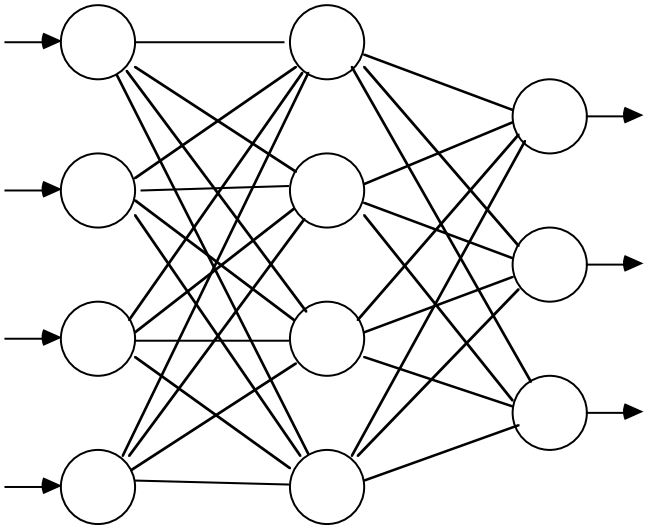
Backpropagation

Multilayer Nets?
Linear Systems

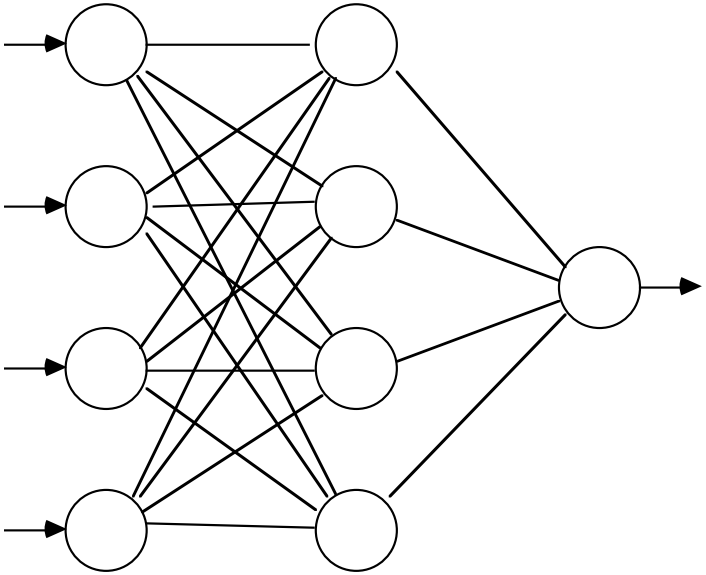
$$F(cx) = cF(x)$$
$$F(x+y) = F(x) + F(y)$$



I N M Z

$$Z = (M(NI)) = (MN)I = PI$$

Early Attempts
Committee Machine



Randomly Connected (Adaptive)

Vote Taking TLU (non-adaptive)

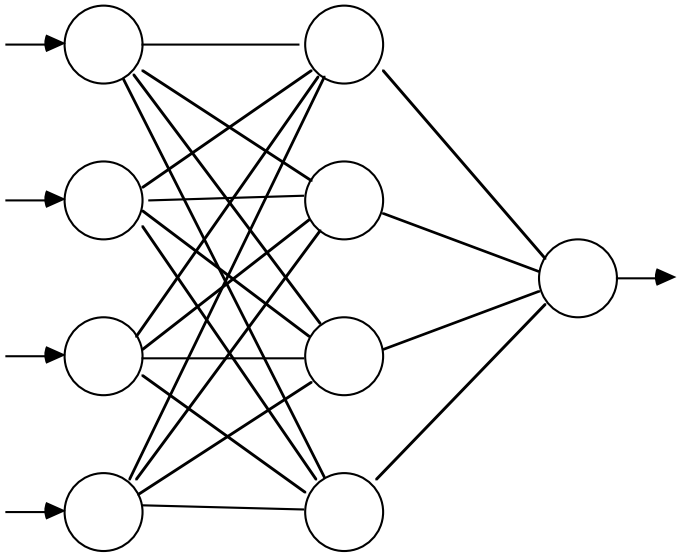
Majority Logic

"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 (Sensor) A-units (Association) R-units (Response)

Random to A-units fixed weights adaptive

Variations on Delta rule learning

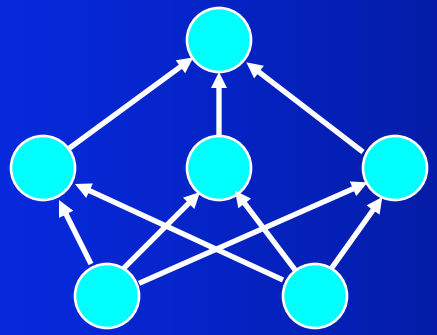
Why S-A units?

Backpropagation

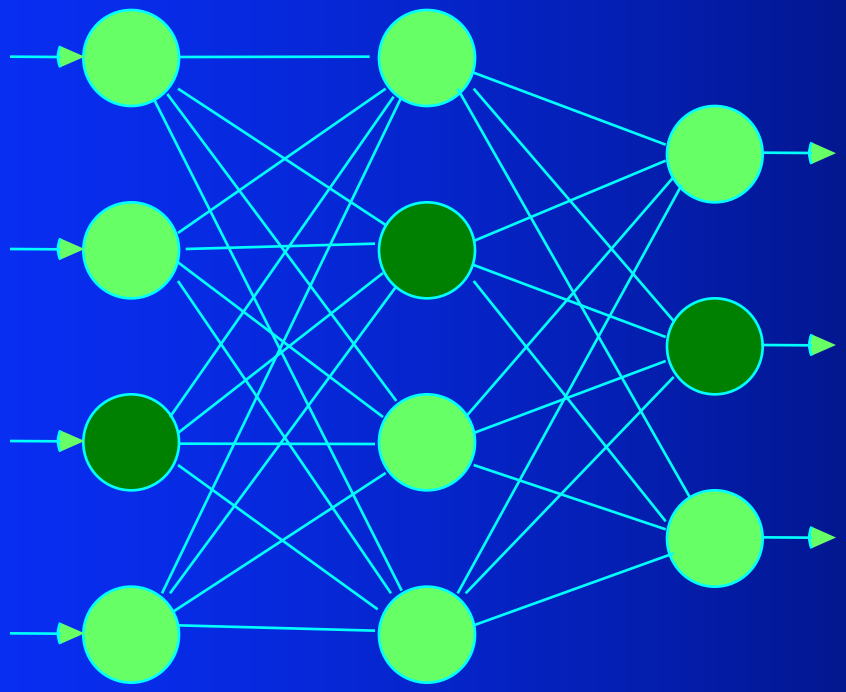
- Rumelhart (1986), Werbos (74),…, explosion of neural net interest
- Multi-layer supervised learning
- Able to train multi-layer perceptrons (and other topologies)
- Uses differentiable sigmoid function which is the smooth (squashed) version of the threshold function
- Error is propagated back through 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

