why?

Quickprop

Scott Fahlman, 1988 :-)

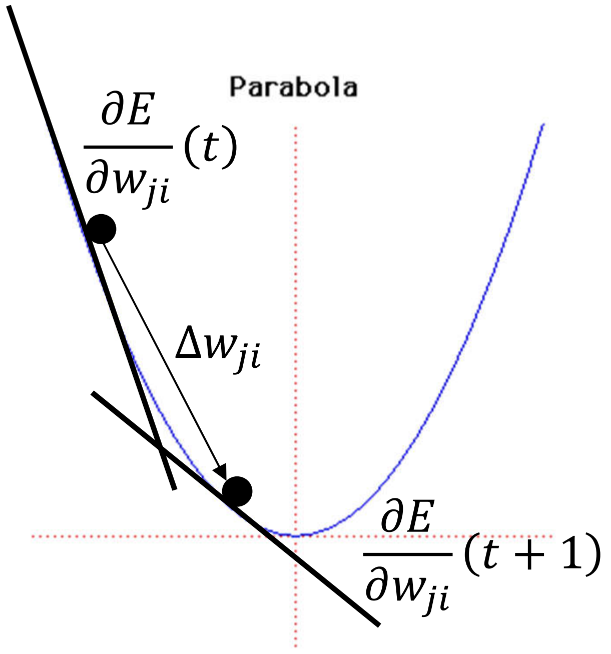
- Second order method
  - based on Newton-Raphson's method
- Requires epoch learning
- Assume, for every weight w_{ji}, that:
  - The error surface (the 'landscape') can be approximated locally by a parabola
  - The change in slope $\frac{\partial E}{\partial w_{ji}}$ from the previous step, is only due to the change of w_{ji}



Quickprop

Scott Fahlman, 1988

- Then, the current and previous slope, together with the latest weight change, can be used to define a parabola
- Jump towards its minimum
- Iterations compensate for errors introduced by the assumptions
- Can be extremely fast, but also very sensitive to choice of control parameters (hyper-parameters)


Parabola
$\frac{\partial E}{\partial w_{ji}}(t)$
Δw_{ji}
$\frac{\partial E}{\partial w_{ji}}(t+1)$

Why are 2nd order methods not more common?

- In an industrial/commercial setting, good enough is good enough
- Deep Learning has made 1st order methods popular (again)
  - Low complexity (w.r.t. the number of parameters)
    - Same complexity as just evaluating the function
    - For deep structures, this can be a big benefit
  - See Kingma & Ba, Adam: A Method for Stochastic Optimization, ICLR, 2015.
- The No-Free-Lunch theorem

UPPSALA UNIVERSITET

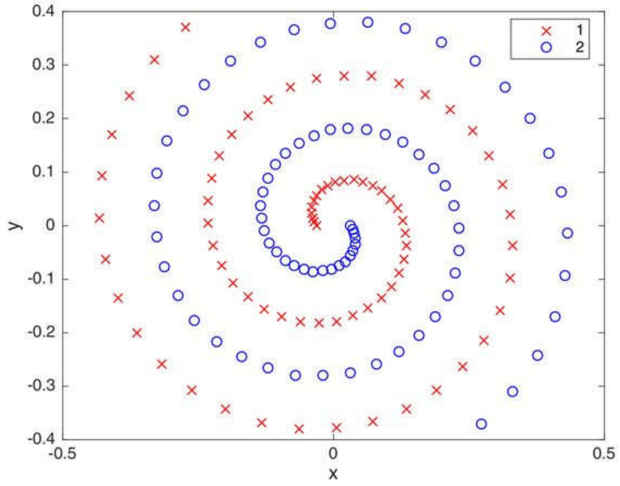
No-Free-Lunch theorem

Wolpert & Macready, 1997

- "Any two optimization algorithms are equivalent when their performance is averaged across all possible problems"
- Notes and implications:
  - "Any optimization algorithm" includes random search!
  - We are not trying to solve "all possible problems"!
    - For a subset of problems, one algorithm may still be best!
  - There is always a catch! All improvements <u>must</u> have drawback (and we should try to find what it is)
  - If your new algorithm is worse than another on a subset of problems, there <u>must</u> be another subset for which it is better!
    - Before discarding it, check if that other subset is interesting!

