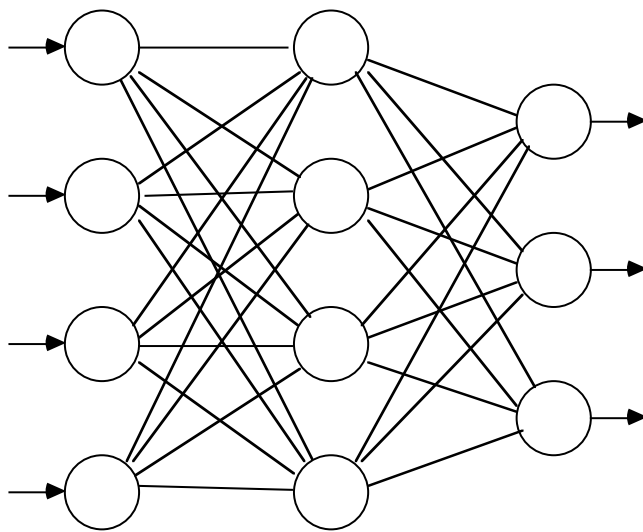


# MLPs with Backpropagation Learning

# Multilayer Nets? Linear Systems

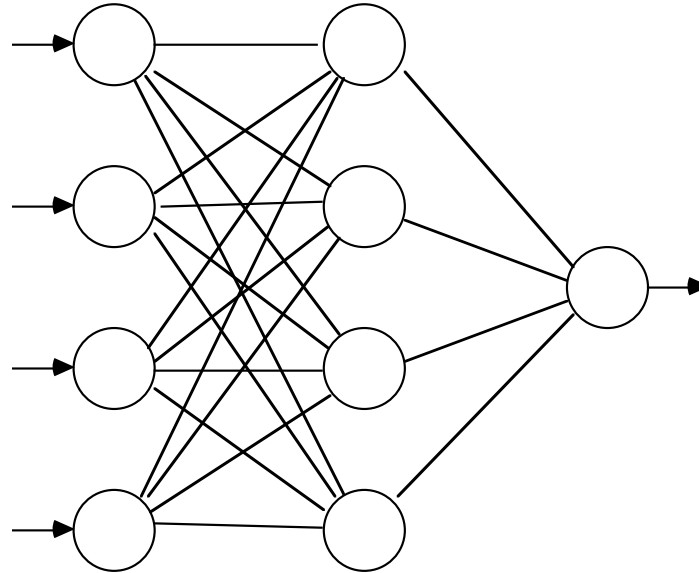
$$\mathbf{F}(c\mathbf{x}) = c\mathbf{F}(\mathbf{x})$$
$$\mathbf{F}(\mathbf{x}+\mathbf{y}) = \mathbf{F}(\mathbf{x}) + \mathbf{F}(\mathbf{y})$$



**I**      **N**      **M**      **Z**

$$\mathbf{Z} = (\mathbf{M}(\mathbf{N}\mathbf{I})) = (\mathbf{M}\mathbf{N})\mathbf{I} = \mathbf{P}\mathbf{I}$$

# Early Attempts Committee Machine



Randomly Connected  
(Adaptive)

Vote

Taking TLU

(non-adaptive)

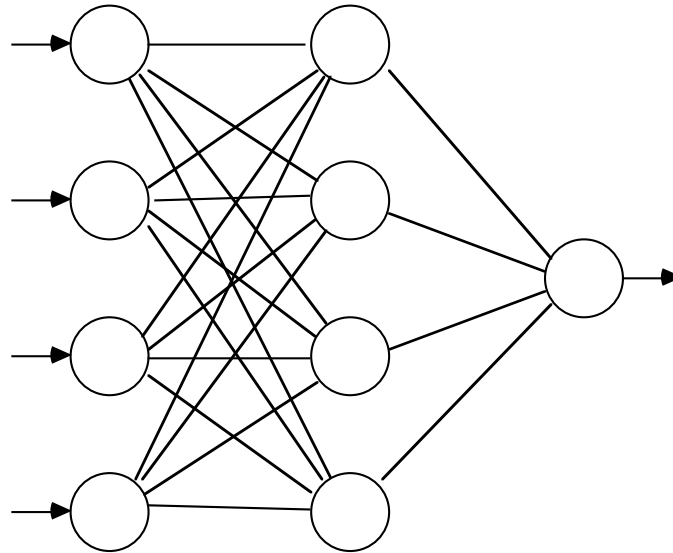
Majority Logic

"Least Perturbation Principle"

For each pattern, if incorrect, change just enough weights into internal units to give majority. Choose those closest to their threshold (LPP & changing undecided nodes)

# Perceptron (Frank Rosenblatt)

## Simple Perceptron



S-Units      A-units      R-units  
(Sensor)    (Association)    (Response)

Random to A-units  
fixed weights    adaptive

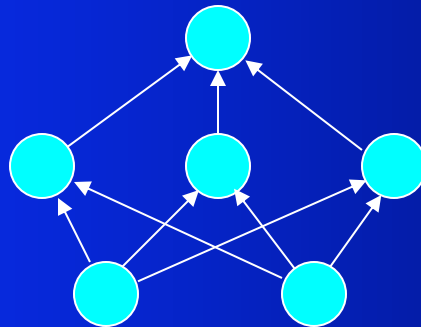
Variations on Delta rule learning  
Why S-A units?

# Backpropagation

- Rumelhart (1986), Werbos (74),..., explosion of neural net interest
- Multi-layer supervised learning
- Able to train multi-layer perceptrons (and other topologies)
- Commonly uses differentiable sigmoid function which is the smooth (squashed) version of the threshold function
- Error is propagated back through earlier layers of the network
- Very fast efficient way to compute gradients!

# Multi-layer Perceptrons trained with BP

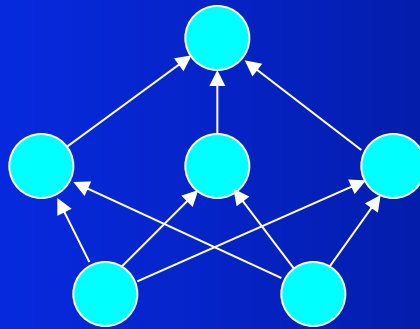
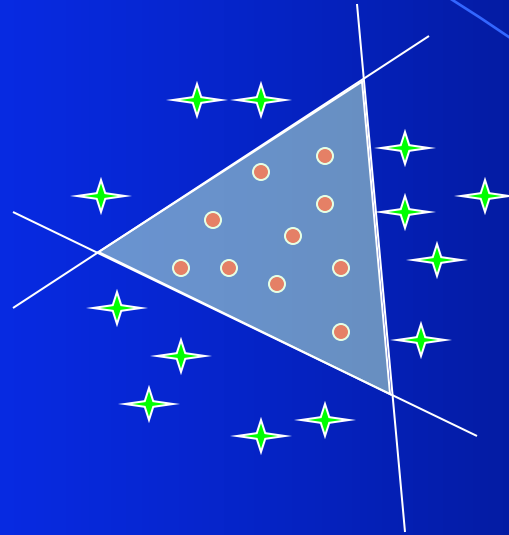
- Can compute arbitrary mappings
- Training algorithm less obvious
- First of many powerful multi-layer learning algorithms



# Responsibility Problem



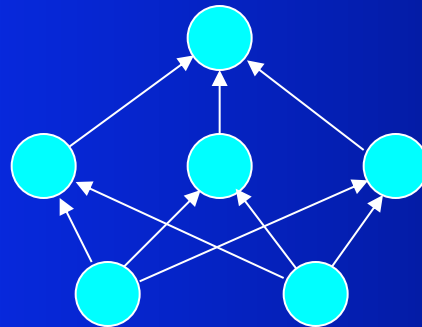
# Multi-Layer Generalization



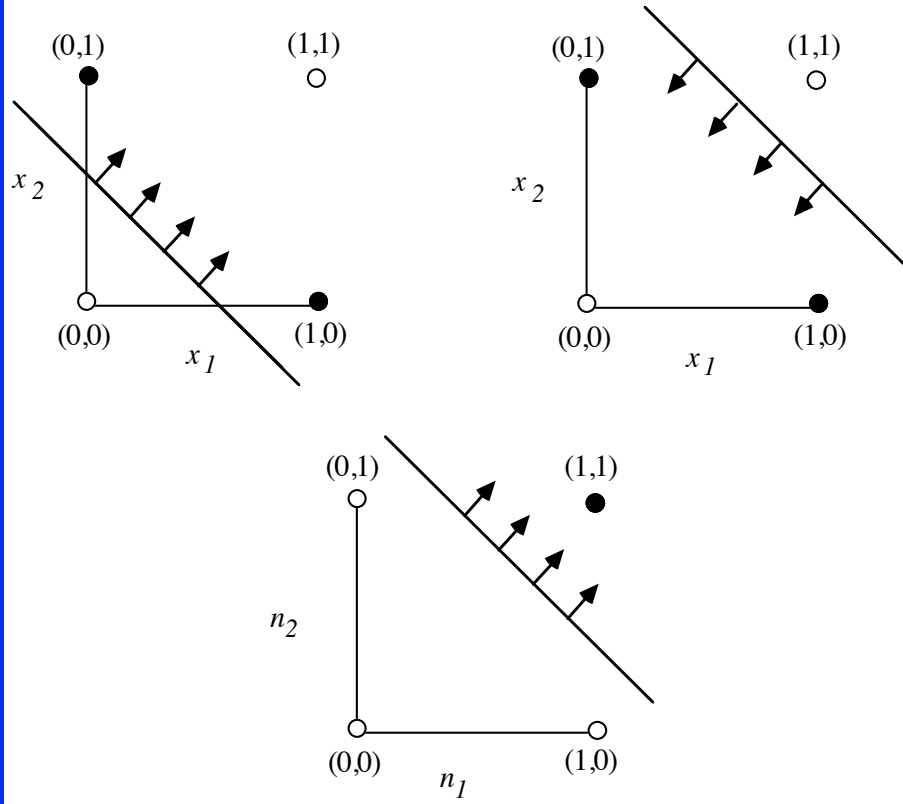
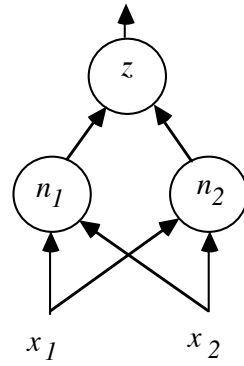


# Multilayer nets are universal function approximators

- Input, output, and arbitrary number of hidden layers

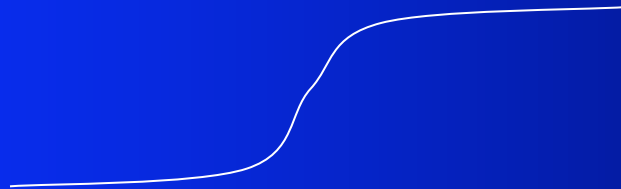


- 1 hidden layer sufficient for DNF representation of any Boolean function - One hidden node per positive conjunct, output node set to the “Or” function
- 2 hidden layers allow arbitrary number of labeled clusters
- 1 hidden layer sufficient to approximate all bounded continuous functions
- 1 hidden layer was the most common in practice, but recently... Deep networks show excellent results!