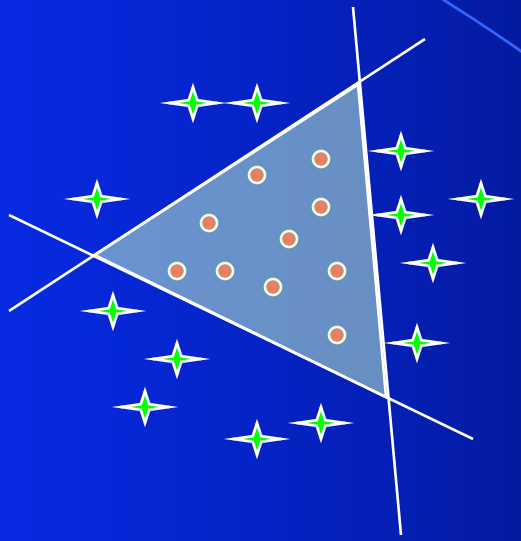


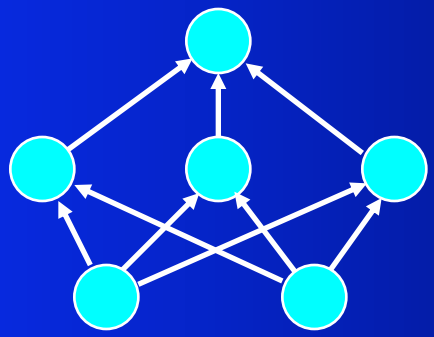
Responsibility Problem



Output 1
Wanted 0

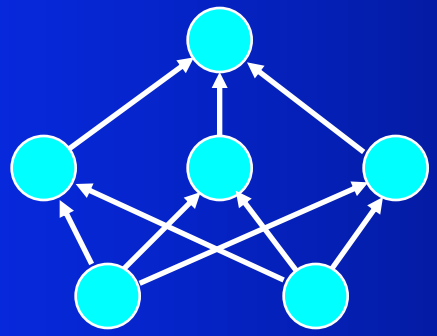
Multi-Layer Generalization



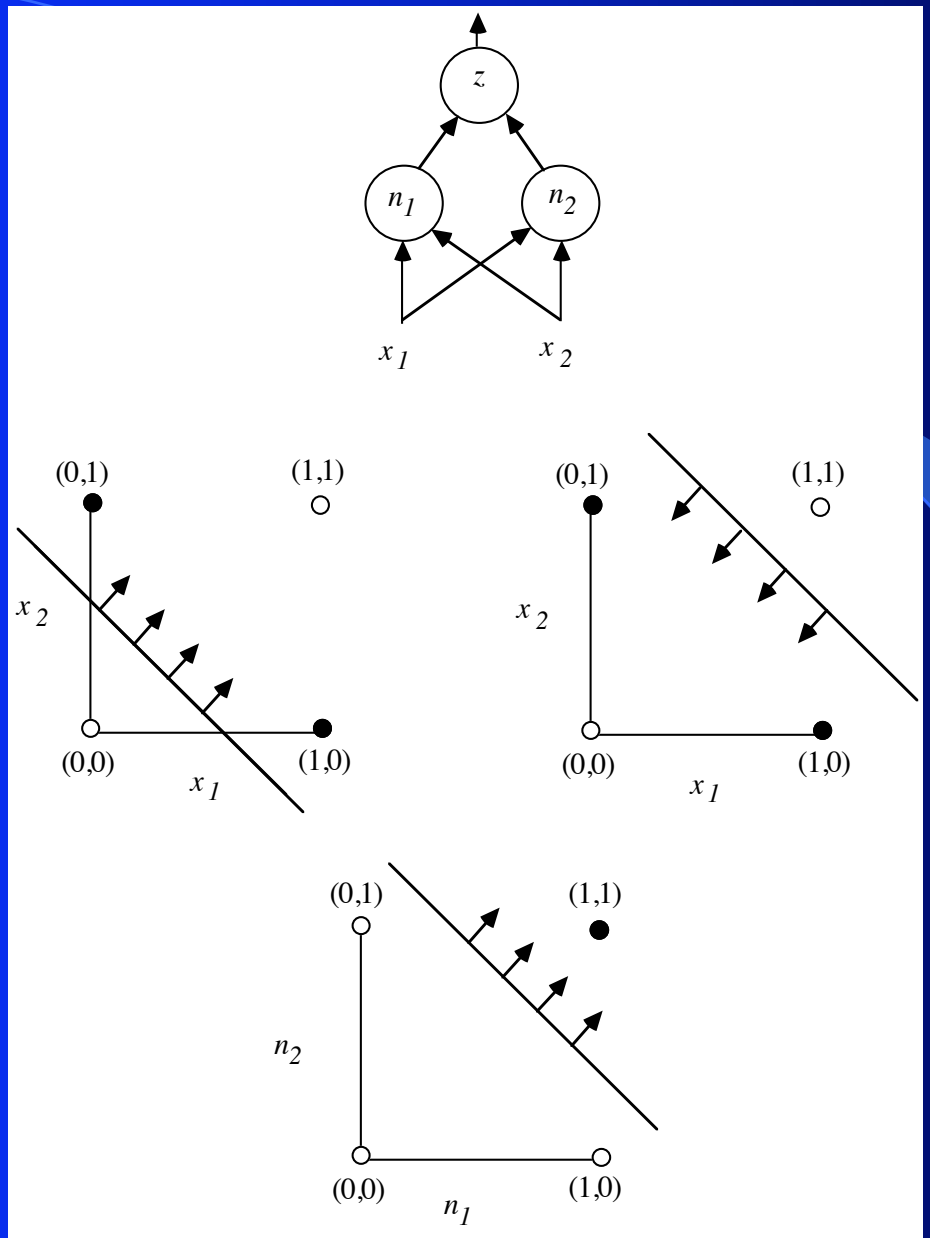


Multilayer nets are universal function approximators

- Input, output, and arbitrary number of hidden layers

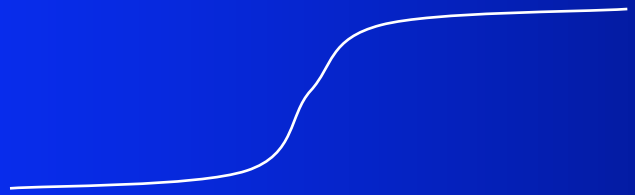


- 1 hidden layer sufficient for DNF representation of any Boolean function - One hidden node per positive conjunct, output node set to the "Or" function
- 2 hidden layers allow arbitrary number of labeled clusters
- 1 hidden layer sufficient to approximate all bounded continuous functions
- 1 hidden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

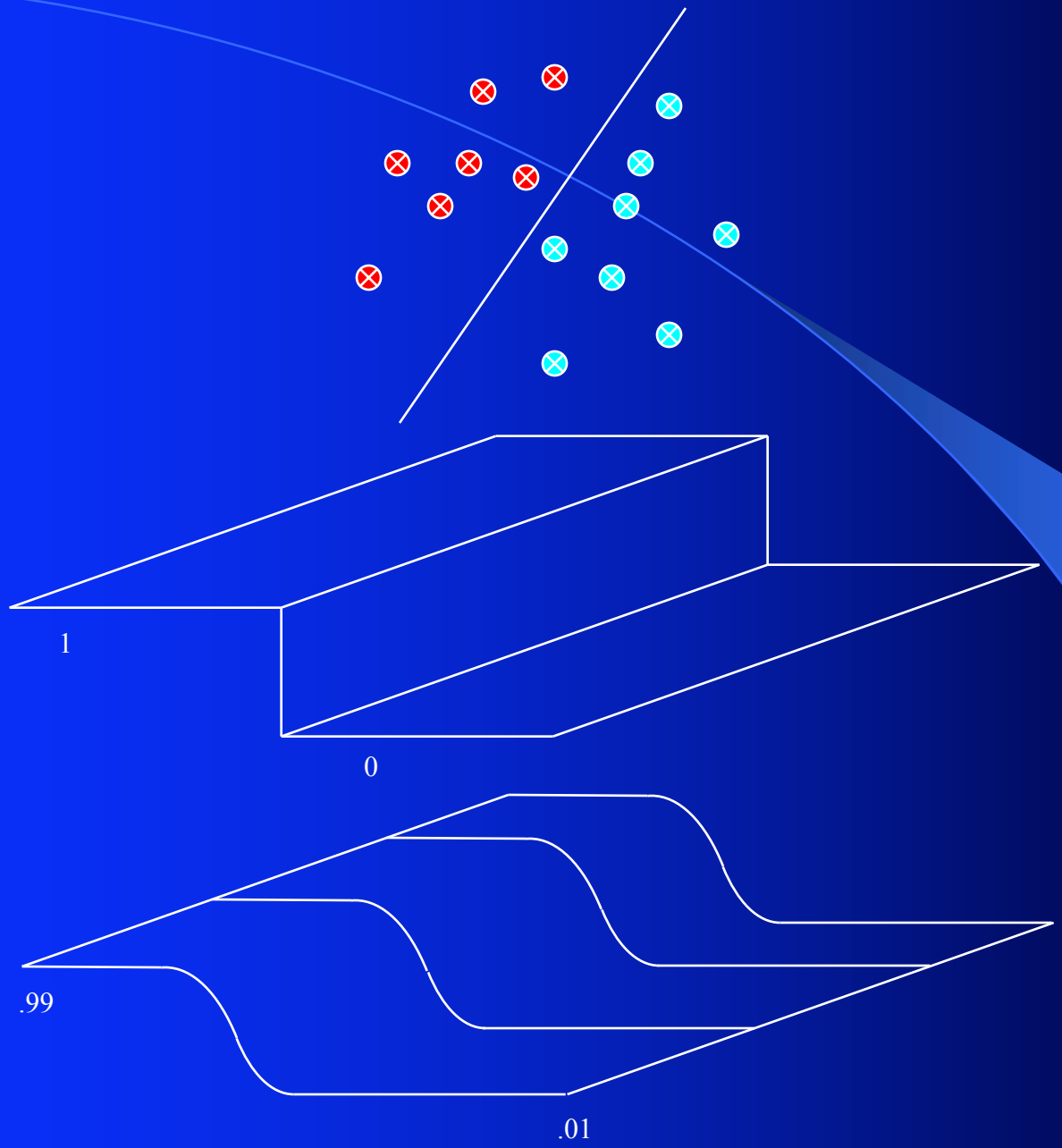

z
n_1
n_2
x_1
x_2
(0,1)
(1,1)
x_2
(0,0)
(1,0)
x_1
(0,1)
(1,1)
x_2
(0,0)
(1,0)
x_1
(0,1)
(1,1)
n_2
(0,0)
(1,0)
n_1