Preprocessing

- The choice of input and output representations is the problem!
  - This choice often decides if we will succeed or fail
  - Difficult problems can be made trivial
  - Simple problems can become unnecessarily hard
  - Example: (two-spirals)

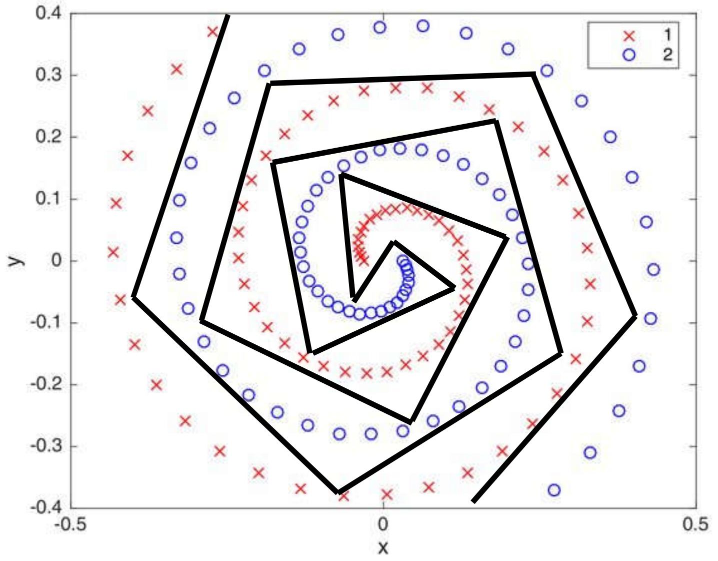


UPPSALA UNIVERSITET

Preprocessing

Two-spirals for a binary MLP: How many hidden nodes?

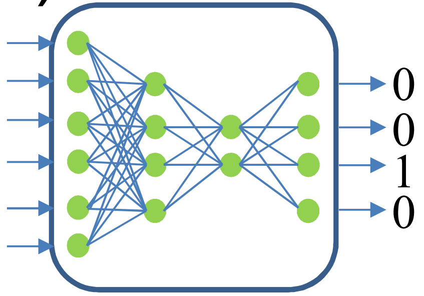
- 17 hidden nodes + one output node
  - In practice, requires two hidden layers
- or
- 1 single node!
  - The problem becomes linearly separable if we provide polar coordinates instead of Cartesian! (simple transform)



General advice on representations

Distribute them!

- In classification, use *one-hot encoding* of outputs (as many nodes as there are classes)
  - The network has to learn the encoding as well the original problem! (and this is probably the simplest)
  - Binary encoding would be more compact (→ fewer weights), but is <u>much</u> more difficult to learn (no use of generalization)
  - A network with one-hot encoded output values will approximate Bayesian probabilities, $P(C_i|\overline{x})$



Classification into 4 classes

General advice on representations

Distribute them!

- Input values can also be distributed!
- Look for numerical properties which can can be encoded as generalizable 'patterns'
- Example: Encoding integer inputs in the range [0,15]

7: 7/15 = 0.47	7: 0111	7:111111100000000
One real number, x_i in [0,1], for each integer, n_i, where $x_i = n_i/15$ (normalizing the integers)	Four binary inputs for each integer (a binary encoding of the integer)	15 binary inputs for each integer – where the integer n_i is encoded by setting the first n_i inputs to 1 and the rest to 0
OK	Worst!	Best!
(though it hides the fact that they are integer)	(the network has to learn to decode them)	(This is a pattern, and similar patterns represent similar values)

UPPSALA UNIVERSITET

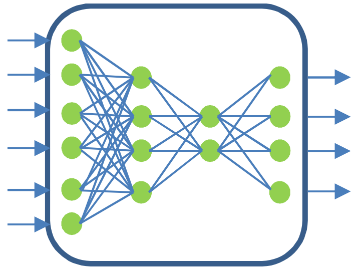
General advice on representations

Normalization / Scaling

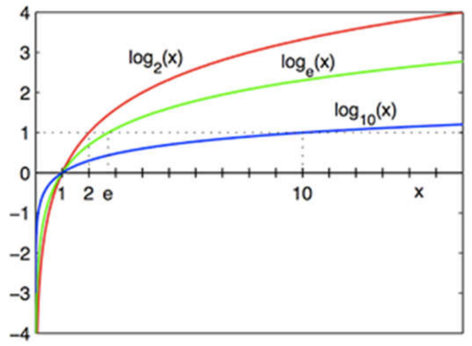
- Normalization of inputs is usually a good idea
  - to give inputs with small ranges a chance ...
  - but we may overcompensate and introduce new bias!