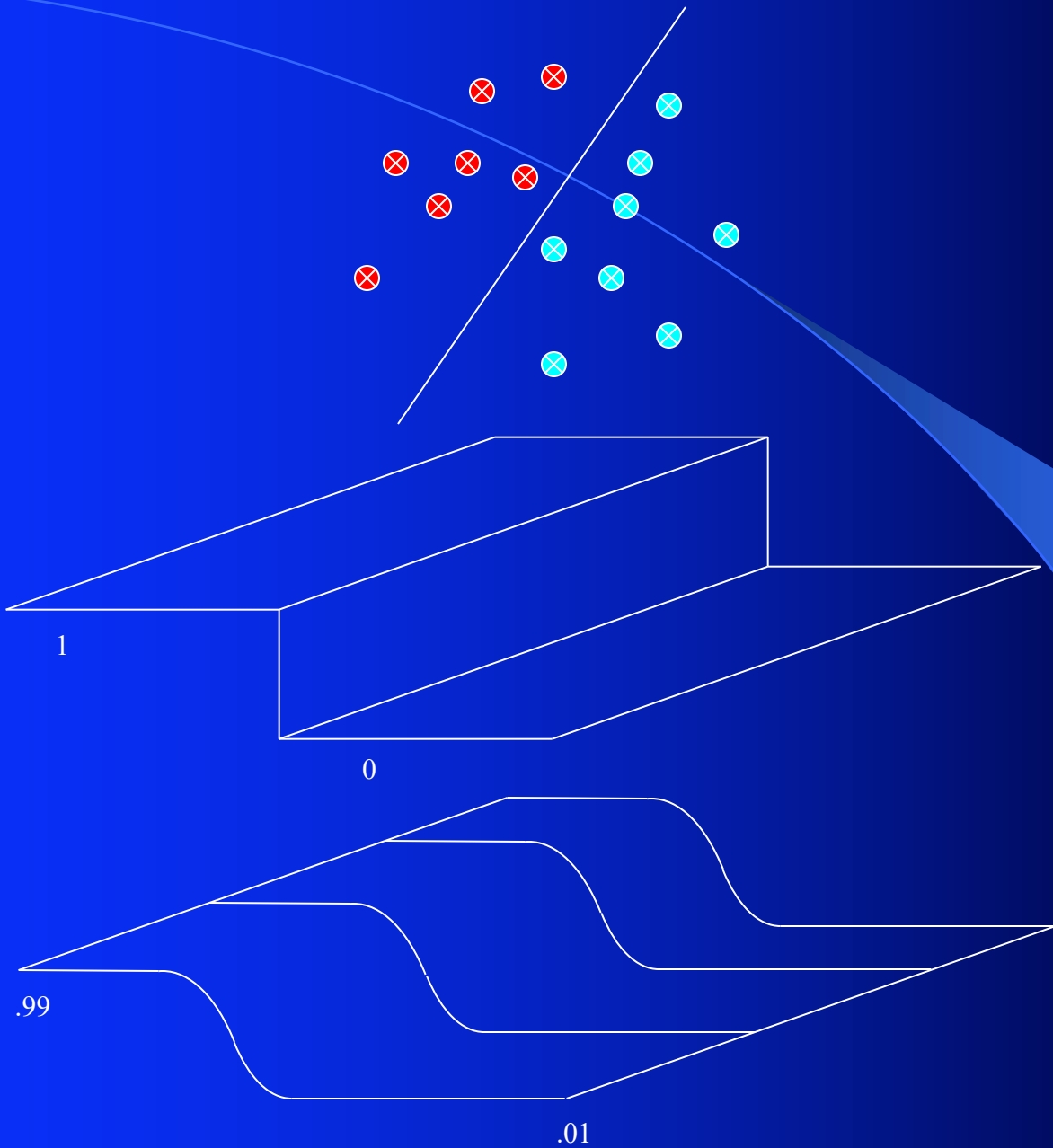


# Backpropagation

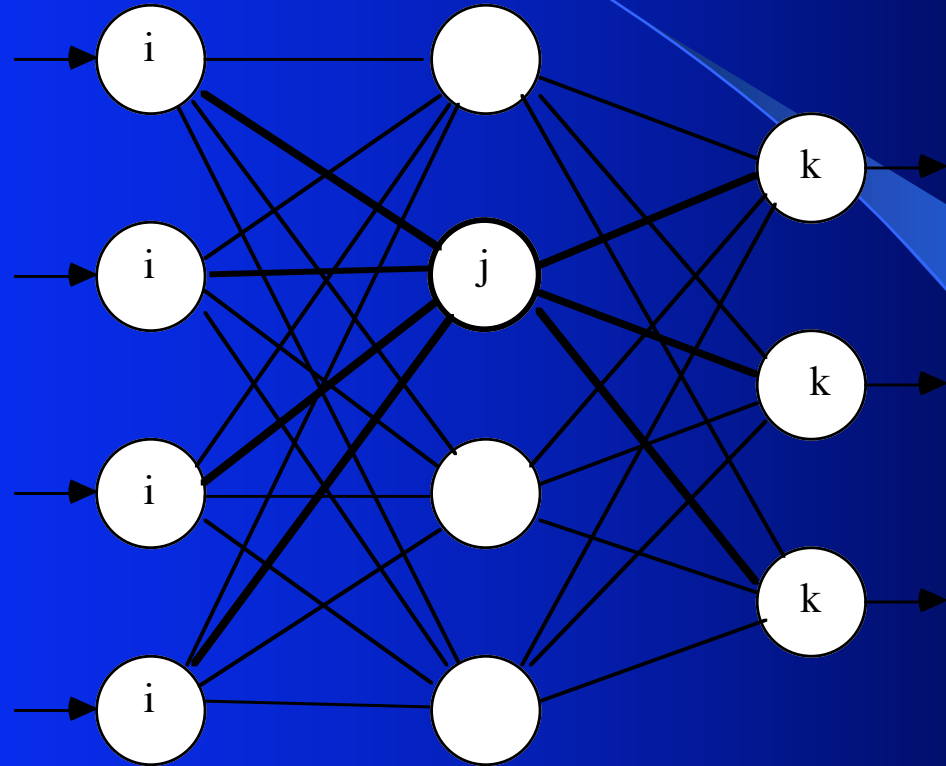
- Multi-layer supervised learner
- Gradient descent weight updates
- Sigmoid activation function (smoothed threshold logic)



- Backpropagation requires a differentiable activation function



# Multi-layer Perceptron (MLP) Topology



Input Layer    Hidden Layer(s)    Output Layer

# Backpropagation Learning Algorithm

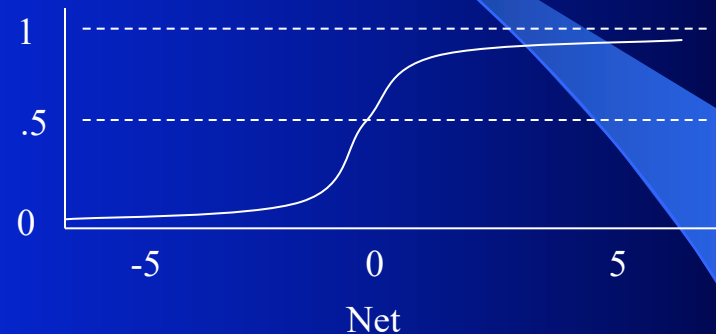
- Until Convergence (low error or other stopping criteria) do
  - Present a training pattern
  - Calculate the error of the output nodes (based on  $T - Z$ )
  - Calculate the error of the hidden nodes (based on the error of the output nodes which is propagated back to the hidden nodes)
  - Continue propagating error back until the input layer is reached
  - Then update all weights based on the standard delta rule with the appropriate error function  $\delta$

$$\Delta w_{ij} = C \delta_j Z_i$$

# Activation Function and its Derivative

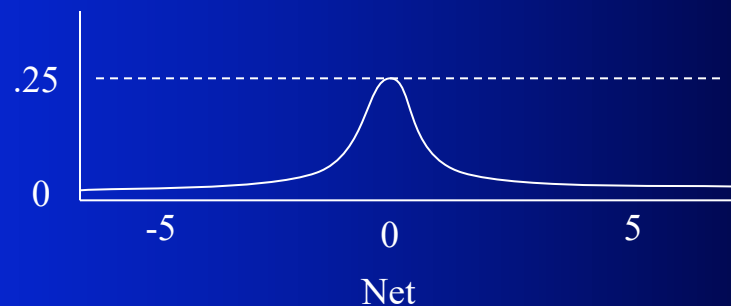
- Node activation function  $f(net)$  is commonly the sigmoid

$$Z_j = f(net_j) = \frac{1}{1 + e^{-net_j}}$$



- Derivative of activation function is a critical part of the algorithm

$$f'(net_j) = Z_j(1 - Z_j)$$

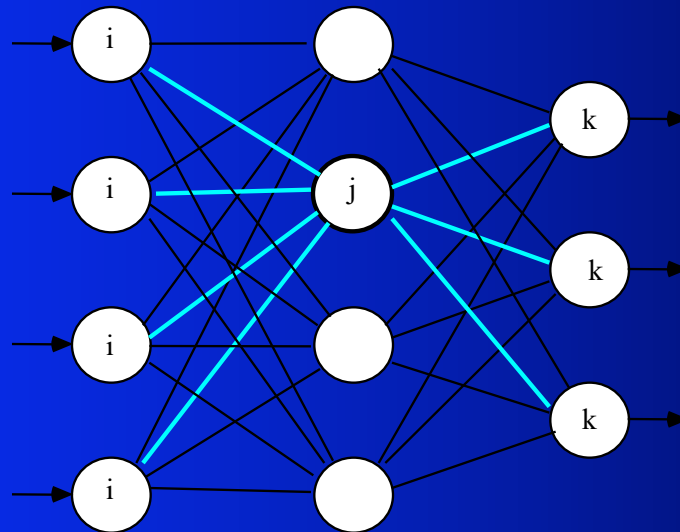


# Backpropagation Learning Equations

$$\Delta w_{ij} = C \delta_j Z_i$$

$$\delta_j = (T_j - Z_j) f'(net_j) \quad [\text{Output Node}]$$

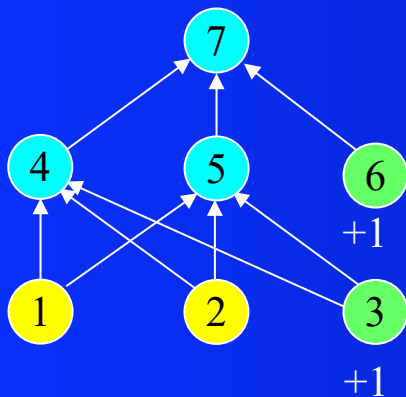
$$\delta_j = \sum_k (\delta_k w_{jk}) f'(net_j) \quad [\text{Hidden Node}]$$





# Backpropagation Learning Example

Assume the following 2-2-1 MLP has all weights initialized to .5. Assume a learning rate of 1. Show the updated weights after training on the pattern .9 .6 -> 0. Show all net values, activations, outputs, and errors. Nodes 1 and 2 (input nodes) and 3 and 6 (bias inputs) are just placeholder nodes and do not pass their values through a sigmoid.



$$Z_j = f(\text{net}_j) = \frac{1}{1 + e^{-\text{net}_j}}$$

$$f'(\text{net}_j) = Z_j(1 - Z_j)$$

$$\Delta w_{ij} = C \delta_j Z_i$$

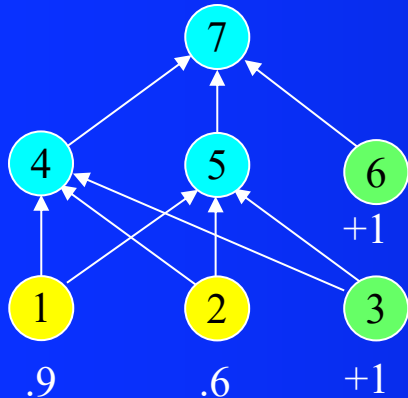
$$\delta_j = (T_j - Z_j) f'(\text{net}_j) \quad [\text{Output Node}]$$

$$\delta_j = \sum_k (\delta_k w_{jk}) f'(\text{net}_j) \quad [\text{Hidden Node}]$$

# Backpropagation Learning Example

$$Z_j = f(\text{net}_j) = \frac{1}{1 + e^{-\text{net}_j}}$$

$$f'(\text{net}_j) = Z_j(1 - Z_j)$$



$$\text{net}_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$$

$$\text{net}_5 = 1.25$$

$$z_4 = 1/(1 + e^{-1.25}) = .777$$

$$z_5 = .777$$

$$\text{net}_7 = .777 * .5 + .777 * .5 + 1 * .5 = 1.277$$

$$z_7 = 1/(1 + e^{-1.277}) = .782$$

$$\delta_7 = (0 - .782) * .782 * (1 - .782) = -.133$$

$$\delta_4 = (-.133 * .5) * .777 * (1 - .777) = -.0115$$

$$\delta_5 = -.0115$$

$$w_{14} = .5 + (1 * -.0115 * .9) = .4896$$

$$w_{15} = .4896$$

$$w_{24} = .5 + (1 * -.0115 * .6) = .4931$$

$$w_{25} = .4931$$

$$w_{34} = .5 + (1 * -.0115 * 1) = .4885$$

$$w_{35} = .4885$$

$$w_{47} = .5 + (1 * -.133 * .777) = .3964$$

$$w_{57} = .3964$$

$$w_{67} = .5 + (1 * -.133 * 1) = .3667$$

$$\Delta w_{ij} = C \delta_j Z_i$$

$$\delta_j = (T_j - Z_j) f'(\text{net}_j) \quad [\text{Output Node}]$$

$$\delta_j = \sum_k (\delta_k w_{jk}) f'(\text{net}_j) \quad [\text{Hidden Node}]$$

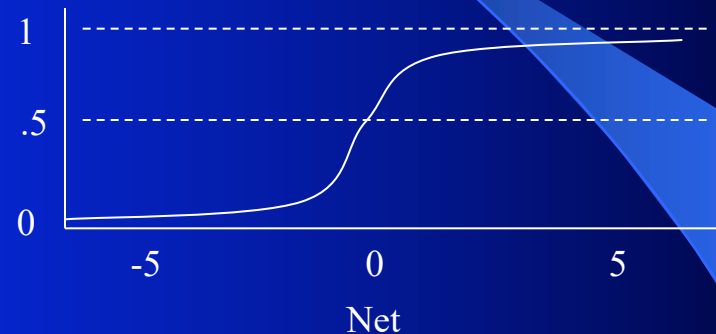
# Backprop Homework

1. For your homework, update the weights for a second pattern  $-1 \ .4 \rightarrow \ .2$ . Continue using the updated weights shown on the previous slide. Show your work like we did on the previous slide.
  2. Then go to the link below: Neural Network Playground using the *tensorflow* tool and play around with the BP simulation. Try different training sets, layers, inputs, etc. and get a feel for what the nodes are doing. You do not have to hand anything in for this part.
- <http://playground.tensorflow.org/>

# Activation Function and its Derivative

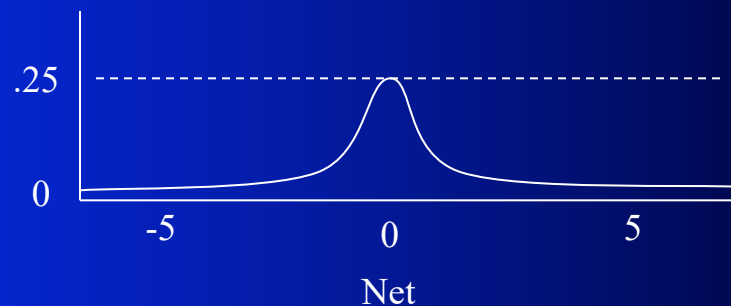
- Node activation function  $f(net)$  is commonly the sigmoid