Backpropagation

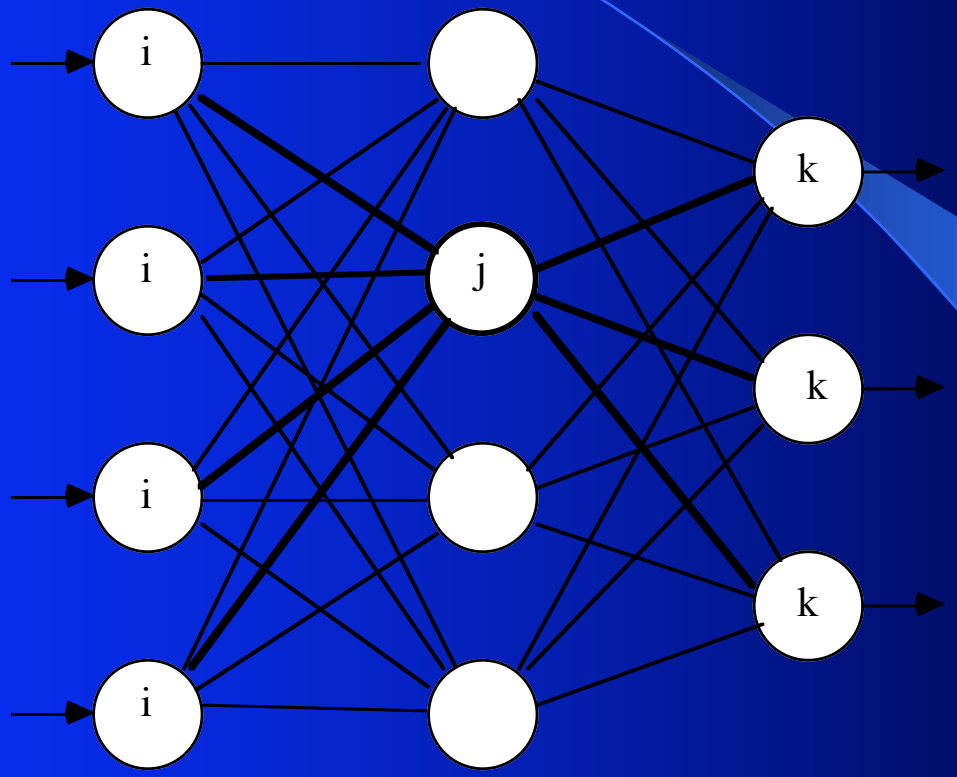
- Multi-layer supervised learner
- Gradient descent weight updates
- Sigmoid activation function (smoothed threshold logic)



- Backpropagation requires a differentiable activation function


1
0
.99
.01

Multi-layer Perceptron (MLP) Topology



Backpropagation Learning Algorithm

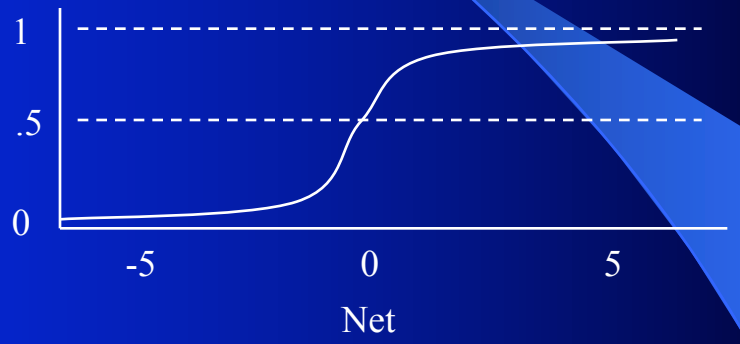
- Until Convergence (low error or other stopping criteria) do
  - Present a training pattern
  - Calculate the error of the output nodes (based on $T - Z$)
  - Calculate the error of the hidden nodes (based on the error of the output nodes which is propagated back to the hidden nodes)
  - Continue propagating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

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

Activation Function and its Derivative

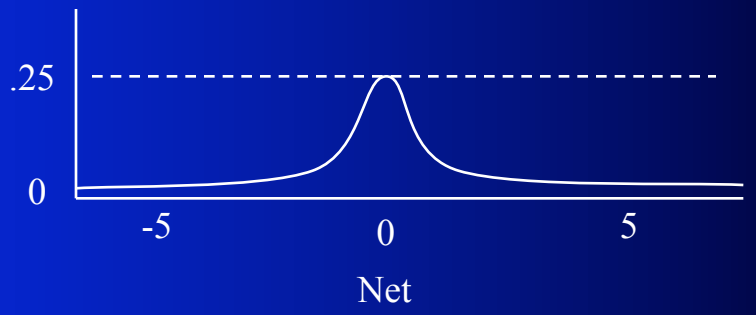
- Node activation function *f(net)* is typically the sigmoid

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



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

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

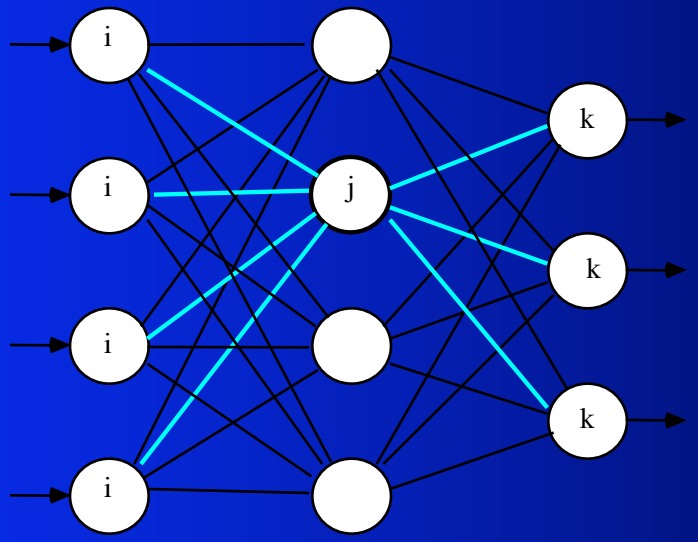


Backpropagation Learning Equations

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

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

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



Network Equations

$$Output: O_j = f(net_j) = \frac{1}{1+e^{-net_j}}$$

$$f'(net_j) = \frac{\partial O_j}{\partial net_j} = O_j(1 - O_j)$$

Δw_{ij} (general node): $C\ O_i\ \delta_j$

Δw_{ij} (output node):

$$\delta_j = (t_j - O_j)\, f'(net_j)$$

$$\Delta w_{ij} = C\ O_i\ \delta_j = C\ O_i\ (t_j - O_j)\, f'(net_j)$$

Δw_{ij} (hidden node)

$$\delta_j = \sum_k (\delta_k \cdot w_{jk})\ \ f'(net_j)$$

$$\Delta w_{ij} = C\ O_i\ \delta_j = C\ O_i\ \left(\sum_k (\delta_k \cdot w_{jk}) \right) f'(net_j)$$

Network Equations

$$Output: O_j = f(net_j) = \frac{1}{1+e^{-net_j}}$$

$$f'(net_j) = \frac{\partial O_j}{\partial net_j} = O_j(1 - O_j)$$

$$\Delta w_{ij}\ (general\ node):\ C\ O_i\ \delta_j$$

$$\Delta w_{ij}\ (output\ node):$$

$$\delta_j = (t_j - O_j)\ f'(net_j)$$

$$\Delta w_{ij} = C\ O_i\ \delta_j = C\ O_i\ (t_j - O_j)\ f'(net_j)$$

$$\Delta w_{ij}\ (hidden\ node)$$

$$\delta_j = \sum_k (\delta_k \cdot w_{jk})\ \ f'(net_j)$$

$$\Delta w_{ij} = C\ O_i\ \delta_j = C\ O_i\ \left(\sum_k (\delta_k \cdot w_{jk})\right) f'(net_j)$$

BP-1) A 2-2-1 backpropagation model has initial weights as shown. Work through one cycle of learning for the following pattern(s). Assume 0 momentum and a learning constant of 1. Round calculations to 3 significant digits to the right of the decimal. Give values for all nodes and links for activation, output, error signal, weight delta, and final weights. Nodes 4, 5, 6, and 7 are just input nodes and do not have a sigmoidal output.

For each node calculate the following (show necessary equation for each). Hint: Calculate bottom-top-bottom.