[0..10] → [0..1]

[-313..+4711] → [0..1]



- Values (input and targets) with extreme ranges may require scaling, e.g. using logarithms
  - Be aware that this may make the network more sensitive to small values than to large ones!

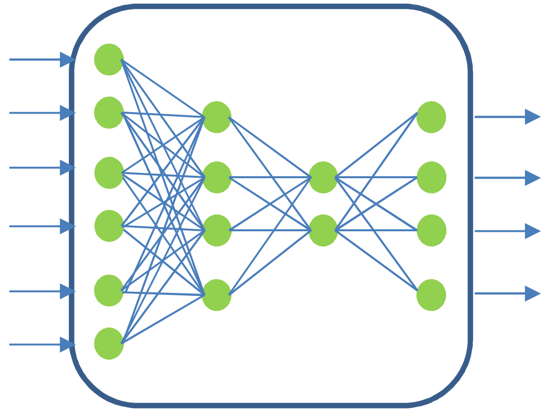


UPPSALA UNIVERSITET

General advice on representations

Exploit prior knowledge!

- Any known prior knowledge (statistical distributions, symmetries, etc) should be exploited in preprocessing!
  - Don't force the network to learn things already known!
- Example: Training a network on a function which you know is commutative w.r.t. to its inputs:
  - Order independent inputs = permutational symmetry = <u>very</u> difficult to learn!
  - Simple solution: Sort the inputs before presenting them!



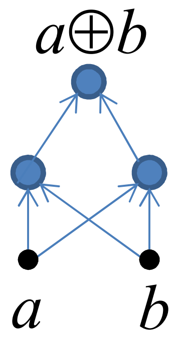
Exploiting prior knowledge

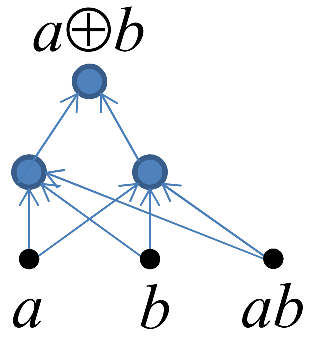
- Initial guess
  - Choice of initial weights (or how to randomize them)
- Known decomposition into subproblems
  - Preprocessing
  - Network Structure
  - Extra Output Learning (special case of Multitask learning)
- Constraints
  - If differentiable, we can add them as (Lagrangian) terms to the loss function, and then derive a new learning rule, as we did for Backprop
- Regions
  - Preprocessing
  - Extra Output Learning

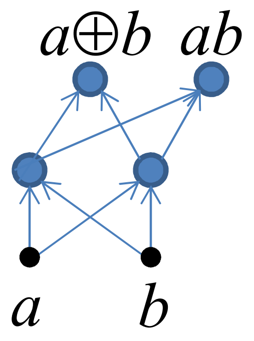
Extra Output Learning

A special case of Multitask Learning

- Some problems become much easier if we add extra <u>inputs</u> – more features
  - For example XOR which even becomes linearly separable, if we add an extra input $c = a$ AND b
- But the extra info then becomes *required*
- In Extra Output Learning we add the info as extra outputs/targets instead!