$$Z_j = f(net_j) = \frac{1}{1 + e^{-net_j}}$$



- Derivative of activation function is a critical part of the algorithm

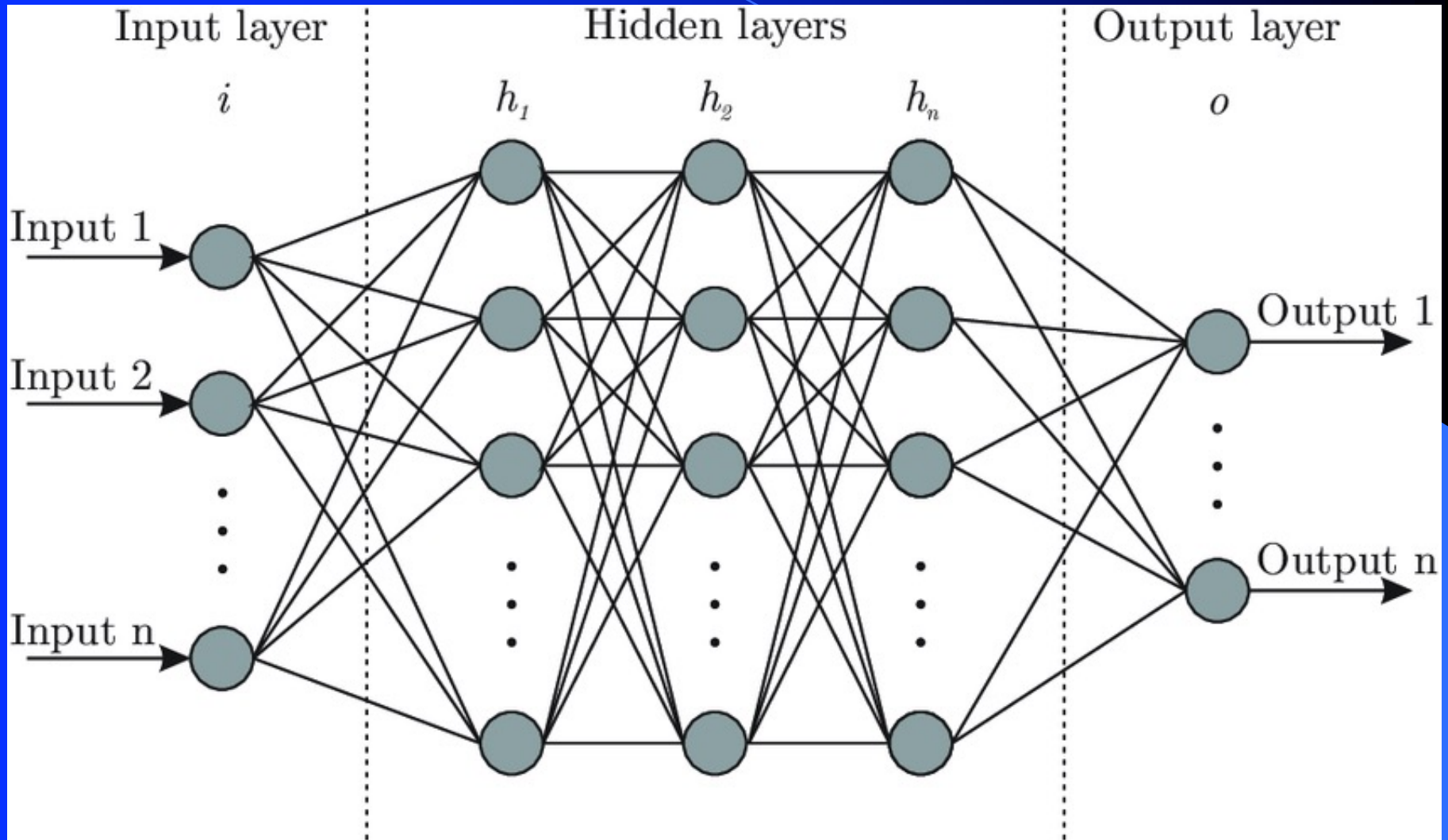
$$f'(net_j) = Z_j(1 - Z_j)$$



# Inductive Bias & Intuition

- Intuition
  - Manager/Worker Interaction
  - Gives some stability
- Node Saturation - Avoid early, but all right later
  - With small weights all nodes have low confidence at first (linear range)
  - When saturated (confident), the output changes only slightly as the net changes. An incorrect output node will still have low error.
  - Start with weights close to 0. Once nodes saturated, hopefully have learned correctly, others still looking for their niche.
  - Saturated error even when wrong? – Multiple TSS drops
  - Don't start with equal weights (can get stuck), random small Gaussian/uniform with 0 mean
- Inductive Bias
  - Start with simple net (small weights, initially linear changes)
  - Gradually build a more complex surface until accurate enough without getting too complex

# Multi-layer Perceptron (MLP) Topology



# Softmax Output Layer Activation

- For *classification* problems it is increasingly popular to use the *softmax* activation function, just at the output layer
- Softmax (softens) 1 of  $n$  targets to mimic a probability vector for the output nodes

$$f(\text{net}_j) = \frac{e^{\text{net}_j}}{\sum_{i=1}^n e^{\text{net}_i}}$$

- If there were 3 output nodes with net values 0, .5, and 1 then the outputs for each node would be  $1/5.37$ ,  $1.65/5.37$ ,  $2.72/5.37 = .18$ ,  $.31$ ,  $.51$  which sums to 1 and can be considered as probability estimates
- All the hidden nodes still do a standard activation function such as logistic, hyperbolic tangent, or ReLU
- Sklearn automatically uses softmax at the output layer for MLP classification, and you choose the activation function for hidden nodes

# Cross-Entropy and Softmax

- For *classification* it is increasingly popular to use the cross-entropy loss function.
- Cross entropy measures the difference (in entropy) between two distributions.
- Cross entropy seeks to find the maximum likelihood hypotheses under the assumption that the observed (1 of  $n$ ) Boolean outputs is a probabilistic function of the input instance which fits what classification does. Maximizing likelihood can be cast as the equivalent minimizing of the negative log likelihood.

$$Loss_{CrossEntropy} = - \sum_{i=1}^n t_i \ln(z_i)$$

$t$	$z$	CE
0	?	0
1	1	0
1	.9	.11
1	.5	.69
1	.1	2.30

- We must recalculate the gradient weight update equation when we use new activation/loss functions. For Softmax with Cross Entropy, Gradient/Error on the output is just  $(t-z)$ , with no  $f'(net)$ . The exponent of softmax is unraveled by the  $\ln$  of cross entropy.
- The hidden layers still update as usual and include  $f'(net)$
- Sklearn always uses this approach for MLP classification



# Regression with MLP/BP

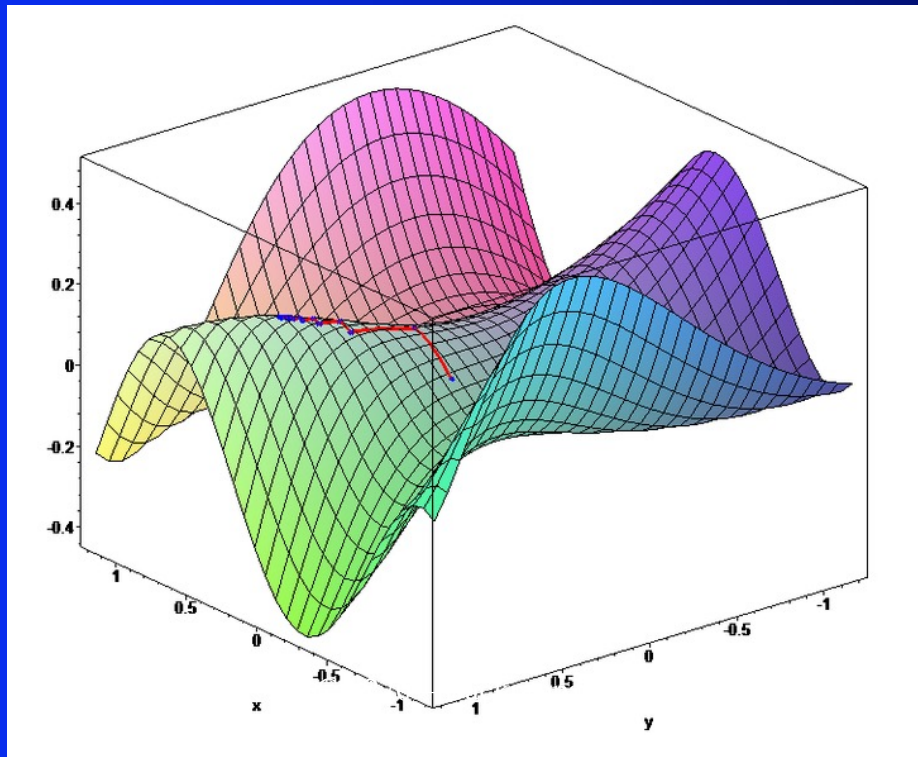
- For regression in MLPs we use the sum-squared error (L2) loss function which seeks the maximum likelihood hypothesis under the assumption that the training data can be modeled by normally distributed noise added to the target function value. More natural for regression than for classification.
- Output nodes use a linear activation (i.e. identity function which just passes the *net* value through). This naturally supports unconstrained regression.
  - Don't typically normalize output
- The output error is still  $(t - z) f'(net)$ , but since  $f'(net)$  is 1 for the linear activation, the output error is just  $(target - output)$
- Hidden nodes still use a non-linear activation function (such as logistic) with the standard  $f'(net)$
- This is how sklearn always does MLP regression

# Local Minima

- Most algorithms which have difficulties with simple tasks get much worse with more complex tasks
- Good news with MLPs
- Many dimensions make for many descent options
- Local minima more common with simple/toy problems, rare with larger problems and larger nets
- Even if there are occasional minima problems, could simply train multiple times and pick the best
- Some algorithms add noise to the updates to escape minima

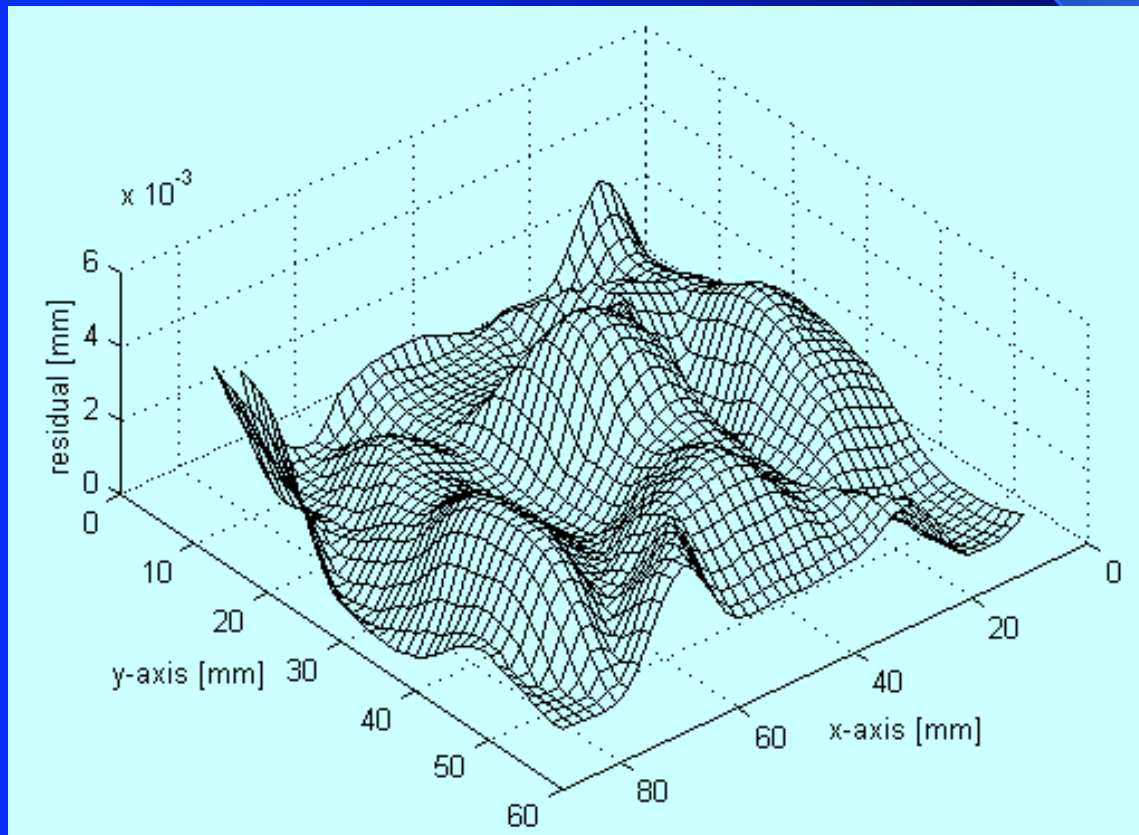
# Local Minima and Neural Networks

- Neural Network can get stuck in local minima for small networks, but for most large networks (many weights), local minima rarely occur in practice
- This is because with so many dimensions of weights it is unlikely that we are in a minima in every dimension simultaneously – almost always a way down

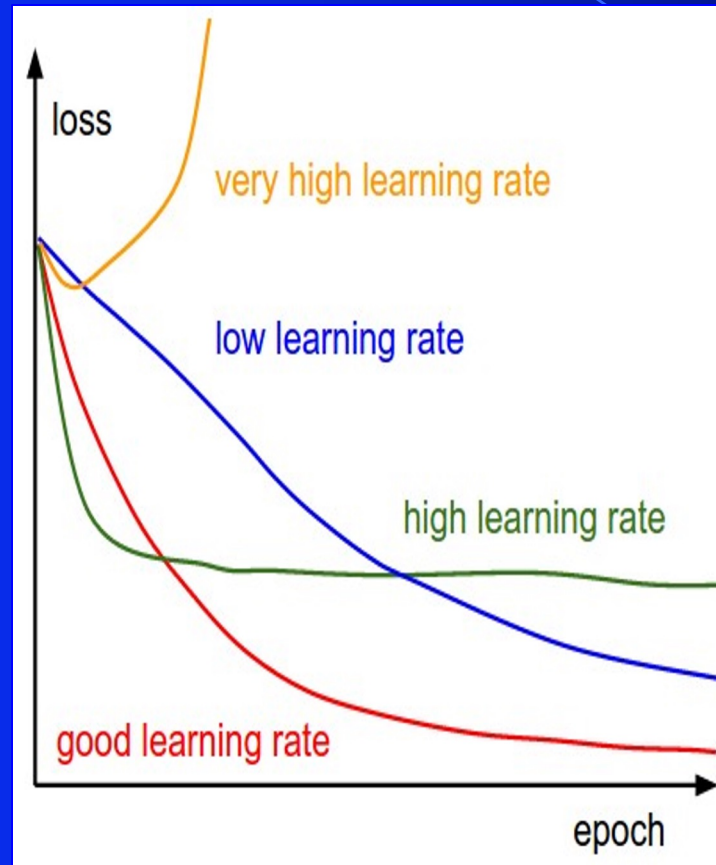


# Learning Rate

- Learning Rate - Relatively small (.01 - .5 common), if too large BP will not converge or be less accurate, if too small it is just slower with no accuracy improvement as it gets even smaller
- Gradient – only where you are, too big of jumps?