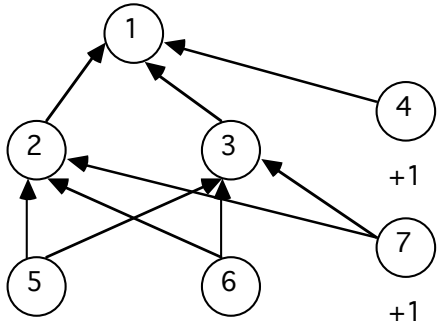
a =

o =

δ =

Δw =

w =



a) All weights initially 1.0

Training Patterns

1) 0 0 -> 1

2) 0 1 -> 0

BP-1)

$net2 = \Sigma w_i x_i = (1*0 + 1*0 + 1*1) = 1$

$net3 = 1$

$o2 = 1/(1+e^{-net}) = 1/(1+e^{-1}) = 1/(1+.368) = .731$

$o3 = .731$

$o4 = 1$

$net1 = (1*.731 + 1*.731 + 1) = 2.462$

$o1 = 1/(1+e^{-2.462}) = .921$

$\delta 1 = (t1 - o1)\, o1\, (1 - o1) = (1 - .921)\, .921\, (1 - .921) = .00575$

$\Delta w21 = \eta\, \delta_j\, o_i = \eta\, \delta 1\, o2 = 1 * .00575 * .731 = .00420$

$\Delta w31 = 1 * .00575 * .731 = .00420$

$\Delta w41 = 1 * .00575 * 1 = .00575$

$\delta 2 = o_j (1 - o_j) \Sigma \delta_k w_{jk} = o2\, (1 - o2)\ \delta 1\, w21 =$

$.731 \quad (1 - .731)\, (.00575 * 1) = .00113$

$\delta 3 = .00113$

$\Delta w52 = \eta\, \delta_j\, o_i = \eta\, \delta 2\, o5 = 1 * .00113 * 0 = 0$

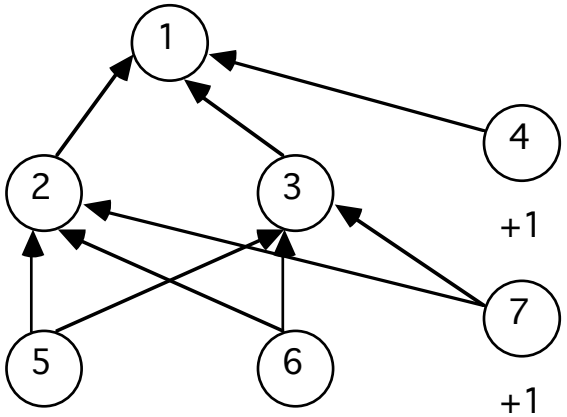
$\Delta w62 = 0$

$\Delta w72 = 1 * .00113 * 1 = .00113$

$\Delta w53 = 0$

$\Delta w63 = 0$

$\Delta w73 = 1 * .00113 * 1 = .00113$



Backprop Homework

- For your homework update the weights for the second pattern of the training set 0 1 -> 0

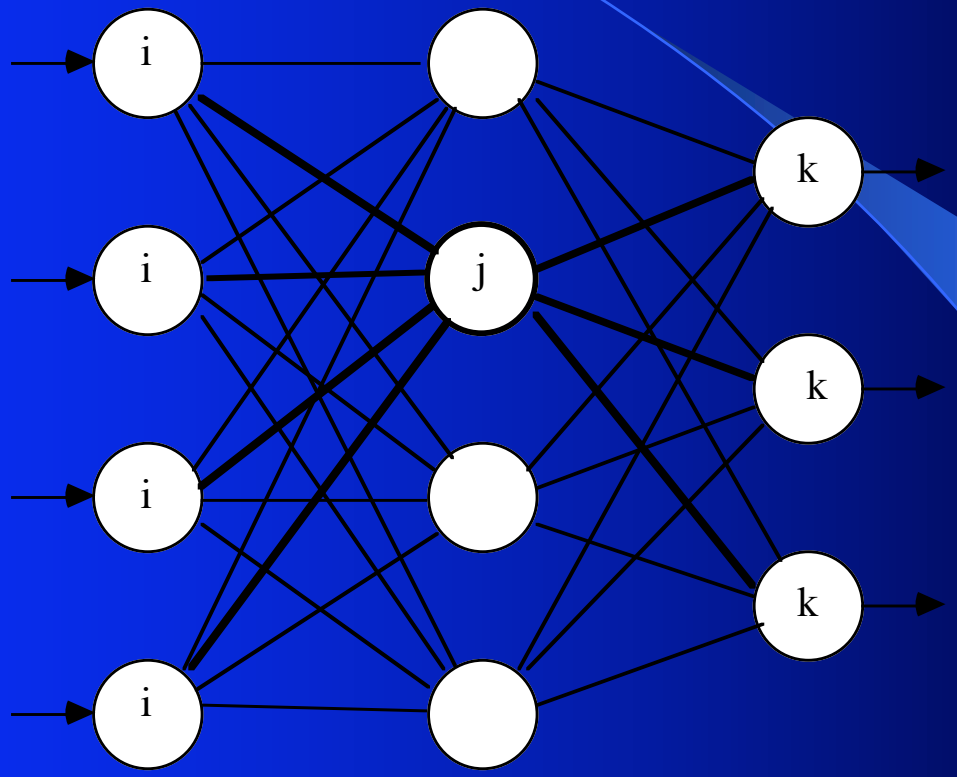
- And then go to link below: Neural Network Playground using the *tensorflow* tool and play around with the BP simulation. Try different training sets, layers, inputs, etc. and get a feel for what the nodes are doing. You do not have to hand anything in for this part.

- http://playground.tensorflow.org/

Inductive Bias & Intuition

- Node Saturation - Avoid early, but all right later
  - When saturated, an incorrect output node will still have low error
  - Start with weights close to 0
  - Saturated error even when wrong? – Multiple TSS drops
  - Not exactly 0 weights (can get stuck), random small Gaussian with 0 mean
  - Can train with target/error deltas (e.g. .1 and .9 instead of 0 and 1)
- Intuition
  - Manager/Worker Interaction
  - Gives some stability
- Inductive Bias
  - Start with simple net (small weights, initially linear changes)
  - Smoothly build a more complex surface until stopping criteria

Multi-layer Perceptron (MLP) Topology

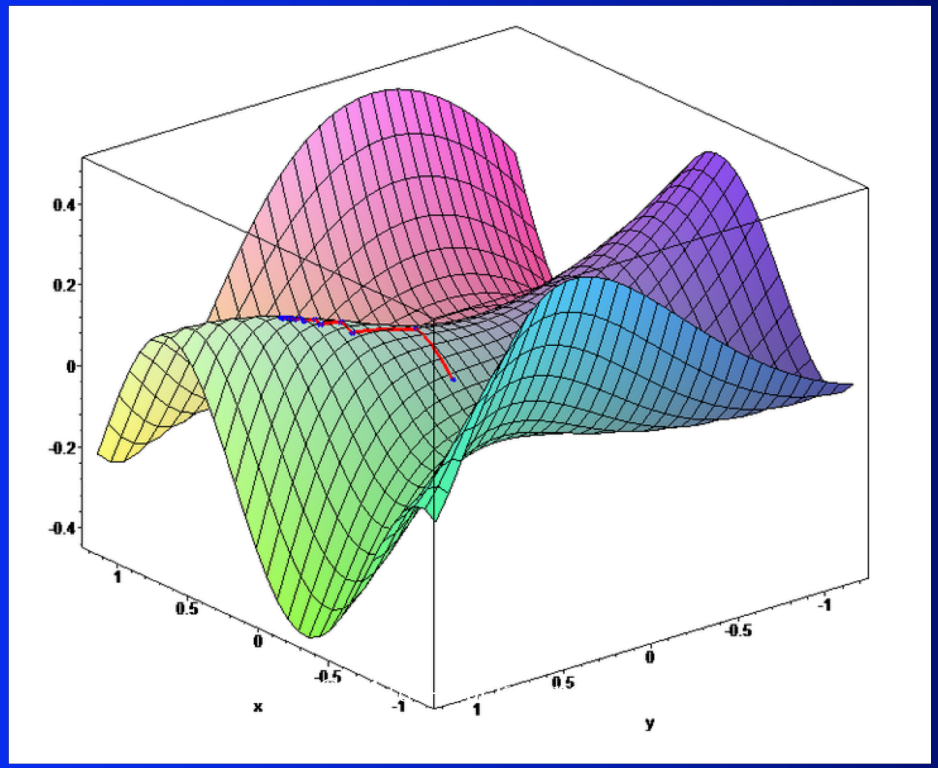


Local Minima

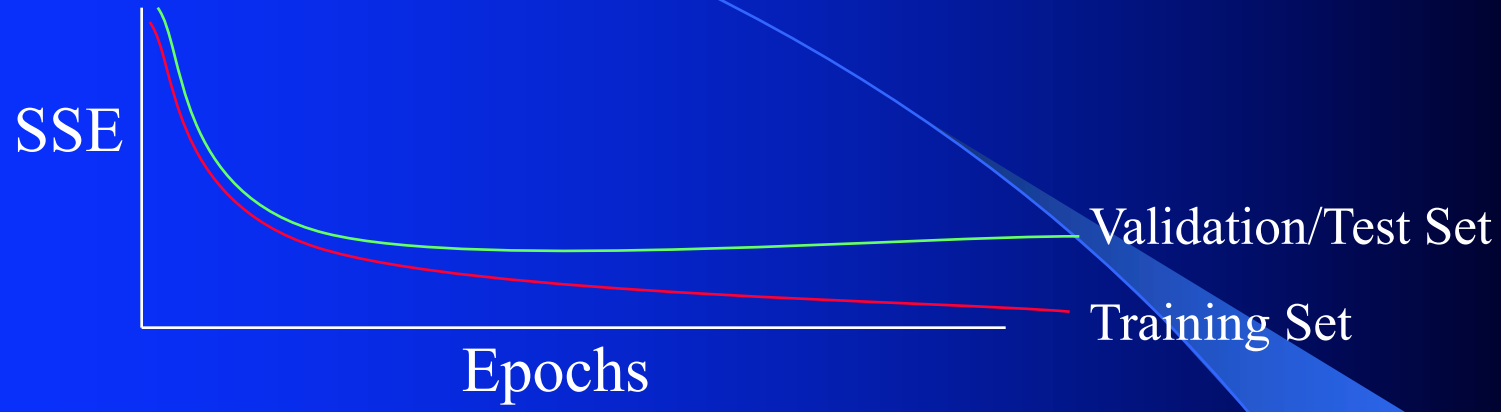
- Most algorithms which have difficulties with simple tasks get much worse with more complex tasks
- Good news with MLPs
- Many dimensions make for many descent options
- Local minima more common with simple/toy problems, rare with larger problems and larger nets
- Even if there are occasional minima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