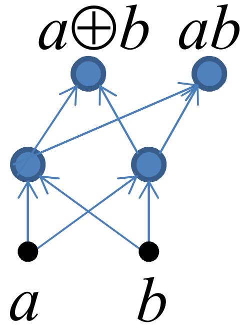


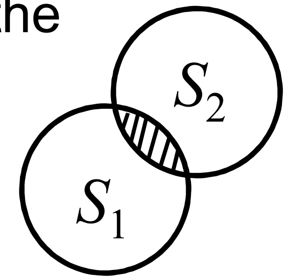
UPPSALA UNIVERSITET

Extra Output Learning

A special case of Multitask Learning

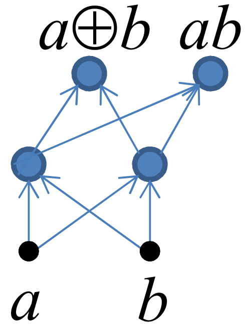
- How does this help?
  - The network is now trained on two functions simultaneously, f_1 and f_2
  - The two functions are correlated
    - AND is a subfunction of XOR so it has to be found by the hidden layer anyway
  - We're giving the network a hint!
  - Restricts freedom of hidden layer
    - S_i = set of models the hidden layer can find to solve f_i
    - Training on all of them at once, restricts this set to the intersection $S = S_1 \cap S_2 \cap \dots S_n$
    - So this is also a regularization technique!





Extra Output Learning

A special case of Multitask Learning

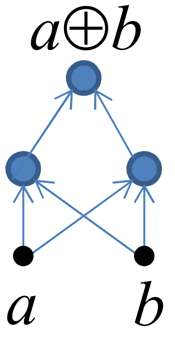


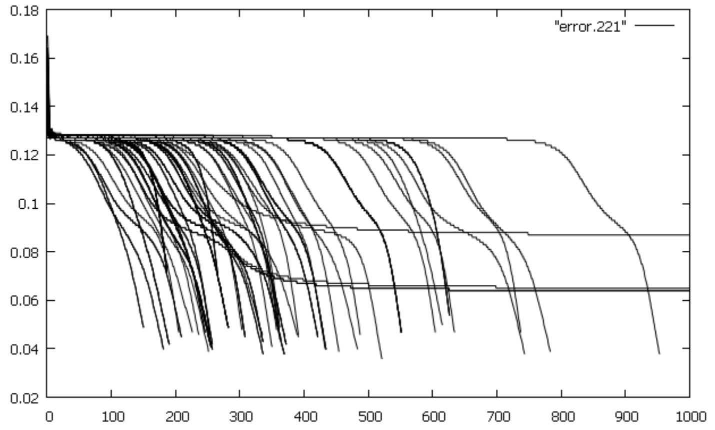
- Faster training
- Less risk of overfitting
- Better generalization
- Less variance in both training times and results
- Should reduce the required number of hidden nodes to <u>find</u> a solution, closer to the required number to <u>represent</u> it
- Once trained, the extra outputs can be removed!
  - We only need them during training

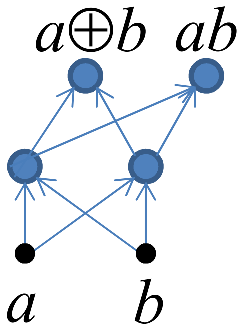
UPPSALA UNIVERSITET

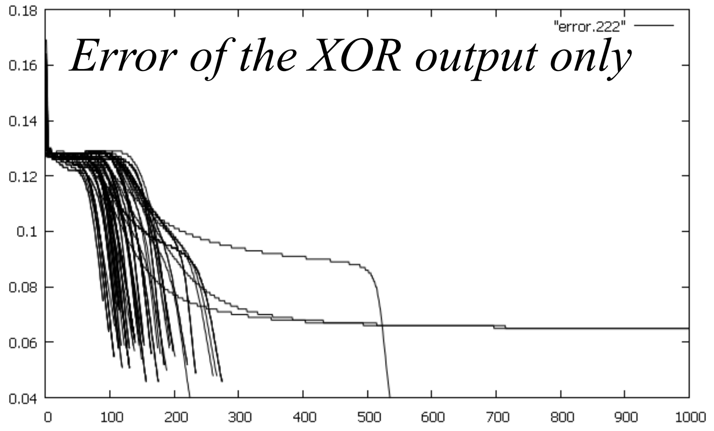
Extra Output Learning

Classification example (decomposition hint)