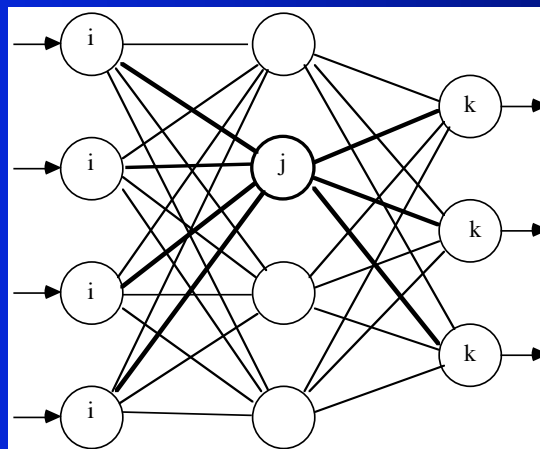


# Learning Rate



# Number of Hidden Nodes

- How many needed is a function of how hard the task is
- Common to use one fully connected hidden layer. Initial number could be  $\sim 2n$  hidden nodes where  $n$  is the number of inputs.
- In practice we train with a small number of hidden nodes, then keep doubling, etc. until no more significant improvement on test sets
  - Too few will underfit
  - Too many nodes can make learning slower and could overfit
    - Having somewhat too many hidden nodes is preferable if using reasonable regularization; avoids underfit and should ignore unneeded nodes
- Each output and hidden node should have its own bias weight



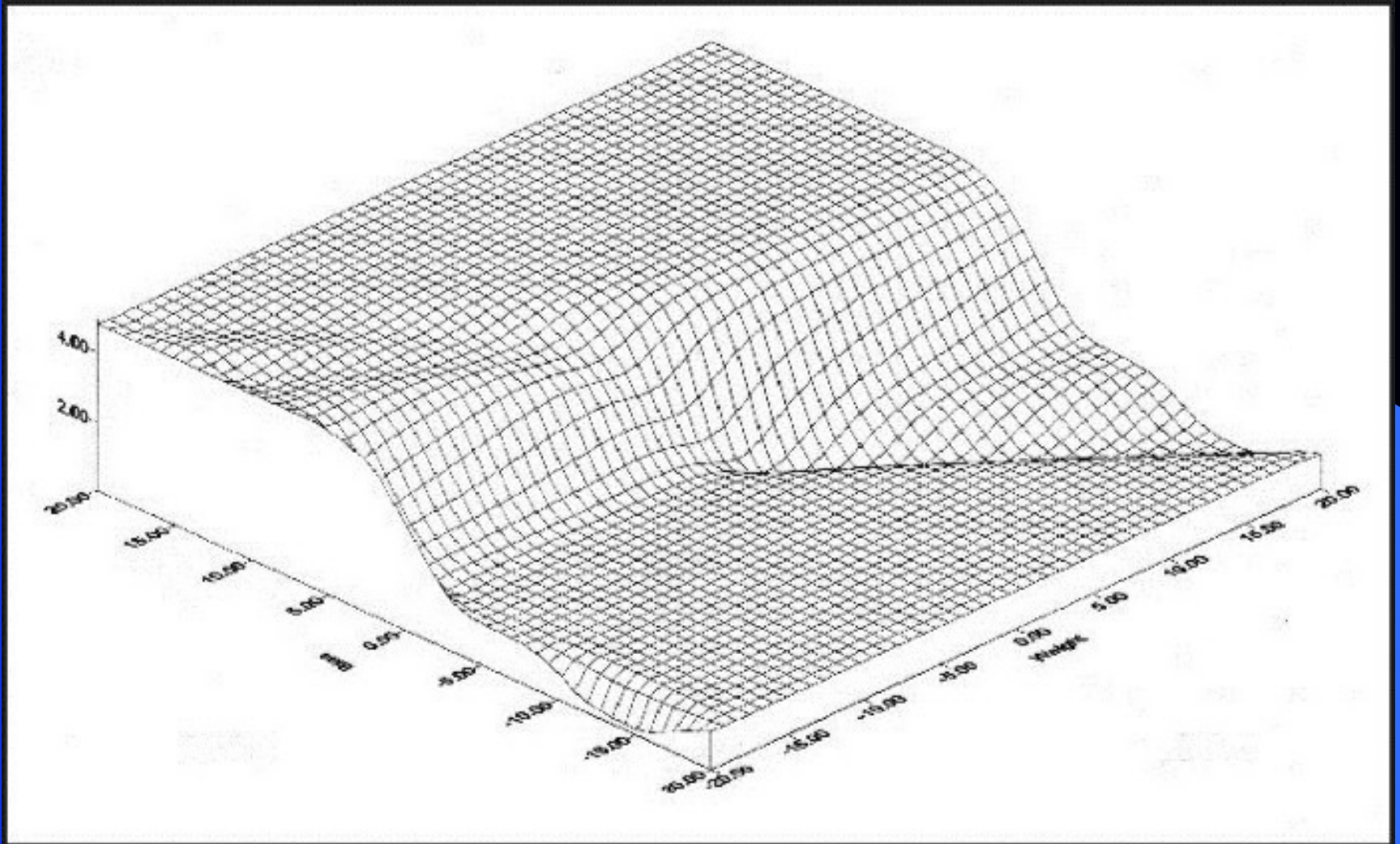
# Momentum

- Simple speed-up modification (type of adaptive learning rate)

$$\Delta w_{ij}(t) = C \delta_j z_i + \alpha \Delta w_{ij}(t-1)$$

- Save  $\Delta w_{ij}(t)$  for each weight to be used as next  $\Delta w_{ij}(t-1)$
- Weight update maintains momentum in the direction it has been going
  - Faster in flatter parts of error surface
  - Significant speed-up, common value  $\alpha \approx .9$
  - Effectively increases learning rate in areas where the gradient is consistently the same sign. (Which is a common approach in adaptive learning rate methods which we will mention later).
- These types of terms make the algorithm less pure in terms of gradient descent. In fact, for SGD (Stochastic Gradient Descent), is like a mini-batch to average gradient
  - Not an issue in terms of local minima (why?)

# Error Surface





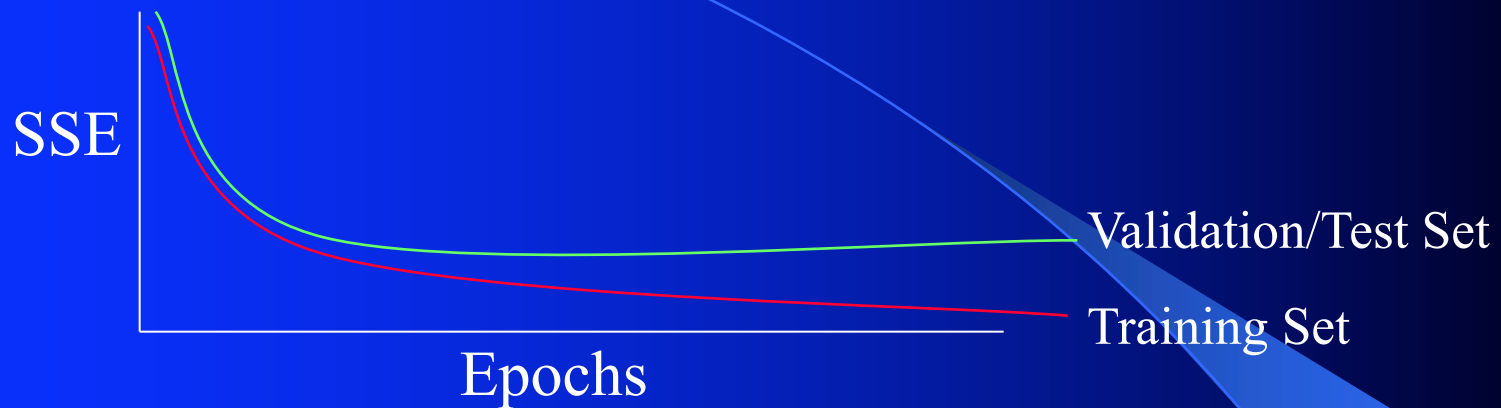
# Hyperparameter Selection

- LR (e.g. .1)
- Momentum – (.5 ... .99)
- Connectivity: fully connected between layers
- Number of hidden nodes: Problem dependent
- Number of layers: 1 (common) or 2 hidden layers which are usually sufficient for good results, attenuation makes learning very slow – modern deep learning approaches show significant improvement using many layers and many hidden nodes
- Manual CV can be used to set hyperparameters: trial and error runs
  - Often sequential: find one hyperparameter value with others held constant, freeze it, find next hyperparameter, etc.
- Hyperparameters could be learned by the learning algorithm in which case you must take care to not overfit the training data – always use a cross-validation technique to measure hyperparameters

# Automated Hyper-Parameter Search

- Can also do an automated search: Grid, Random, others
- User chooses which CV technique to use for each trial
- Grid Search: User chooses a set of possible parameter values and grid search exhaustively tries all possibilities
  - #hidden\_nodes: [6, 12, 24, 48], LR: [.001, .01, .1], ...
- Random Search: User chooses a distribution over chosen hyperparameters and the space is sampled randomly
  - #hidden\_nodes: uniform[5, 50], LR: loguniform[.001, .1], Activation: [logistic, relu, tanh] ... (If no distribution given, then uniformly samples)
  - User chooses how many iterations to try (better time control)
- Some advantages of Random search
  - Grid becomes very slow for lots of parameters – too many runs
  - Grid can only choose from specified parameter values, no tweeners
- You will try Grid Search and Random Search in your lab

# Stopping Criteria and Overfit Avoidance



- More Training Data (vs. overtraining - One epoch in the limit)
- Validation Set - save weights which do best job so far on the validation set. Keep training for enough epochs to be sure that no more improvement will occur (e.g. once you have trained  $m$  epochs with no further improvement (bssf), stop and use the best weights so far, or retrain with all data).
  - Note: If using  $N$ -way CV with a validation set, do  $n$  runs with 1 of  $n$  data partitions as a validation set. Save the number of training updates for each run. To get a final model you can train on all the data and stop after the average number of training updates.
- Specific BP techniques for avoiding overfit
  - Less hidden nodes NOT a great approach because may underfit
  - Weight decay (regularization), Adjusted Error functions (deltas  $.9/.1$ , CB), Dropout

# Validation Set

- Often you will use a validation set (separate from the training or test set) for stopping criteria, etc.
- In these cases you should take the validation set out of the training set
- For example, you might use the random test set method to randomly break the original data set into 80% training set and 20% test set. Independent and subsequent to the above split you would take  $n\%$  (10-20%) of the training set to be a validation set for that particular training run.
- You will usually shuffle the weight training part of your training set for each epoch, but you use the same unchanged validation set throughout the entire training
  - Never use an instance in the VS which has been used to train weights
  - Sklearn does all this for you by just setting the `early_stopping = True`

# Backpropagation Regularization

- How to avoid overfit – Keep the model simple
  - Keep decision surfaces smooth
  - Smaller overall weight values lead to simpler models with less overfit
- Early stopping with validation set is a common approach to avoid overfitting (since weights don't have time to get too big)
- Could make complexity an explicit part of the loss function
  - Then we don't need early stopping (though sometimes one is better than the other and we can even do both simultaneously)
- Regularization approach: Model ( $h$ ) selection
  - Minimize  $F(h) = Error(h) + \lambda \cdot Complexity(h)$
  - Tradeoff accuracy vs complexity
- Two common approaches
  - Lasso (L1 regularization)
  - Ridge (L2 regularization)

# L1 (Lasso) Regularization

- Standard BP update is based on the derivative of the loss function with respect to the weights. We can add a model complexity term directly to the loss function such as:
  - $L(\mathbf{w}) = \text{Error}(\mathbf{w}) + \lambda \sum |w_i|$
  - $\lambda$  is a hyperparameter which controls how much we value model simplicity vs training set accuracy
  - Gradient of  $L(\mathbf{w})$ : Gradient of  $\text{Error}(\mathbf{w}) + \lambda$
  - To make it gradient descent we negate the Gradient:  $(-\nabla \text{Error}(\mathbf{w}) - \lambda)$ 
    - This is also called weight decay
    - Gradient of Error is just equations we have used if  $\text{Error}(\mathbf{w})$  is TSS, but may differ for other error functions
- Common values for lambda are 0, .001, .01, .03, etc.
- Weights that really should be significant stay large enough, but weights just being nudged by a few data instances go to 0

# L2 (Ridge) Regularization

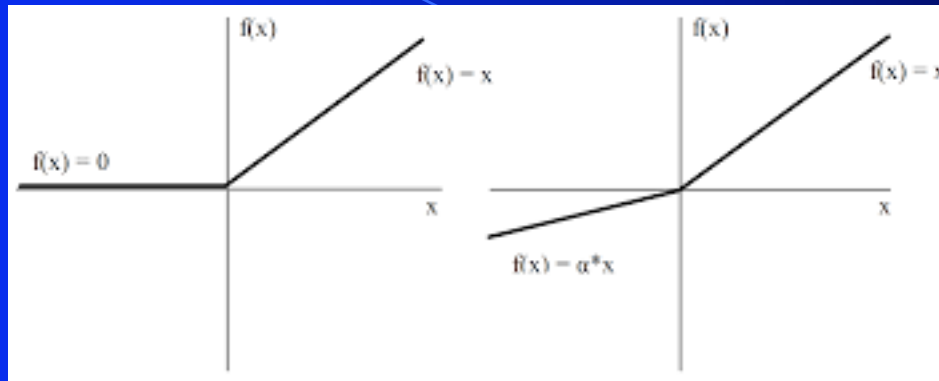
- $L(\mathbf{w}) = \text{Error}(\mathbf{w}) + \lambda \sum w_i^2$
- -Gradient of  $L(\mathbf{w})$ : -Gradient of Error( $\mathbf{w}$ ) -  $2\lambda w_i$
- Regularization portion of weight update is scaled by weight value (fold 2 into  $\lambda$ )
  - Decreases change when weight small ( $<0$ ), otherwise increases
  - $\lambda$  is % of weight change, .03 means 3% of the weight is decayed each time
- L1 vs L2 Regularization
  - L1 drives many weights all the way to 0 (Sparse representation and feature reduction)
  - L1 more robust to large weights (outliers), while L2 makes larger decay with large weights
  - L1 leads to simpler models, but L2 often more accurate with more complex problems which require a bit more complexity