Stopping Criteria and Overfit Avoidance



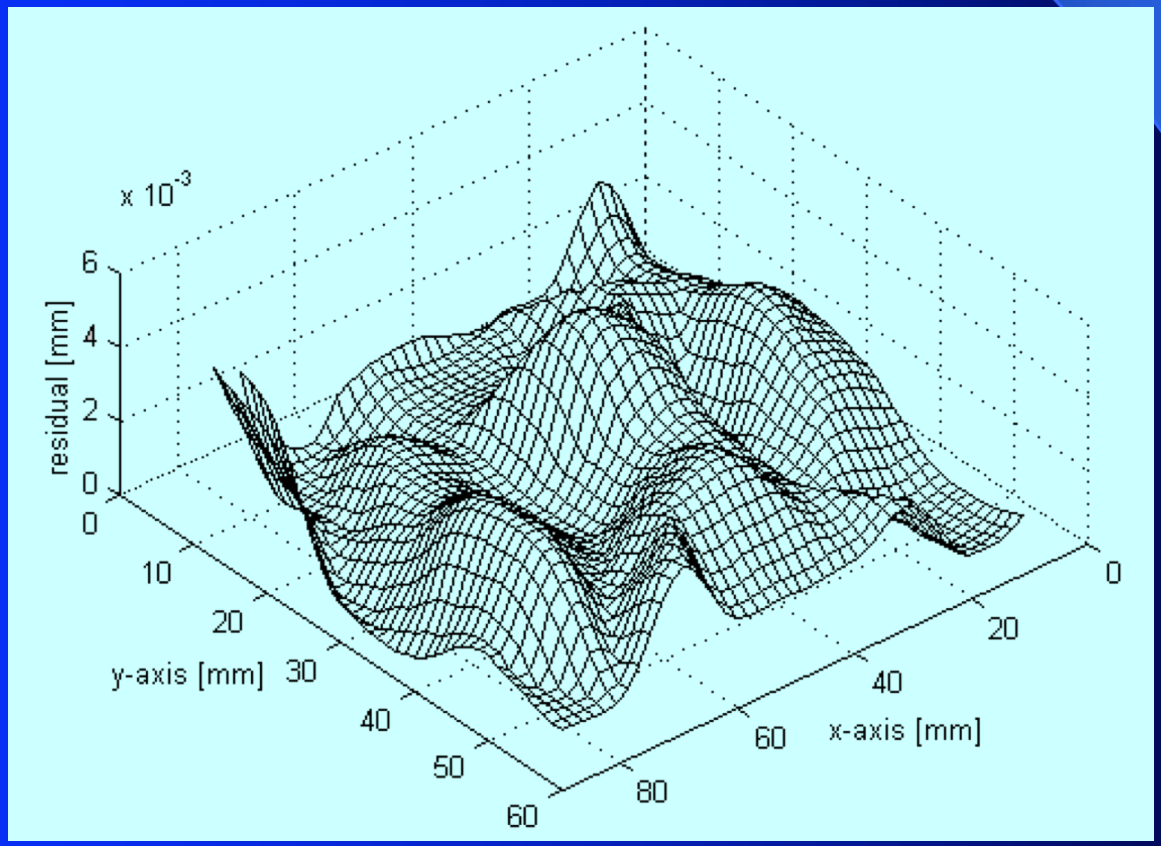
- More Training Data (vs. overtraining - One epoch limit)
- Validation Set - save weights which do best job so far on the validation set. Keep training for enough epochs to be fairly sure that no more improvement will occur (e.g. once you have trained *m* epochs with no further improvement, stop and 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 *i* of training epochs for each run. To get a final model you can train on all the data and stop after the average number of epochs, or a little less than the average since there is more data.
- Specific BP techniques for avoiding overfit
  - Less hidden nodes not a great approach because may underfit
  - Weight decay (later), Error deltas, Dropout (discuss with ensembles)

Validation Set - ML Manager

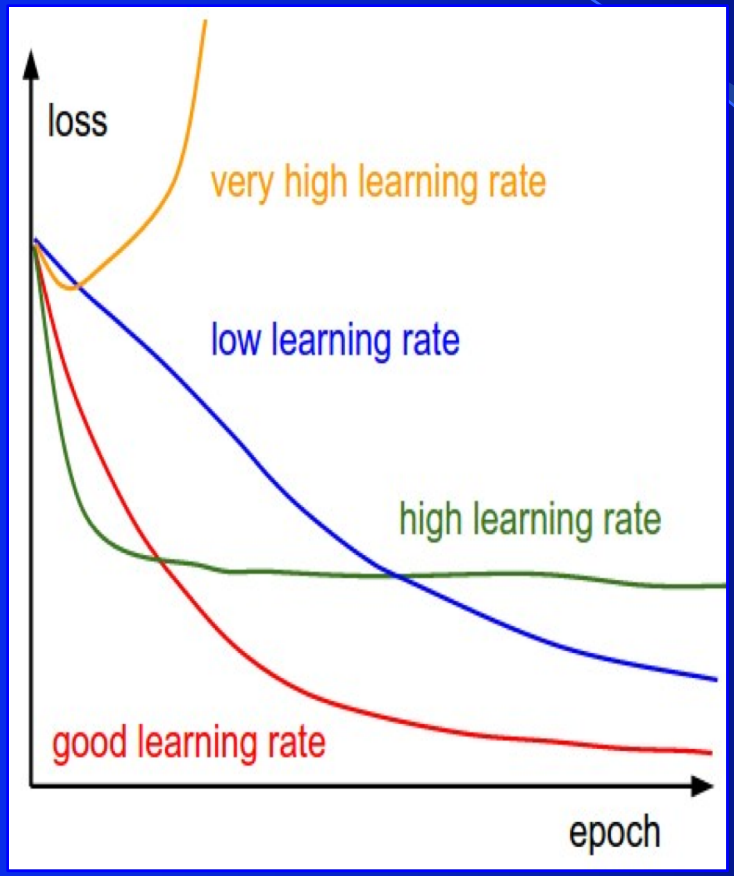
- Often you will use a validation set (separate from the training or test set) for stopping criteria, etc.
- In these cases you should take the validation set out of the training set which has already been allocated by the ML manager.
- For example, you might use the random test set method to randomly break the original data set into 80% training set and 20% test set. Independent and subsequent to the above routines you would take *n*% of the training set to be a validation set for that particular training exercise.
- 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 insert an instance into the VS which has been used to train weights

Learning Rate

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



Learning Rate



Momentum

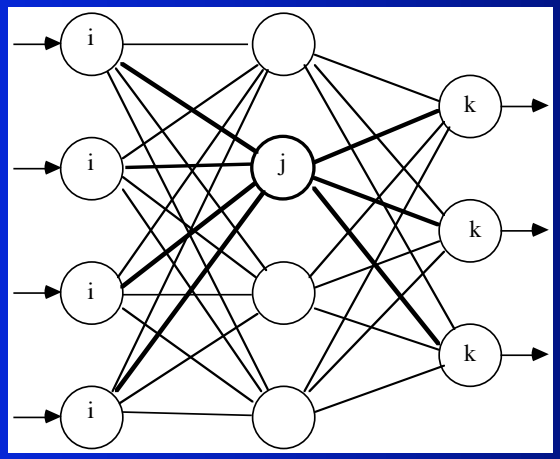
- Simple speed-up modification (type of adaptive learning rate)

$$\Delta w(t+1) = C\delta x_i + \alpha \Delta w(t)$$

- Weight update maintains momentum in the direction it has been going
  - Faster in flats
  - Could leap past minima (good or bad)
  - 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).
- These types of terms make the algorithm less pure in terms of gradient descent. In fact, for SGD (Stochastic Gradient Descent), type of mini-batch to average gradient
  - Not a big issue in overcoming local minima
  - Not a big issue in entering bad local minima

Number of Hidden Nodes

- How many needed is a function of how hard the task is
- Typically one fully connected hidden layer. Common initial number is $2n$ or $2\log n$ hidden nodes where n is the number of inputs
- In practice train with a small number of hidden nodes, then keep doubling, etc. until no more significant improvement on test sets
  - Too many nodes make learning slower, could overfit
    - Somewhat too many hidden nodes is preferable if using a reasonable stopping criteria, to make sure you don't underfit
  - Too few will underfit
- All output and hidden nodes should have bias weights



Hyperparameter Selection

- LR (e.g. .1)
- Momentum – (.5 … .99)
- Connectivity: typically 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 common method to set hyperparameters: trial and error runs
  - Often sequential: find one hyperparameter value with others held constant, freeze it, find next hyperparameter, etc.
- Can also do an automated search: Grid, Random, Bayesian – Random is empirically most consistently effective – typically each hyperparameter is chosen with a uniform distribution from a log scale for each trial
- Hyperparameters could be learned by the learning algorithm in which case you must take care to not overfit the training data