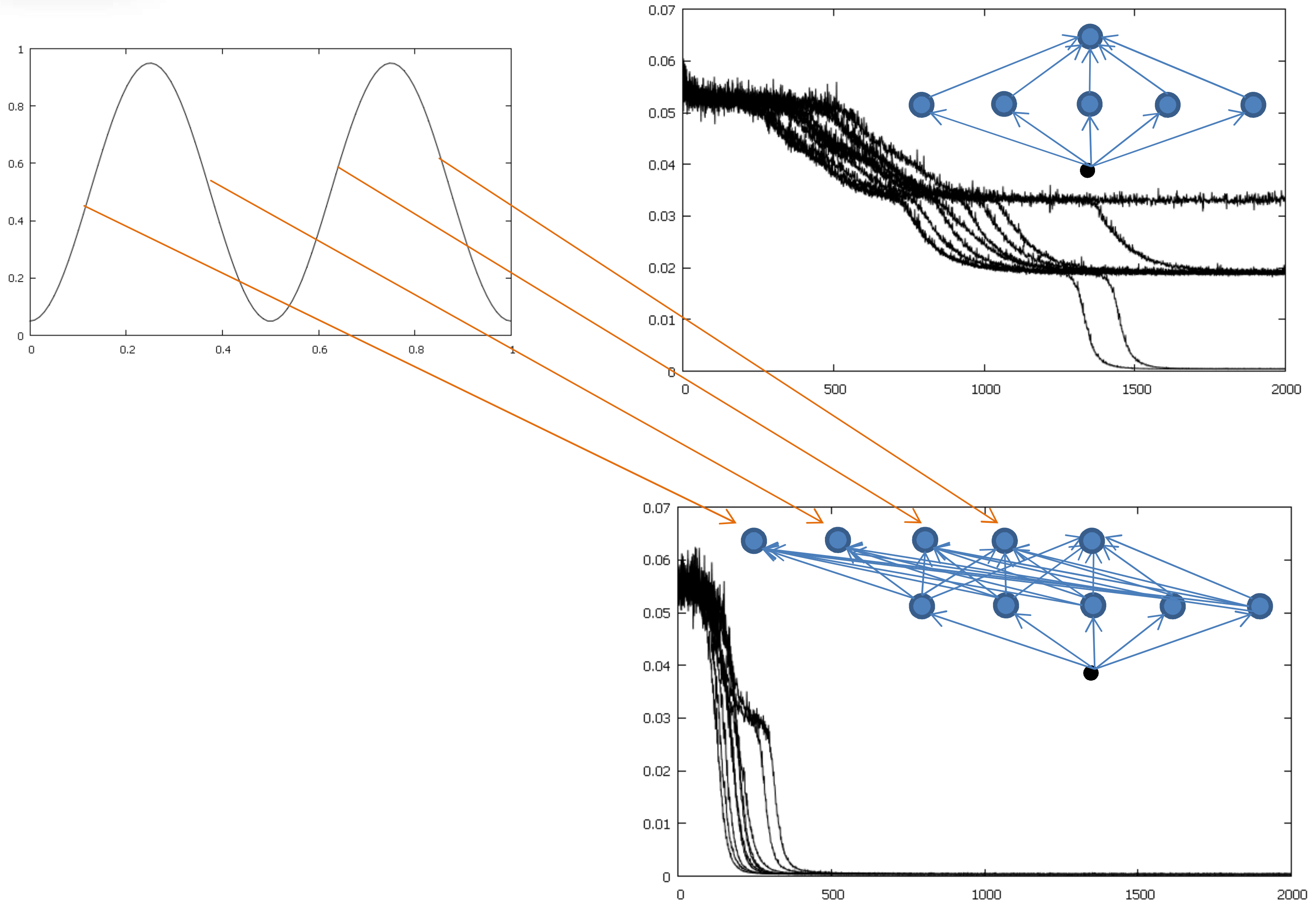






Extra Output Learning

Function approximation example (region hint)



Multitask Learning

Training on several tasks at once

- In Extra Output Learning, the hint is just hint
  - Only needed during training
  - to help the network solve the original problem
- Sometimes, we really want to solve several tasks, and, if correlated, we can use that
  - Self driving cars, for example
  - Lots of image recognition problems to solve
    - many of which are correlated
    - or based on the same features
  - So a common hidden layer should be good
  - This is indeed how it's done (by Tesla and others)