# BP Lab

- Go over Lab together



# Rectified Linear Units



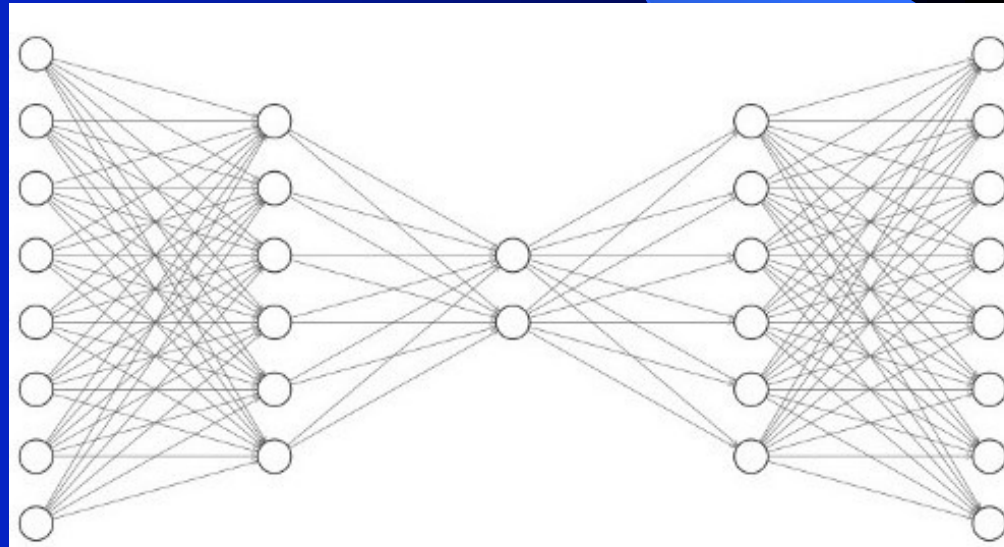
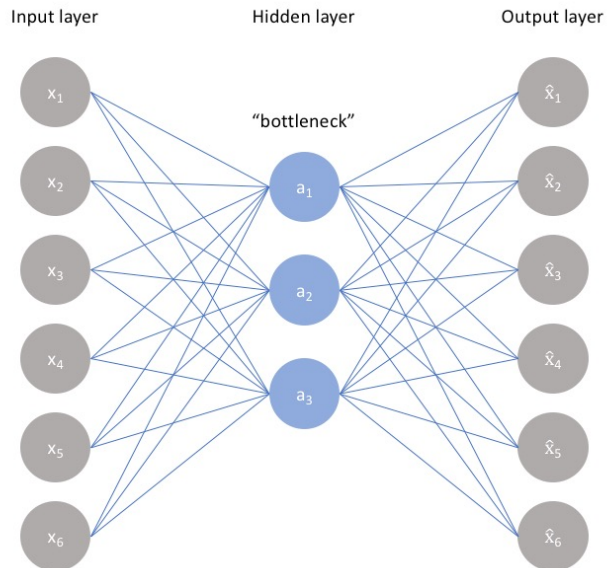
- BP can work with any differentiable non-linear activation function (e.g. sine)
- *ReLU* is common these days especially with deep learning:  $f(x) = \text{Max}(0, x)$ 
  - More efficient computation: Only comparison, addition and multiplication
  - $f'(net)$  is 0 or constant, just fold into learning rate
- Leaky ReLU  $f(x) = x$  if  $x > 0$ , else  $ax$ , where  $0 \leq a \leq 1$ , so for net  $< 0$  the derivate is not 0 and can do some learning (does not “die”).
  - Lots of other variations
- Sparse activation: For example, in a randomly initialized networks, only about 50% of hidden units are activated (having a non-zero output)
- Not differentiable but we just “cheat” and include the discontinuity point with either side of the linear part of the ReLU function – piecewise linear

# Debugging ML algorithms

- Debugging ML algorithms can be difficult
  - Unsure beforehand about what the results should be, differ for different tasks, data splits, initial random weights, hyperparameters, etc.
  - Adaptive algorithm can learn to compensate somewhat for bugs
  - Bugs in accuracy evaluation code common – false hopes!
- **\*\*Do a small example by hand (e.g. your homework) and make sure your algorithm gets the exact same results (and accuracy)**
- Compare results with our supplied debug and LS examples
- Compare results (not code, etc.) with classmates
- Compare results with a published version of the algorithm (e.g. sklearn), won't be exact because of different training/test splits, etc.
  - Use Zarndt's thesis (or other publications) to get a ballpark feel of how well you should expect to do on different data sets.  
<http://axon.cs.byu.edu/papers/Zarndt.thesis95.pdf>

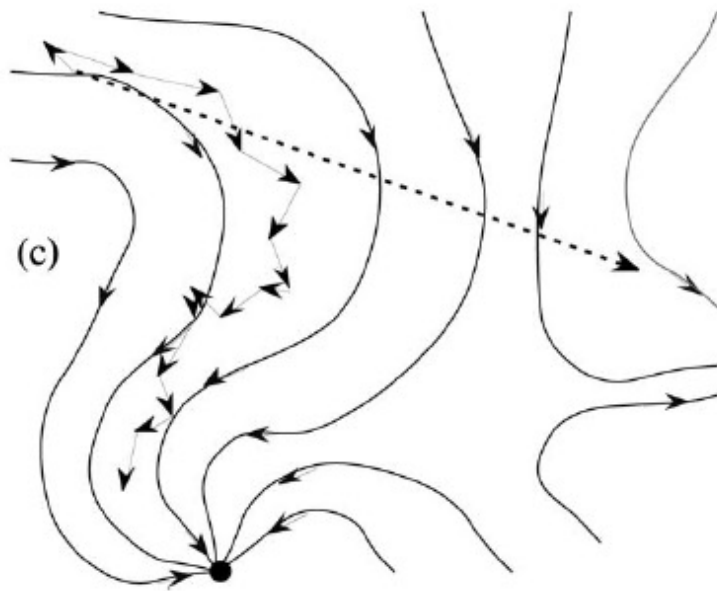
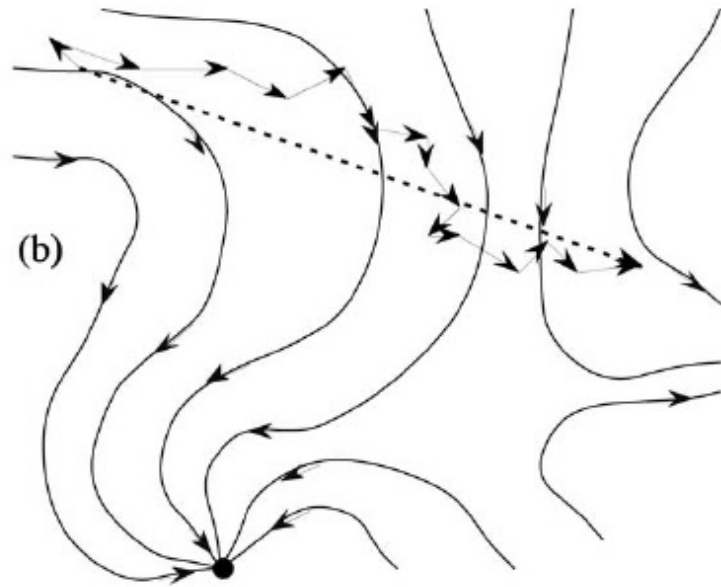
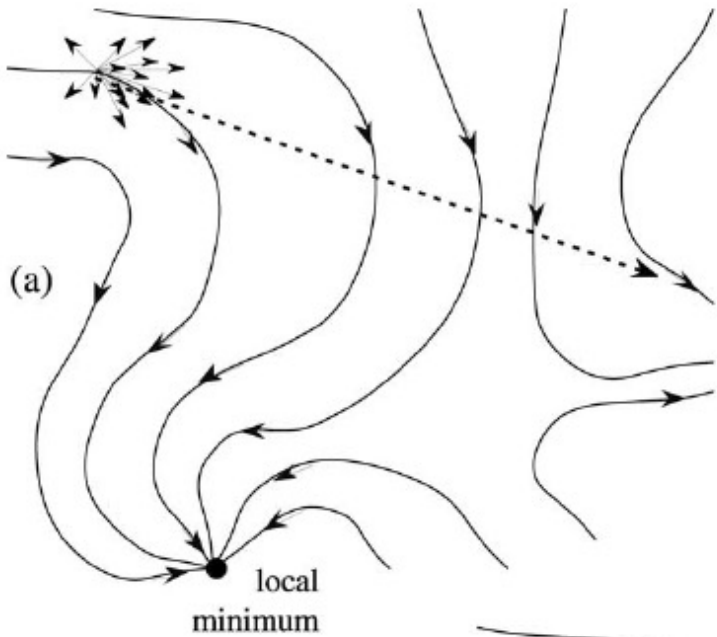
# What are the Hidden Nodes Doing?

- Higher order features vs 1<sup>st</sup> order features (perceptron/Us)
  - The real power of machine learning (exponential # of variations)
- Hidden nodes discover new *higher order* features which are fed into subsequent layers
- Zipser - Linguistics
- Compression



# Batch Update

- With On-line (stochastic) update we update weights after every pattern
- With Batch update we accumulate the changes for each weight, but do not update them until the end of each epoch
- Batch update gives a correct direction of the gradient for the entire data set, while on-line could do some weight updates in directions quite different from the average gradient of the entire data set
  - Based on noisy instances and also just that specific instances will not usually be at the average gradient
- Proper approach? - Conference experience/Parallel experience
  - Most (including us) assumed batch more appropriate, but batch/on-line a non-critical decision with similar results
- We show that batch is less efficient
  - Wilson, D. R. and Martinez, T. R., The General Inefficiency of Batch Training for Gradient Descent Learning, *Neural Networks*, vol. 16, no. 10, pp. 1429-1452, 2003



Point of evaluation

Direction of gradient

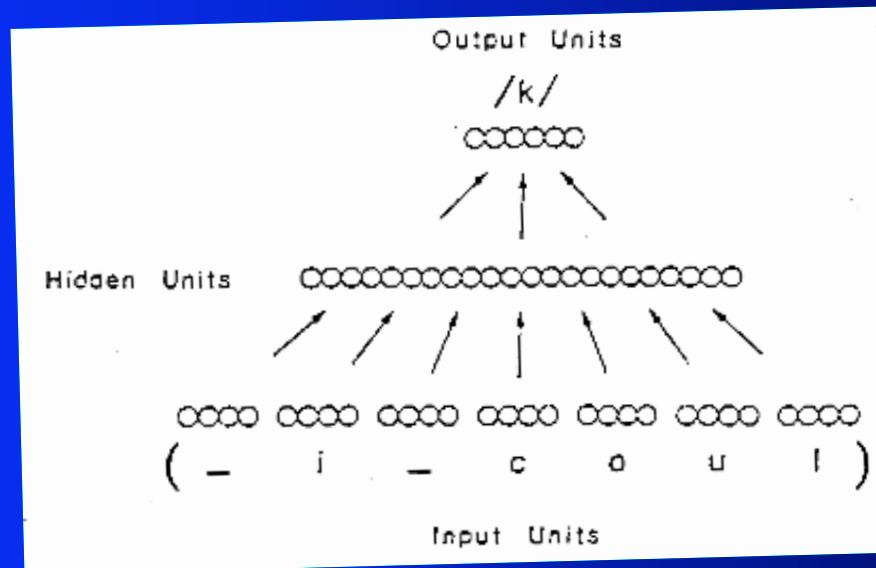
True underlying gradient

# Localist vs. Distributed Representations

- Is Memory Localist (“grandmother cell”) or distributed
- Output Nodes
  - One node for each class (classification) – “one-hot”
  - One or more graded nodes (classification or regression)
  - Distributed representation
- Input Nodes
  - Normalize real and ordered inputs
  - Nominal Inputs - Same options as above for output nodes
- Hidden nodes - Can potentially extract rules if localist representations are discovered. Difficult to pinpoint and interpret distributed representations.

# Application Example - NetTalk

- One of first application attempts
- Train a neural network to read English aloud
- Input Layer - Localist representation of letters and punctuation
- Output layer - Distributed representation of phonemes
- 120 hidden units: 98% correct pronunciation
  - Note steady progression from simple to more complex sounds



# Adaptive Learning Rate Approaches

- Momentum is a type of adaptive learning rate mechanism

$$\Delta w_{ij}(t) = C \delta_j z_i + \alpha \Delta w_{ij}(t-1)$$

- Adaptive Learning rate methods

- Start LR small
- As long as weight change is in the same direction, increase a bit (e.g. scalar multiply  $> 1$ , etc.)
- If weight change changes directions (i.e. sign change) reset LR to small, could also backtrack for that step, or ...



# Speed up variations of SGD

- Use mini-batch rather than single instance for better gradient estimate
  - *Sometimes* helpful if SGD variation more sensitive to bad gradient, and also for some parallel (GPU) implementations.
- Adaptive learning rate approaches (and other speed-ups) are often used for deep learning since there are so many training updates
  - Standard Momentum
    - Note that these approaches already do an averaging of gradient, also making mini-batch less critical
  - Nesterov Momentum – Calculate point you would go to if using normal momentum. Then, compute gradient at that point. Do normal update using *that* gradient and momentum.
  - Rprop – Resilient BP, if gradient sign inverts, decrease it's individual LR, else increase it – common goal is faster in the flats, variants that backtrack a step, etc.
  - Adagrad – Scale LRs inversely proportional to  $\sqrt{\text{sum}(\text{historical values})}$
  - RMSprop – Adagrad but uses exponentially weighted moving average, older updates basically forgotten
  - Adam (Adaptive moments) – Momentum terms on both gradient and squared gradient (uncentered variance) (1<sup>st</sup> and 2<sup>nd</sup> moments) – updates based on a moving average of both - Popular

# Learning Variations

- Different activation functions - need only be differentiable
- Different objective functions
  - Cross-Entropy
  - Classification Based Learning
- Higher Order Algorithms - 2nd derivatives (Hessian Matrix)
  - Quickprop
  - Conjugate Gradient
  - Newton Methods
- Constructive Networks
  - Cascade Correlation
  - DMP (Dynamic Multi-layer Perceptrons)

# Higher order "shortcut"

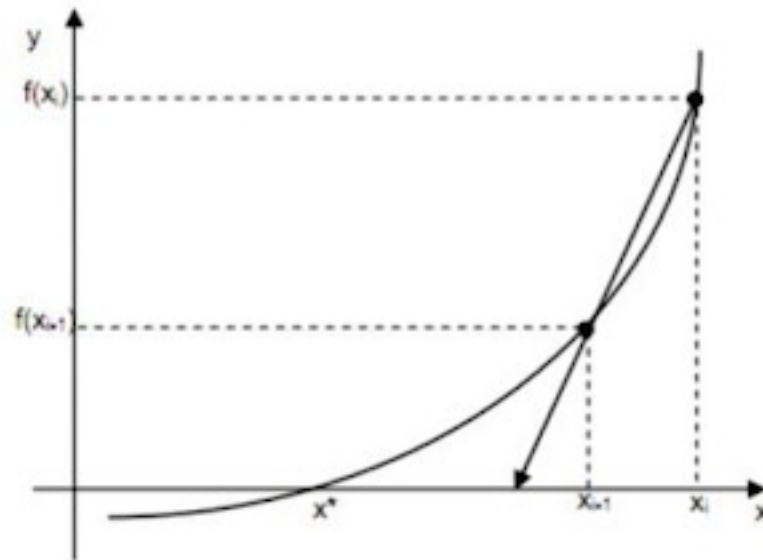
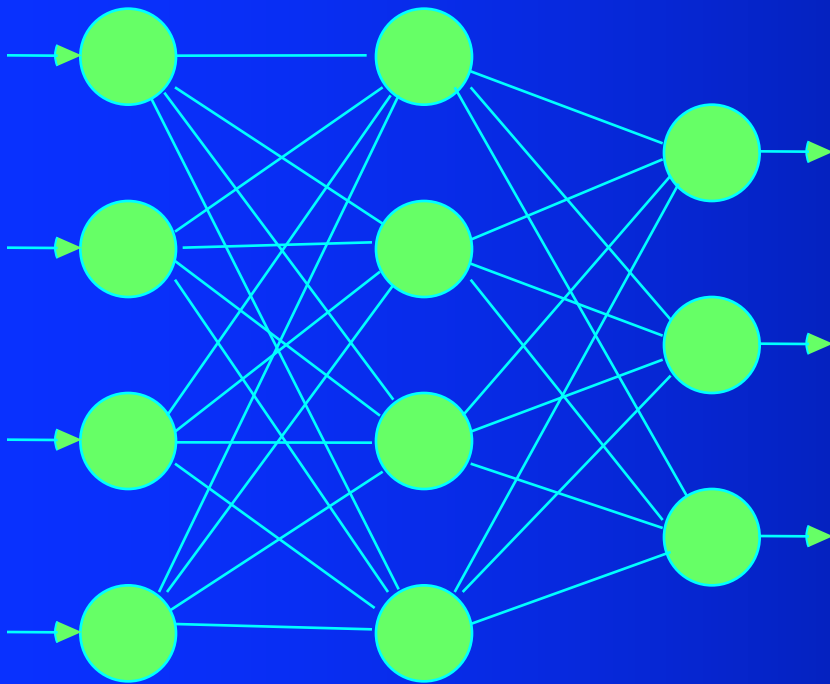