BP Lab

- Go over Lab together

Debugging your ML algorithms

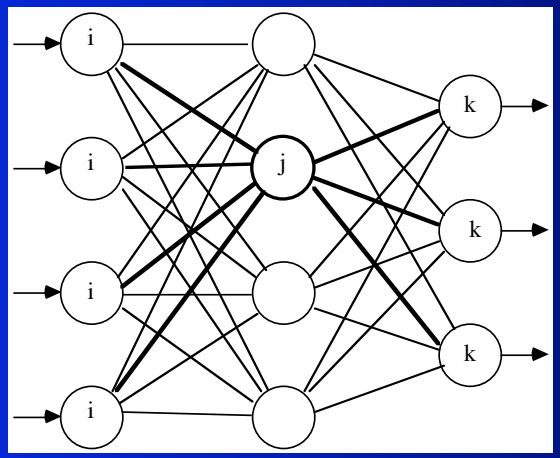
- Debugging ML algorithms is difficult
  - Unsure beforehand about what the results should be
  - Adaptive algorithm can learn to compensate somewhat for bugs (sign wrong on bias update, etc.)
  - Bugs in accuracy evaluation code common – false hopes!
- Do a small example by hand (e.g. your homework) and make sure your algorithm gets the exact same results (and accuracy)
- Compare results with our supplied snippets in learning suite
- Compare results (not code, etc.) with classmates
- Compare results with a published version of the algorithms (e.g. WEKA), 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

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 always be at 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 show that batch is less efficient – more in 678

What are the Hidden Nodes Doing?

- Hidden nodes discover new *higher order* features which are fed into the output layer
- Zipser - Linguistics
- Compression



Multiple Outputs

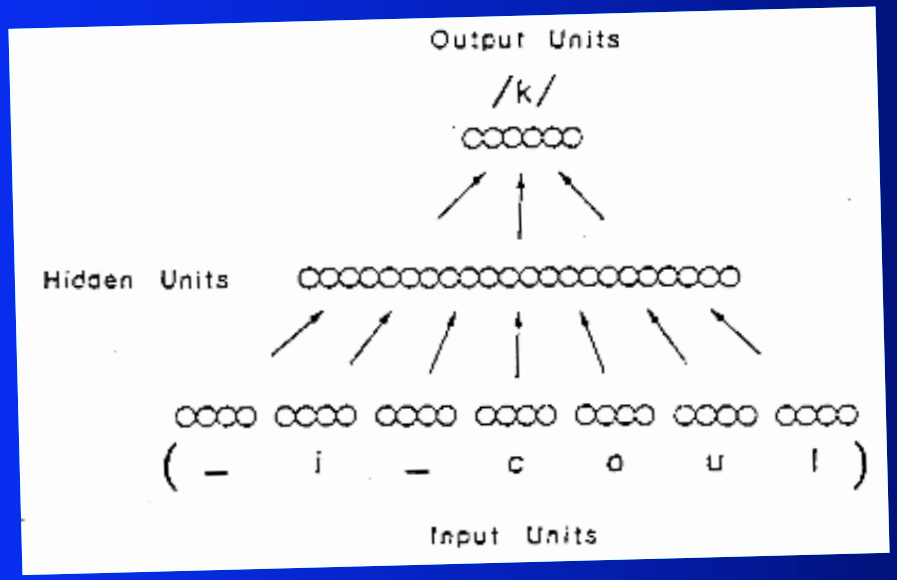
- Typical to have multiple output nodes, even with just one output feature (e.g. Iris data set)
- Would if there are multiple "independent output features"
  - Could train independent networks
  - Also common to have them share hidden layer
    - May find shared features
    - Transfer Learning
  - Could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

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

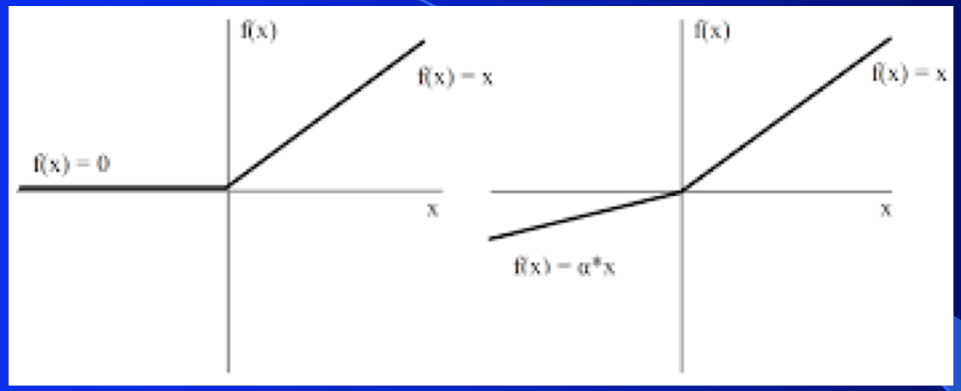
- One of first application attempts
- Train a neural network to read English aloud
- Input Layer - Localist representation of letters and punctuation
- Output layer - Distributed representation of phonemes
- 120 hidden units: 98% correct pronunciation
  - Note steady progression from simple to more complex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



Regression with MLP/BP

- Replace output node with a linear activation (i.e. identity which just passes the *net* value through) which more naturally supports unconstrained regression
- Since $f'(net)$ is 1 for the linear activation, the output error is just $(target - output)$
- Hidden nodes still use a standard non-linear activation function (such as sigmoid) with the standard $f'(net)$

Rectified Linear Units



- BP can work with any differentiable non-linear activation function (e.g. sine)
- *ReLU* is common these days: $f(x) = \text{Max}(0,x)$. More efficient gradient propagation, derivative is 0 or constant, just fold into learning rate
- More efficient computation: Only comparison, addition and multiplication.
  - Leaky ReLU $f(x) = x$ if $x > 0$ else ax, where $0 \leq a <= 1$, so that derivate is not 0 and can do some learning for net < 0 (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

Softmax and Cross-Entropy

- Sum-squared error (L2) loss gradient 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. Fine for regression but less natural for classification.
- For *classification* problems it is advantageous and increasingly popular to use the *softmax* activation function, just at the output layer, with the cross-entropy loss function. Softmax (softens) 1 of *n* targets to mimic a probability vector for each output.

$$f(net_j) = \frac{e^{net_j}}{\sum_{i=1}^{n} e^{net_i}}$$

- Cross entropy seeks to 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 (softmax goal). Maximizing likelihood is cast as the equivalent minimizing of the negative log likelihood.

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

- With new loss and activation functions, we must recalculate the gradient equation. Gradient/Error on the output is just $(t\text{-}z)$, no $f'(net)$! The exponent of softmax is unraveled by the ln of cross entropy. Helps avoid gradient saturation.
- Common with deep networks, with ReLU activations for the hidden nodes

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
- Regularization approach: Model (h) selection
  - Minimize $F(h) = Error(h) + \lambda \cdot Complexity(h)$
  - Tradeoff accuracy vs complexity
- 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)
- 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:
  - $L(\boldsymbol{w}) = \text{Error}(\boldsymbol{w}) + \lambda \Sigma |w_i|$
  - λ is a hyperparameter which controls how much we value model simplicity vs training set accuracy
  - Gradient of $L(\boldsymbol{w})$: Gradient of $\text{Error}(\boldsymbol{w}) + \lambda$
    - This is also called weight decay
    - Gradient of Error is just equations we have used if $\text{Error}(\boldsymbol{w})$ is TSS, but differs for other error functions
- Common values for lambda are 0, .01, .03 (3% of weight is decayed each time), 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

- $L(\boldsymbol{w}) = \text{Error}(\boldsymbol{w}) + \lambda \Sigma w_i^2$
- Gradient of $L(\boldsymbol{w})$: Gradient of $\text{Error}(\boldsymbol{w}) + 2\lambda w_i$
- Regularization portion of weight update is scaled by weight value (fold 2 into λ)
  - Decreases change when weight small (<0), otherwise increases
- 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 acts more dramatically with large weights
  - L1 leads to simpler models, but L2 often more accurate with more complex problems which require a bit more complexity