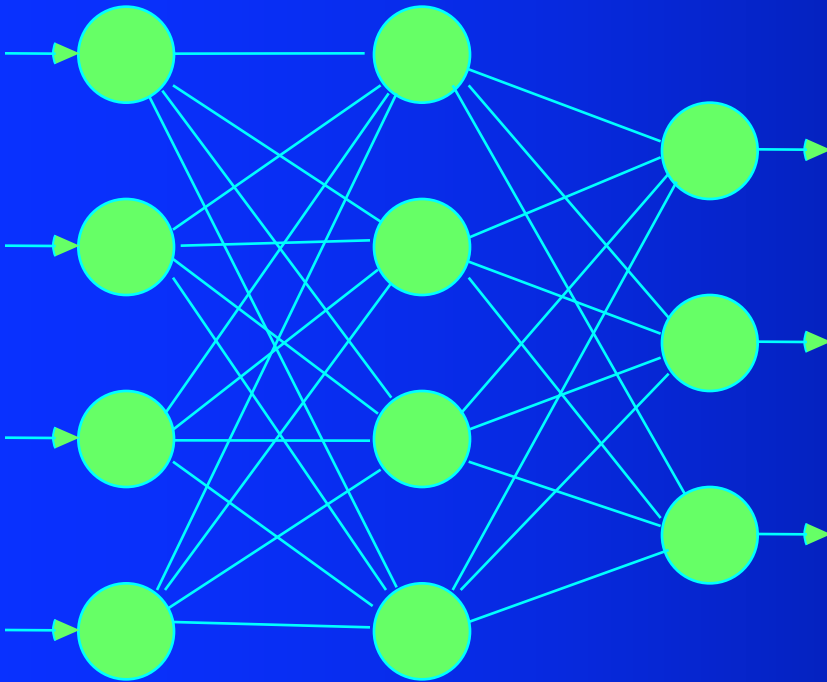
Fig. 6 from 'Metodos numéricos' book

# Classification Based (CB) Learning



Target	Actual	BP Error	CB Error
1	.6	$.4 * f'(net)$	0
0	.4	$-.4 * f'(net)$	0
0	.3	$-.3 * f'(net)$	0

# Classification Based Errors


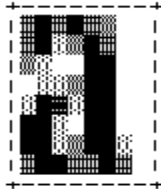


Target	Actual	BP Error	CB Error
1	.6	$.4 * f'(net)$	.1
0	.7	$-.7 * f'(net)$	-.1
0	.3	$-.3 * f'(net)$	0

# Results

- Standard BP: **97.8%**


Sample Output:

	0: a 1.00
	1: A 0.56
	2: Ww 0.05
	3: OoO 0.01
	4: 8 0.01
	5: b 0.00
	6: D 0.00
	7: B 0.00
	8: 3 0.00
	9: Vv 0.00
	10: T 0.00
	11: Cc 0.00
	12: Xx 0.00
	13: Yy 0.00
	14: E 0.00
	15: F 0.00
	16: 4 0.00
	17: 5 0.00
	18: 7 0.00
	19: G6 0.00
	20: Jj 0.00
	21: Q 0.00
	22: Ss 0.00
	23: Zz 0.00
	24: 2 0.00
	25: d 0.00
	26: e 0.00
	27: f 0.00
	28: g9 0.00
	29: q 0.00
	30: t 0.00
	31: N 0.00
	32: R 0.00
	33: H 0.00
	34: Mm 0.00
	35: H 0.00
	36: Iil 0.00
	37: Zz 0.00
	38: Pp 0.00
	39: 1 0.00


# Results

- Classification Based Training:  
**99.1%**

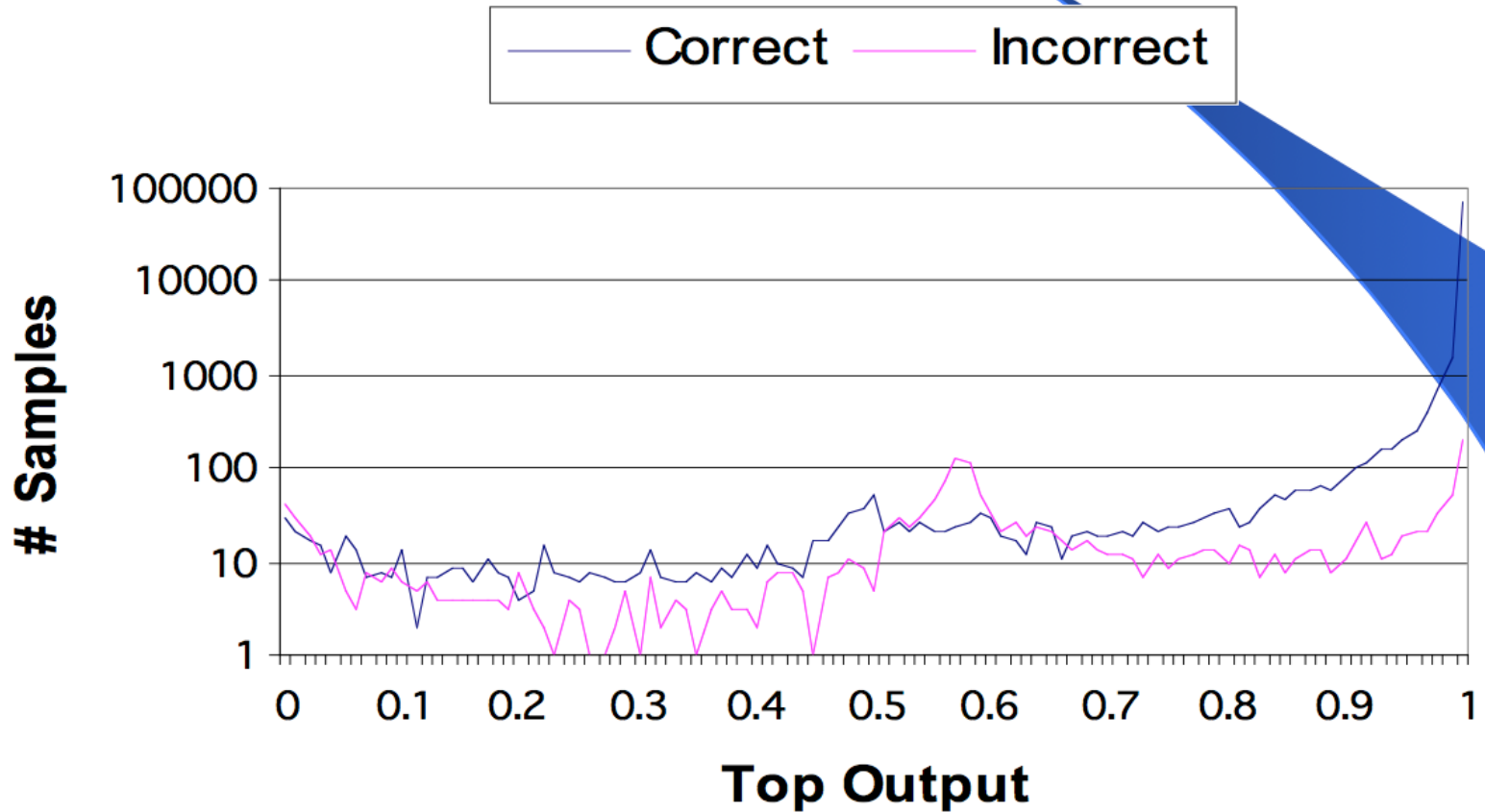
Sample Output:



0:	A	0.71
1:	a	0.53
2:	N	0.44
3:	R	0.44
4:	Ss	0.42
5:	H	0.39
6:	Kk	0.38
7:	Mm	0.36
8:	B	0.35
9:	Ww	0.34
10:	6	0.33
11:	3	0.32
12:	8	0.31
13:	n	0.31
14:	h	0.30
15:	Xx	0.29
16:	Iil	0.28
17:	5	0.28
18:	9	0.28
19:	t	0.27
20:	g	0.24
21:	G	0.23
22:	J	0.22
23:	E	0.22
24:	Uu	0.21
25:	Zz	0.20
26:	4	0.19
27:	d	0.18
28:	Oo0	0.18
29:	L	0.17
30:	2	0.16
31:	b	0.14
32:	f	0.14
33:	e	0.11
34:	Q	0.10
35:	Cc	0.09
36:	Yy	0.09
37:	F	0.08
38:	D	0.07
39:	7	0.06
40:	r	0.06
41:	Pp	0.05
42:	j	0.05
43:	q	0.05
44:	T	0.04
45:	Vv	0.02
46:	1	0.01



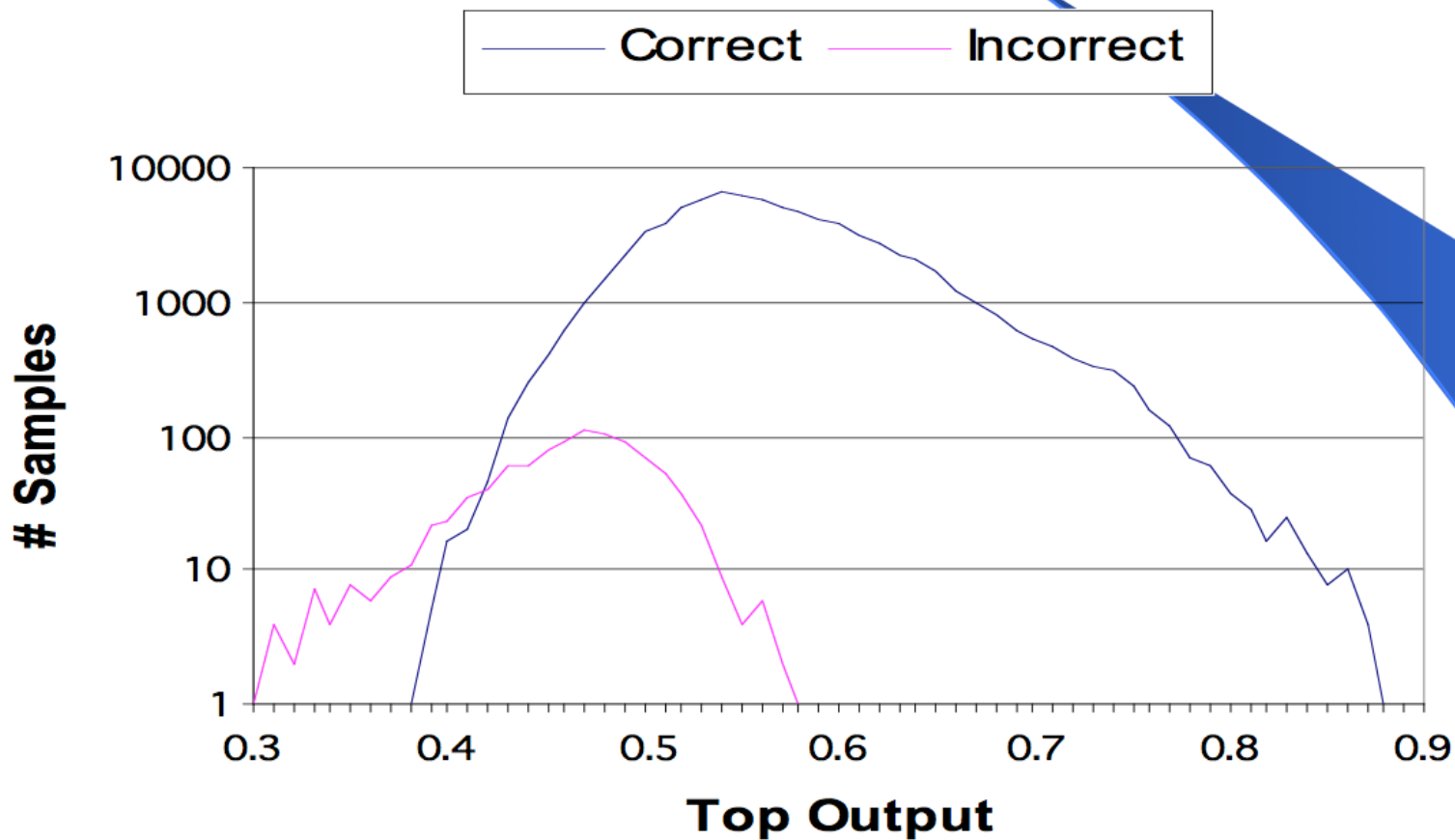
# Analysis



Network outputs on test set after standard backpropagation training.



# Analysis



Network outputs on test set after CB training.