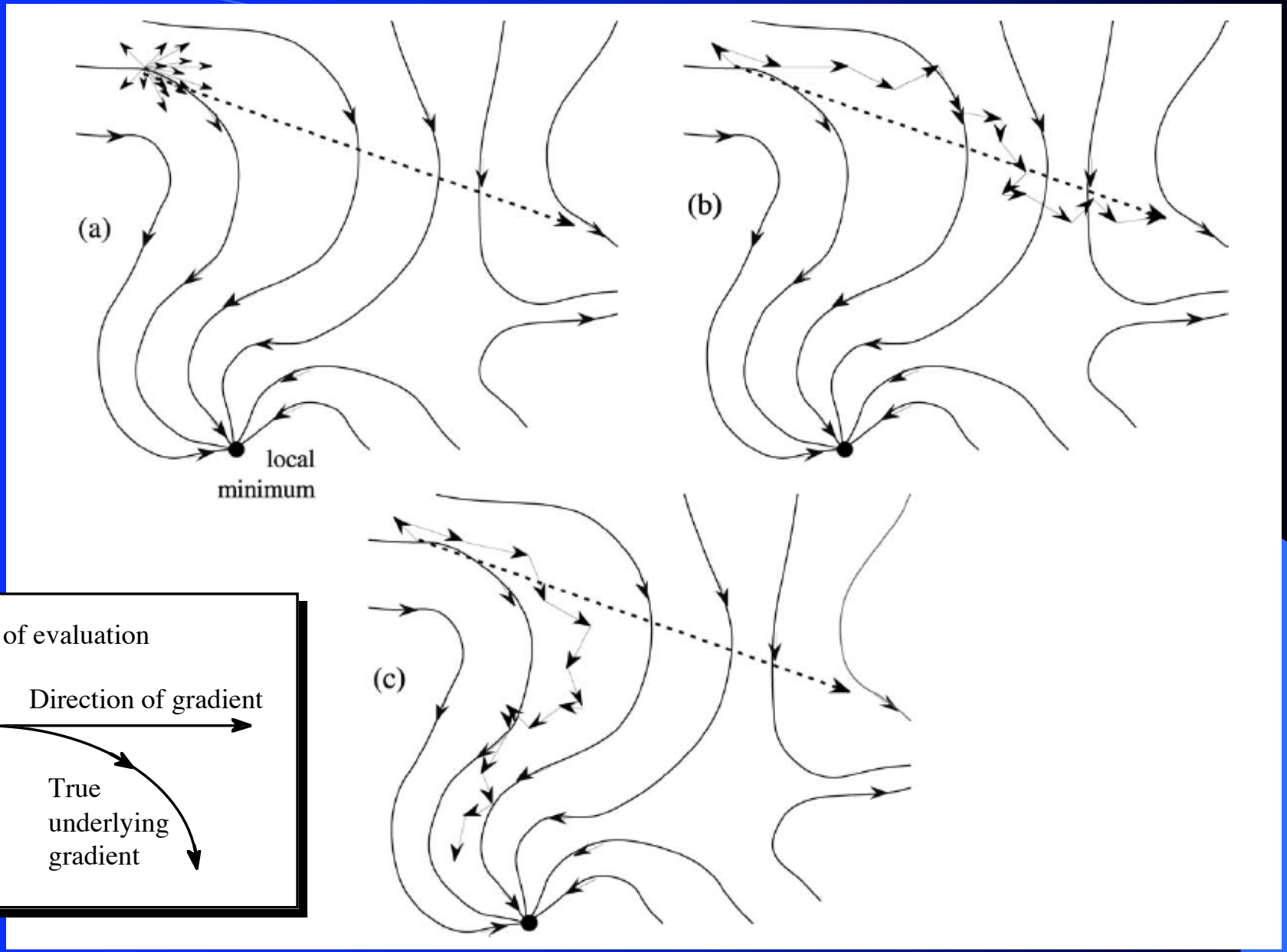
Batch Update

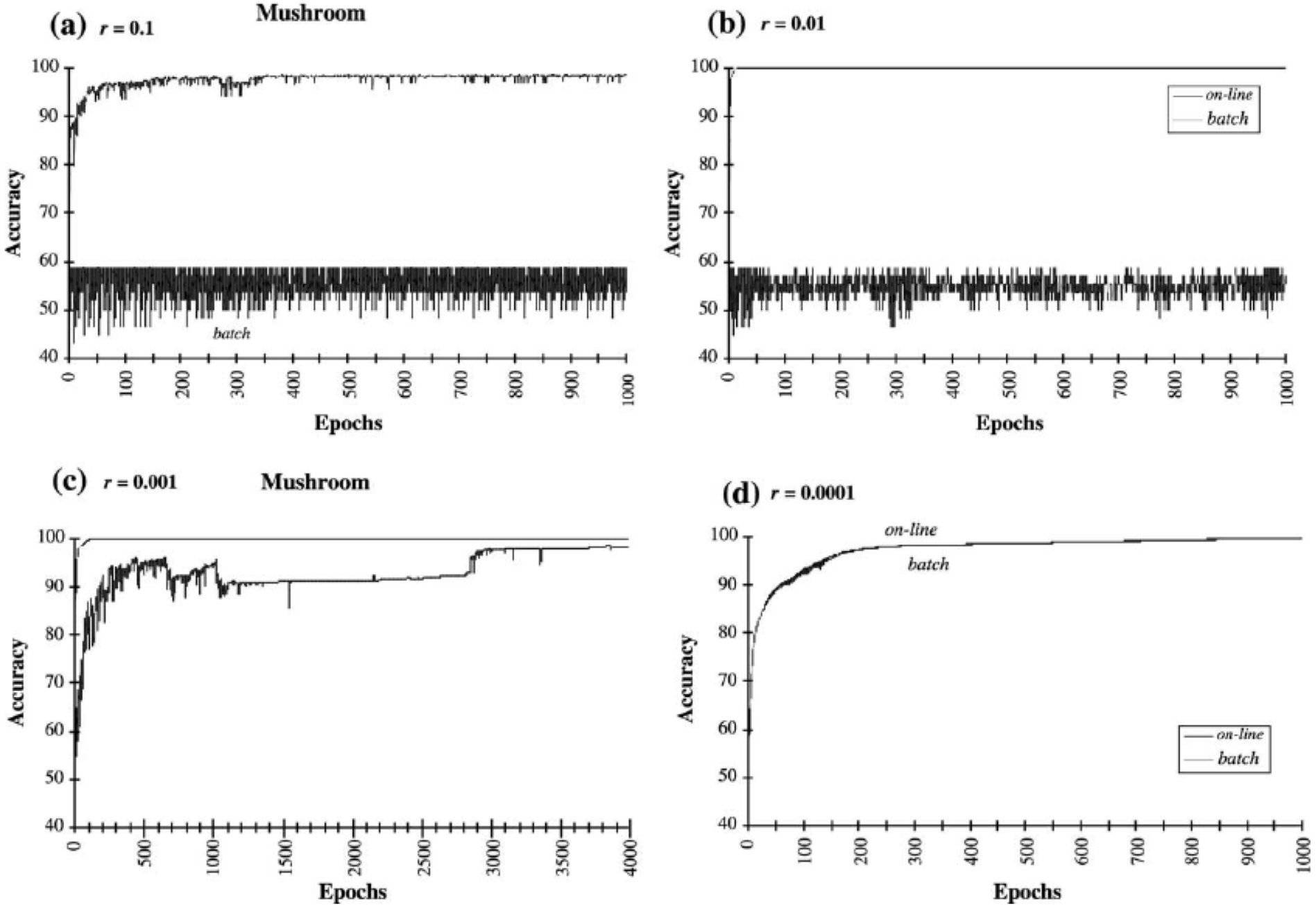
- With On-line (stochastic) update we update weights after every pattern
- With Batch update we accumulate the changes for each weight, but do not update them until the end of each epoch
- Batch update gives a correct direction of the gradient for the entire data set, while on-line could do some weight 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"