# Classification Based Models

- CB1: Only backpropagates error on misclassified training patterns
- CB2: Adds a confidence margin,  $\mu$ , that is increased globally as training progresses
- CB3: Learns a confidence  $C_i$  for each training pattern  $i$  as training progresses
  - Patterns often misclassified have low confidence
  - Patterns consistently classified correctly gain confidence
  - Best overall results and robustness

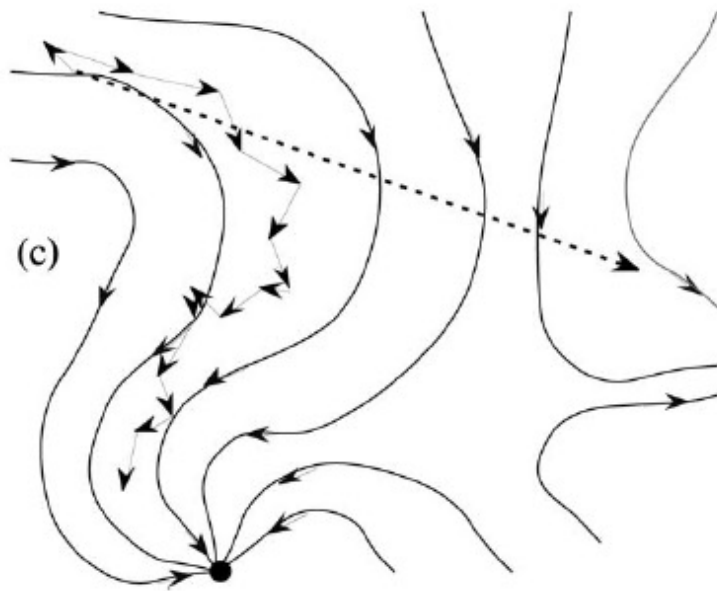
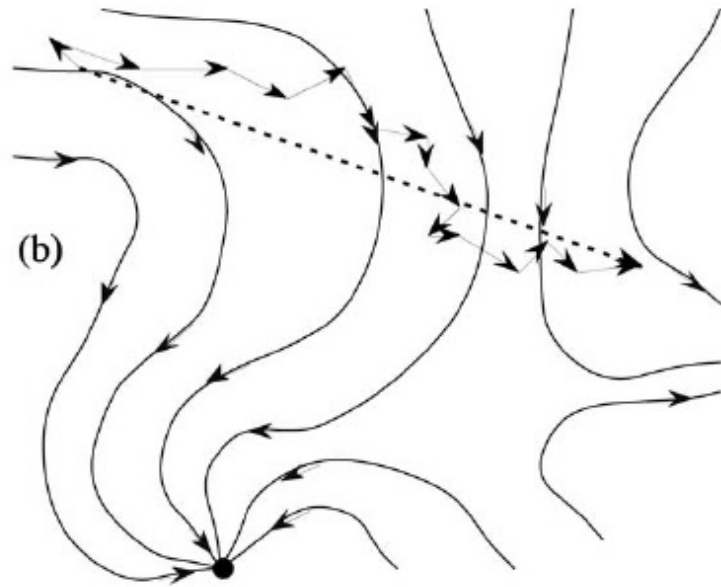
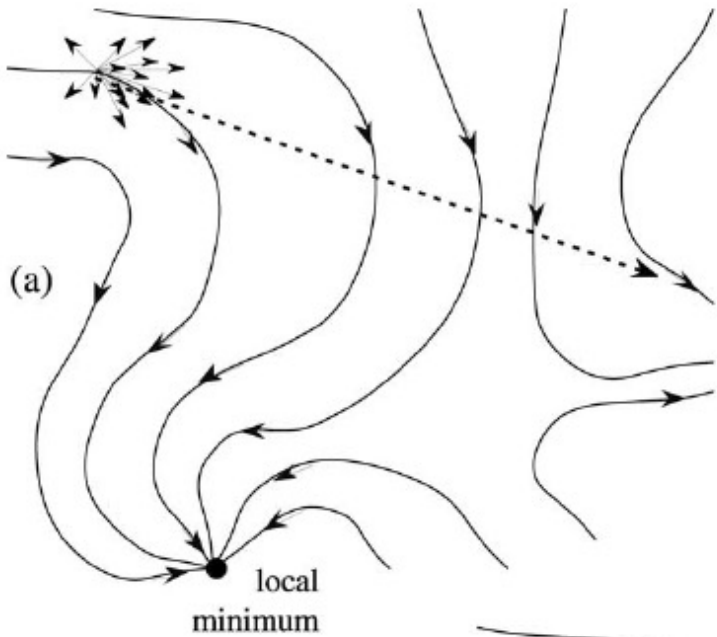
# Batch Update

- With On-line (stochastic) update we update weights after every pattern
- With Batch update we accumulate the changes for each weight, but do not update them until the end of each epoch
- Batch update gives a correct direction of the gradient for the entire data set, while on-line could do some weight updates in directions quite different from the average gradient of the entire data set
  - Based on noisy instances and also just that specific instances will not represent the average gradient
- Proper approach? - Conference experience
  - Most (including us) assumed batch more appropriate, but batch/on-line a non-critical decision with similar results
- We tried to speed up learning through "batch parallelism"

# On-Line vs. Batch

Wilson, D. R. and Martinez, T. R., The General Inefficiency of Batch Training for Gradient Descent Learning, *Neural Networks*, vol. 16, no. 10, pp. 1429-1452, 2003

- Many people still not aware of this issue – Changing
- Misconception regarding “Fairness” in testing batch vs. on-line with the same learning rate
  - BP already sensitive to LR - why? Both approaches need to make a *small* step in the calculated gradient direction – (about the same magnitude)
  - With batch need a "smaller" LR since weight changes accumulate (alternatively divide by  $|TS|$ )
  - To be "fair", on-line should have a comparable LR??
  - Initially tested on relatively small data sets
- On-line update approximately follows the curve of the gradient as the epoch progresses
- With appropriate learning rate batch gives correct result, just less efficient, since you have to compute the entire training set for each small weight update, while on-line will have done  $|TS|$  updates



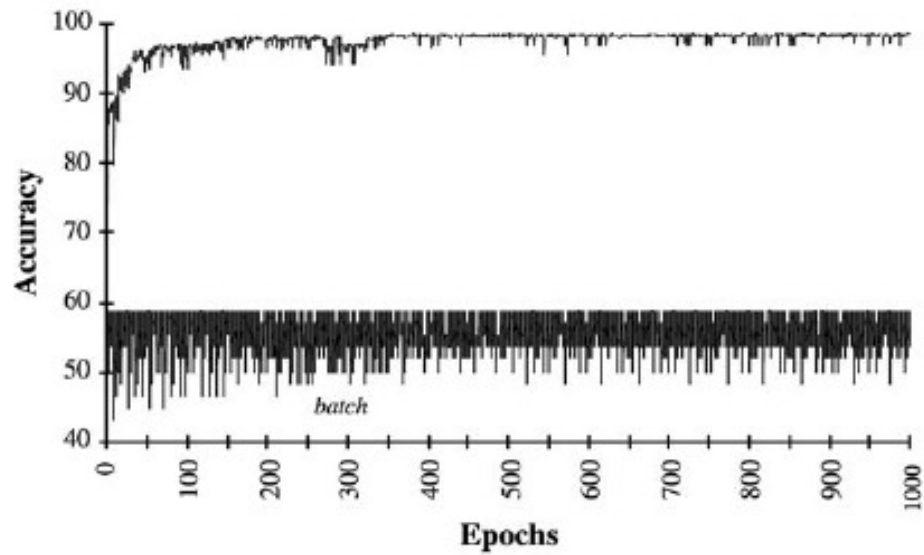
Point of evaluation

Direction of gradient

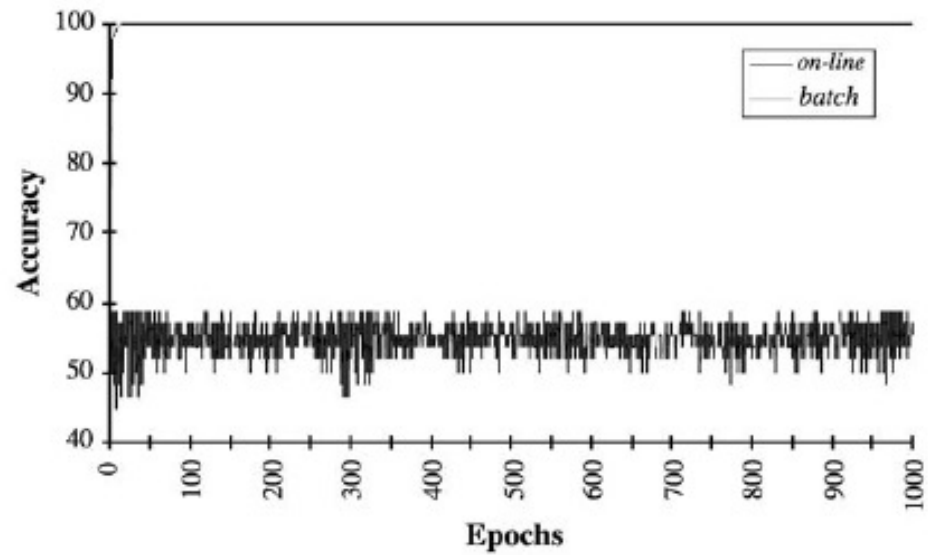
True underlying gradient

### Mushroom

(a)  $r = 0.1$

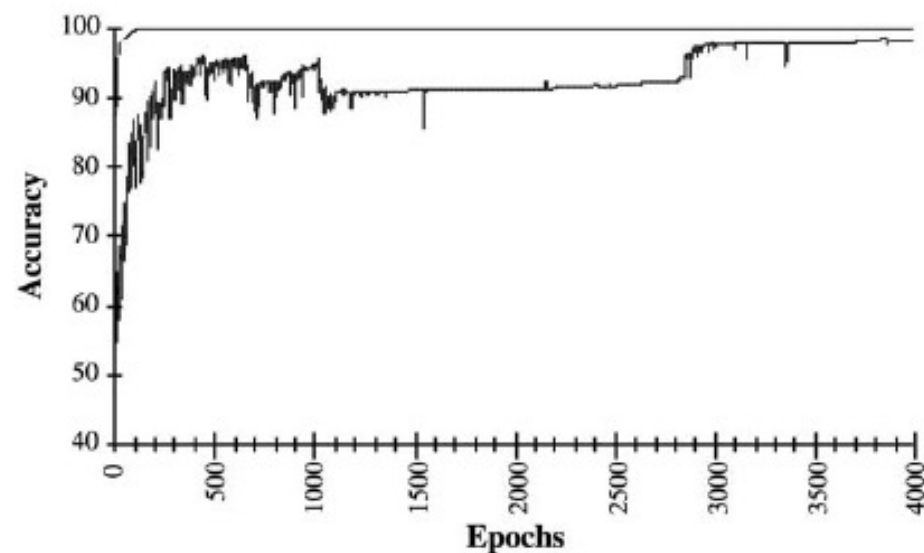


(b)  $r = 0.01$

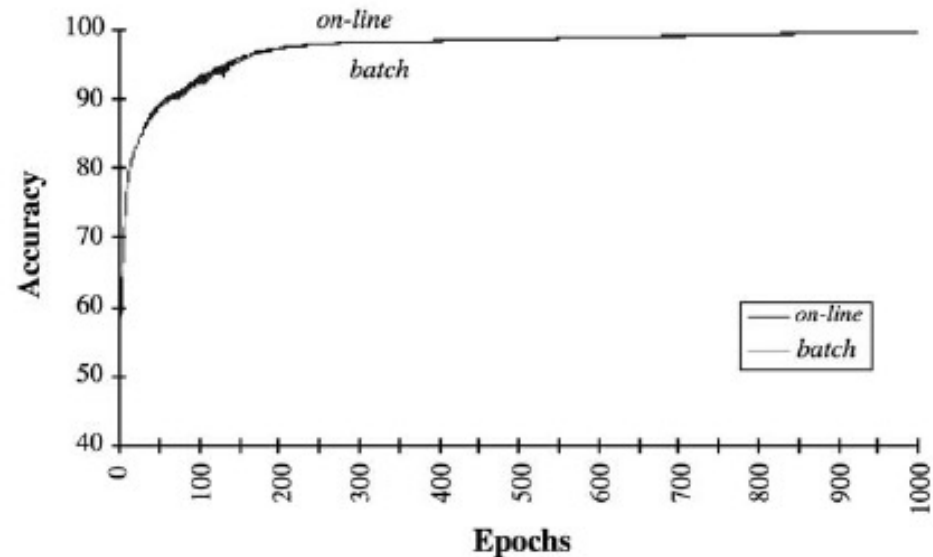


(c)  $r = 0.001$

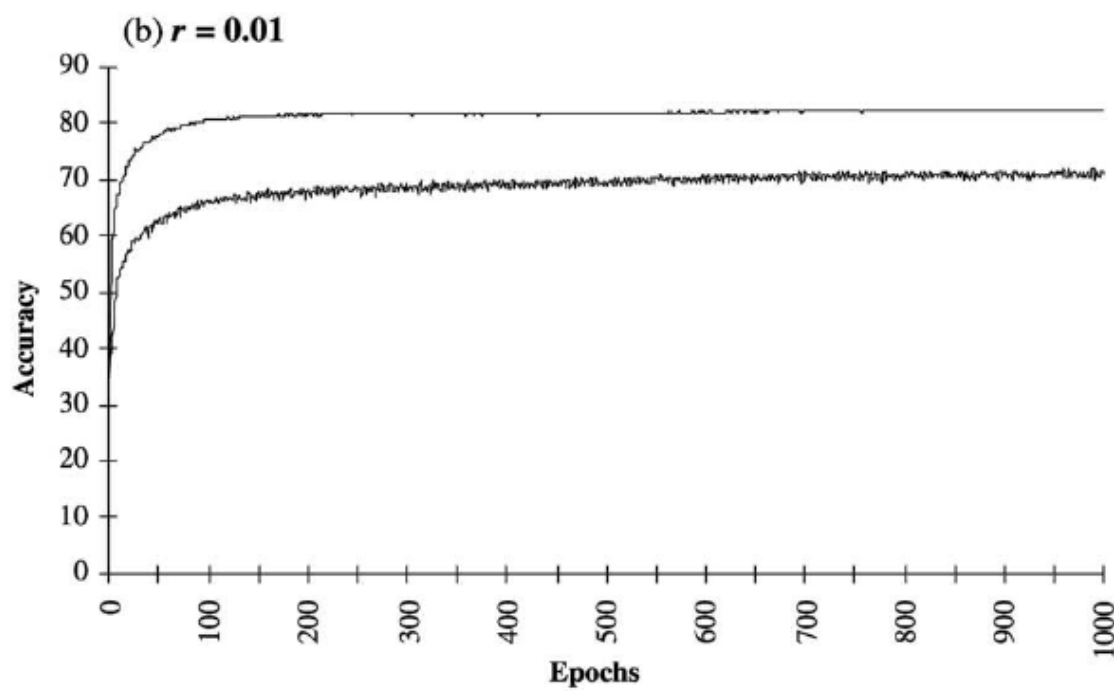
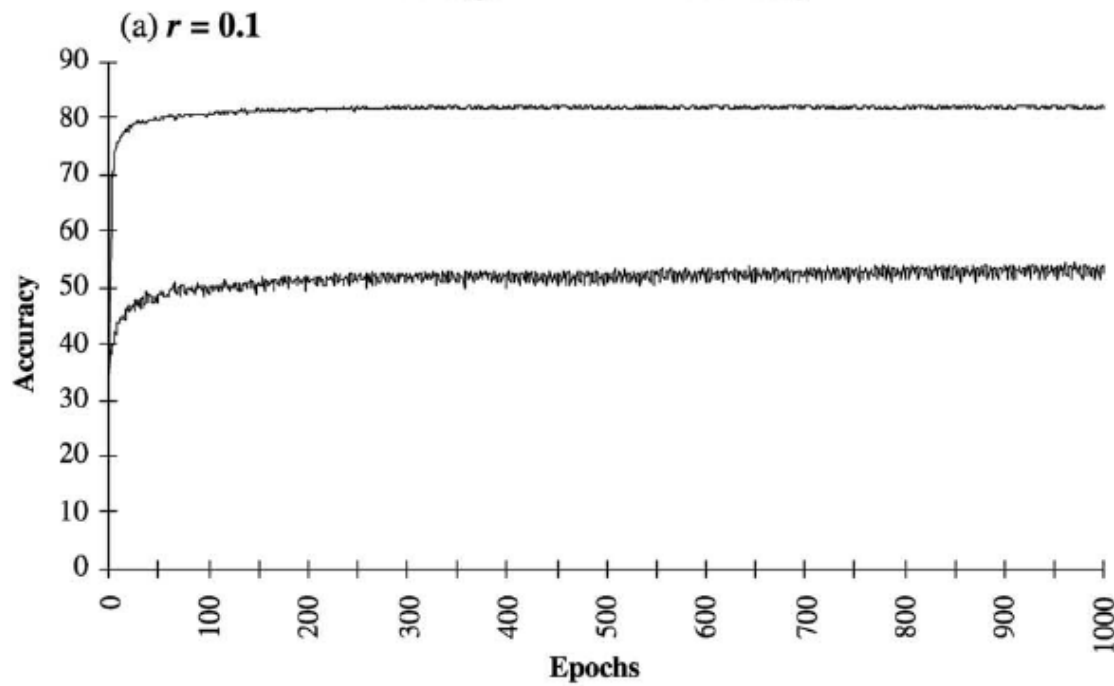
### Mushroom



(d)  $r = 0.0001$



# Average MLDB Accuracy



## Average

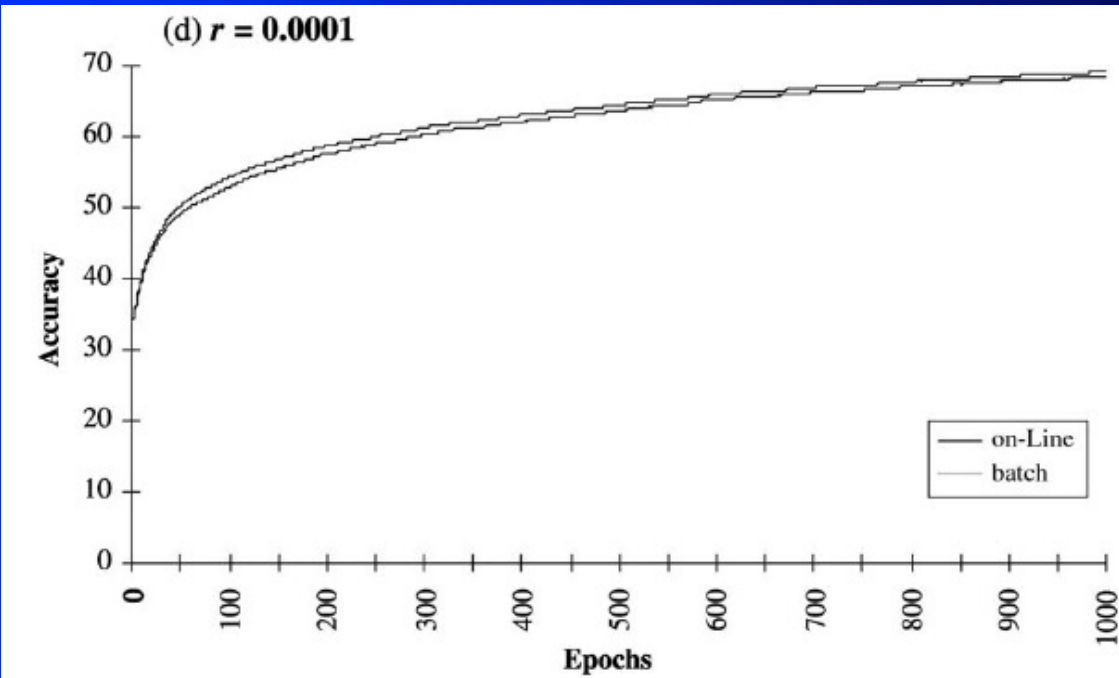
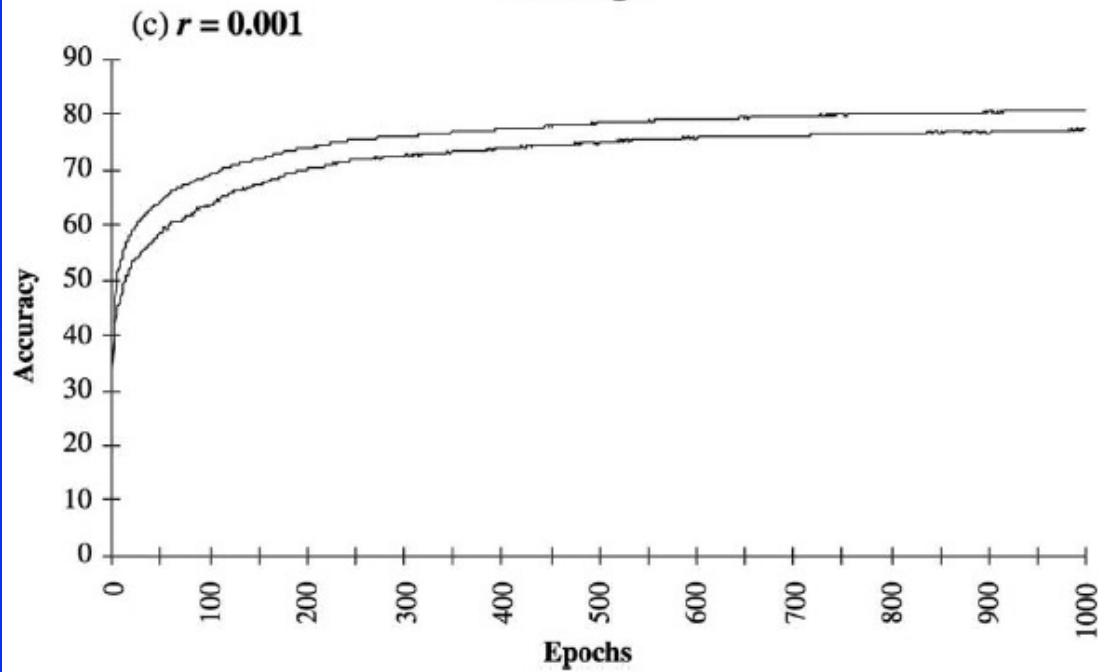




Table 1

Summary of MLDB experiments. The maximum average generalization accuracy for on-line and batch training for each task are shown, along with the 'best' learning rate and the number of epochs required for convergence with the learning rate shown. The ratio of required epochs for batch to on-line training is shown as the 'speedup' of on-line training

Dataset	Output classes	Training set size	Max accuracy			Best LR		Epochs needed		On-line's speedup
			On-line	Batch	Diff.	On-line	Batch	On-line	Batch	
Bridges	7	42	82.14	82.86	-0.72	0.1	0.1	29	156	5.4
Hepatitis	2	48	93.75	93.75	0.00	0.01	0.01	85	87	1.0
Zoo	7	54	94.44	94.44	0.00	0.1	0.1	14	41	2.9
Iris	3	90	100.00	100.00	0.00	0.01	0.01	596	602	1.0
Wine	3	107	100.00	100.00	0.00	0.1	0.01	68	662	9.7
Flag	8	116	55.13	56.67	-1.54	0.1	0.01	9	115	12.8
Sonar	2	125	81.47	81.95	-0.48	0.01	0.01	103	576	5.6
Glass	7	128	72.32	73.02	-0.70	0.1	0.01	921	6255	6.8
Voting	2	139	95.74	95.74	0.00	0.1	0.01	6	49	8.2
Heart	2	162	80.37	79.07	1.30	0.1	0.01	825	29	0.0 <sup>a</sup>
Heart (Cleaveland)	5	178	85.33	84.33	1.00	0.01	0.01	80	500	6.3
Liver (Bupa)	2	207	72.9	72.32	0.58	0.1	0.01	435	3344	7.7
Ionosphere	2	211	93.72	93.57	0.15	0.1	0.01	500	978	2.0
Image segmentation	7	252	97.86	97.5	0.36	0.1	0.01	305	2945	9.7
Vowel	11	317	90.76	89.53	1.23	0.1	0.01	999	9434	9.4
CRX	2	392	89.77	89.85	-0.08	0.01	0.001	26	569	21.9
Breast cancer (WI)	2	410	96.62	96.47	0.15	0.01	0.01	38	54	1.4
Australian	2	414	88.41	88.41	0.00	0.001	0.0001	134	1257	9.4
Pima Indians diabetes	2	461	76.41	76.73	-0.32	0.1	0.01	58	645	11.1
Vehicle	4	508	85.27	83.43	1.84	0.1	0.01	1182	9853	8.3
LED Creator	10	600	74.35	74.05	0.30	0.1	0.01	29	179	6.2
Sat	7	2661	92.02	91.65	0.37	0.01	0.001	3415	14049	4.1
Mushroom	2	3386	100.00	100.00	0.00	0.01	0.0001	20	1820	91.0
Shuttle	7	5552	99.69	97.44 +	2.25	0.1	0.0001	7033	9966	100.0 <sup>b</sup>
LED creator + 17	10	6000	73.86	73.79	0.07	0.01	0.001	6	474	79
Letter recognition	26	12000	83.53	73.25 +	10.28	0.001	0.0001	4968	9915	100.0 <sup>b</sup>
Average	6	1329	86.76	86.15	0.62	0.061	0.014	842	2867	20.03
Median	4	232	89.09	88.97	0.04	0.1	0.01	94	624	7.95

<sup>a</sup> On the *Heart* task, on-line achieved 78.52% accuracy in 26 epochs with a learning rate of 0.01.

<sup>b</sup> Batch training had not finished after 10,000 epochs for the *Shuttle* and *Letter-Recognition* tasks, but appeared to be progressing about 100 times slower than on-line.

# Semi-Batch on Digits

Learning Rate	Batch Size	Max Word Accuracy	Training Epochs
0.1	1	96.49%	21
0.1	10	96.13%	41
0.1	100	95.39%	43
0.1	1000	84.13%+	4747+
0.01	1	96.49%	27
0.01	10	96.49%	27
0.01	100	95.76%	46
0.01	1000	95.20%	1612
0.01	20,000	23.25%+	4865+
0.001	1	96.49%	402
0.001	100	96.68%	468
0.001	1000	96.13%	405
0.001	20,000	90.77%	1966
0.0001	1	96.68%	4589
0.0001	100	96.49%	5340
0.0001	1000	96.49%	5520
0.0001	20,000	96.31%	8343

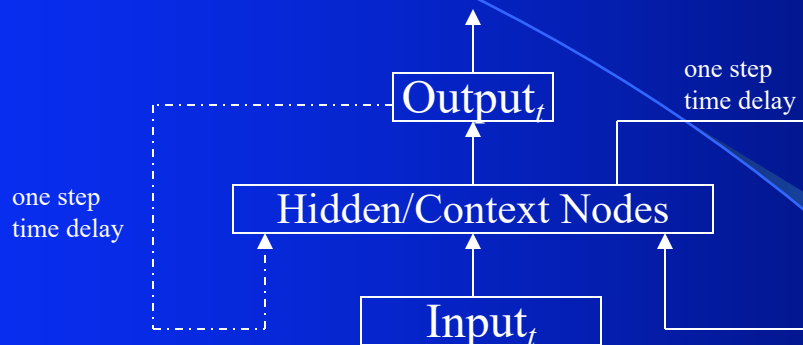
# On-Line vs. Batch Issues

- Some say just use on-line LR but divide by  $n$  (training set size) to get the same feasible LR for both (non-accumulated), but on-line still does  $n$  times as many updates per epoch as batch and is thus much faster
- True Gradient - We just have the gradient of the training set anyways which is an approximation to the true gradient and true minima
- Momentum and true gradient - same issue with other enhancements such as adaptive LR, etc.
- Training sets are getting larger - makes discrepancy worse since we would do batch update relatively less often
- Large training sets great for learning and avoiding overfit - best case scenario is huge/infinite set where never have to repeat - just 1 partial epoch and just finish when learning stabilizes – batch in this case?
- Mini-batches can be useful for algorithms which are sensitive to a bad gradient direction, and when GPU parallelism gets it for free

# Multiple Outputs

- Typical to have multiple output nodes, even with just one output feature (e.g. Iris data set)
- Would if there are multiple "independent output features"
  - Could train independent networks
  - Also common to have them share hidden layer
    - May find shared features
    - Transfer Learning
  - Could have shared and separate subsequent hidden layers, etc.
- Structured Outputs
- Multiple Output Dependency? (MOD)
  - New research area

# Recurrent Networks



- Some problems happen over time - Speech recognition, stock forecasting, target tracking, etc.
- Recurrent networks can store state (memory) which lets them learn to output based on both current and past inputs
- Learning algorithms are more complex but are becoming increasingly better at solving more complex problems (LSTM - more with deep)
- Alternatively, for some problems we can use a larger “snapshot” of features over time with standard backpropagation learning and execution (e.g. NetTalk)

# MLP/Backpropagation Summary

- Excellent Empirical results
- Scaling – The pleasant surprise
  - Local minima very rare as problem and network complexity increase
- Most common neural network approach
  - Many other different styles of neural networks (RBF, Hopfield, etc.)
- Hyper-parameters usually handled by trial and error
- Many variants
  - Adaptive Parameters, Ontogenic (growing and pruning) learning algorithms
  - Many different learning algorithm approaches
  - Recurrent networks
  - Deep networks!
  - An active research area