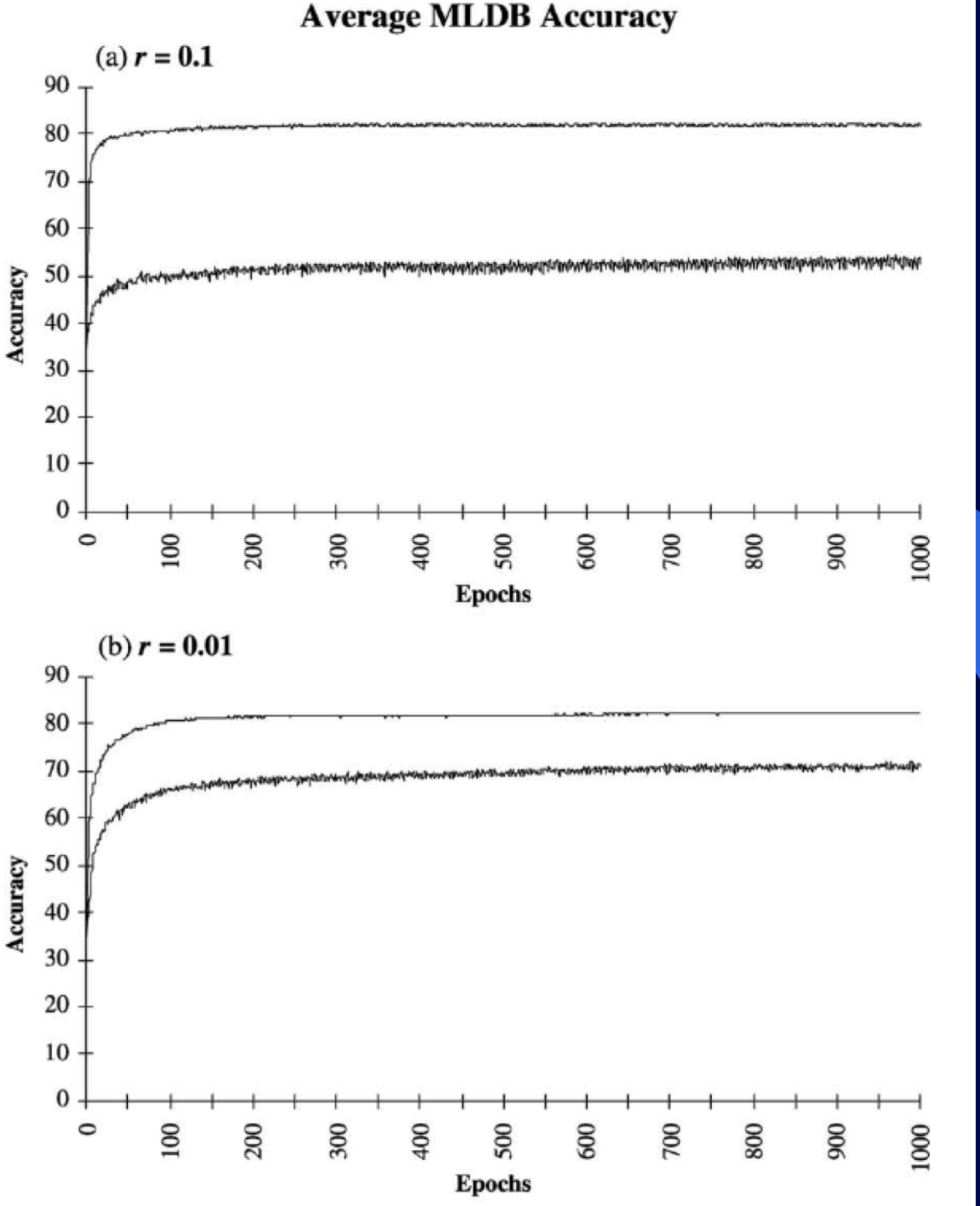
On-Line vs. Batch

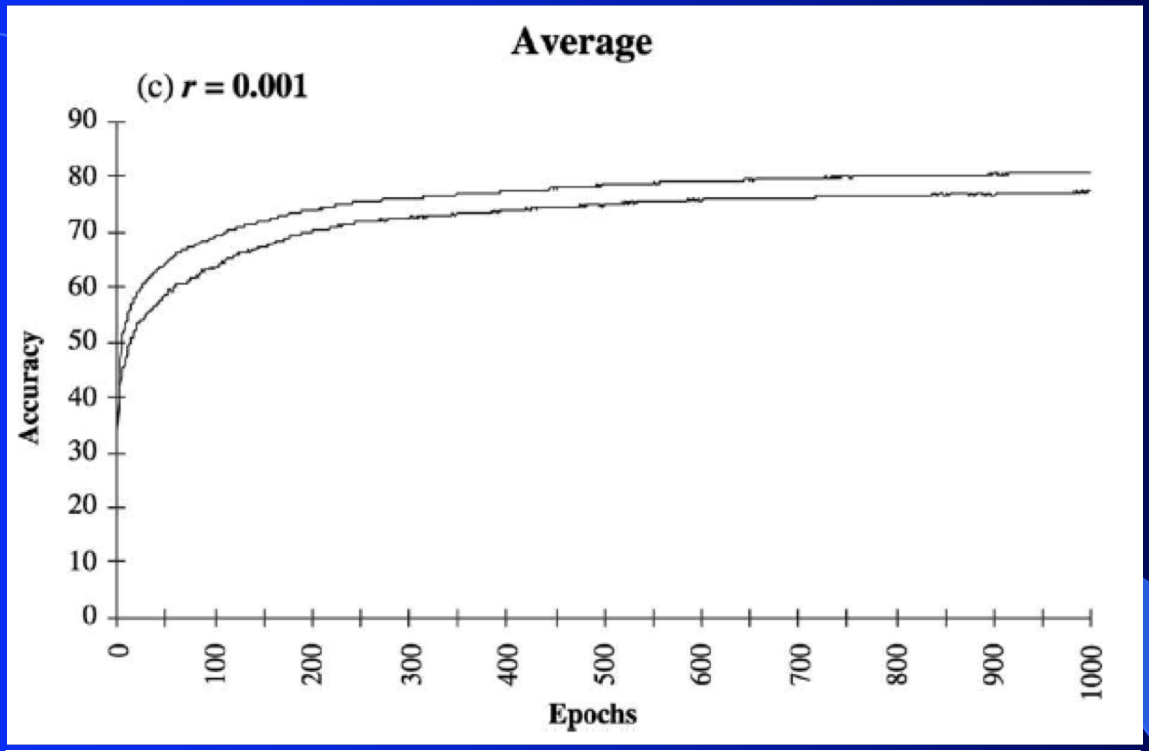
Wilson, D. R. and Martinez, T. R., The General Inefficiency of Batch Training for Gradient Descent Learning, *Neural Networks*, vol. **16**, no. 10, pp. 1429-1452, 2003

- 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


(a)
local
minimum
(b)
(c)
Point of evaluation
Direction of gradient
True
underlying
gradient


(a) r = 0.1
Mushroom
Accuracy
Epochs
batch
(b) r = 0.01
on-line
batch
(c) r = 0.001
Mushroom
(d) r = 0.0001
on-line
batch


Average MLDB Accuracy
(a) r = 0.1
Accuracy
Epochs
0
10
20
30
40
50
60
70
80
90
0
100
200
300
400
500
600
700
800
900
1000
(b) r = 0.01
Accuracy
Epochs


Average
(c) r = 0.001
Accuracy
Epochs
0
10
20
30
40
50
60
70
80
90
0
100
200
300
400
500
600
700
800
900
1000

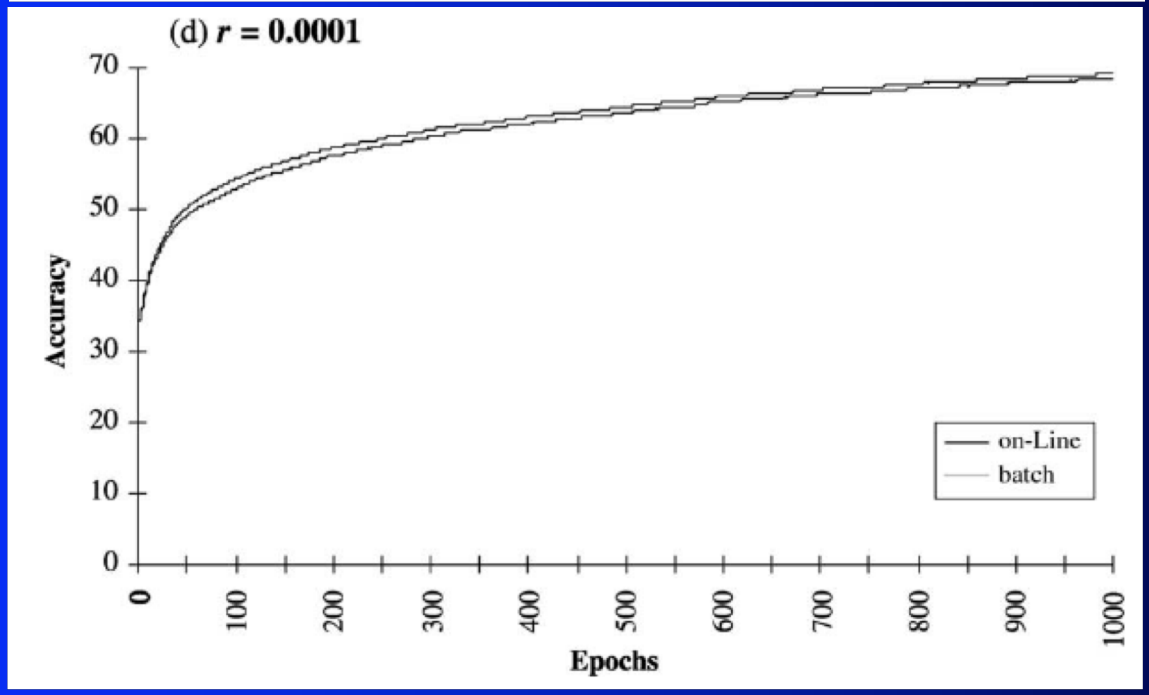

(d) r = 0.0001
Accuracy
Epochs
on-Line
batch
0
10
20
30
40
50
60
70
0
100
200
300
400
500
600
700
800
900
1000

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[a]
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[b]
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[b]
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

[a] On the *Heart* task, on-line achieved 78.52% accuracy in 26 epochs with a learning rate of 0.01.
[b] 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?
- Still difficult to convince some people

Adaptive Learning Rate/Momentum

- Momentum is a simple speed-up modification

$$\Delta w(t+1) = C\delta x_i + \alpha \Delta w(t)$$

- Are we true gradient descent when using this?
  - Note it is kind of a mini-batch following the local avg gradient
- Weight update maintains momentum in the direction it has been going
  - Faster in flats
  - Could leap past minima (good or bad), but not a big issue in practice
  - Significant speed-up, common value $\alpha \approx .9$
  - Effectively increases learning rate in areas where the gradient is consistently the same sign.
  - 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 …

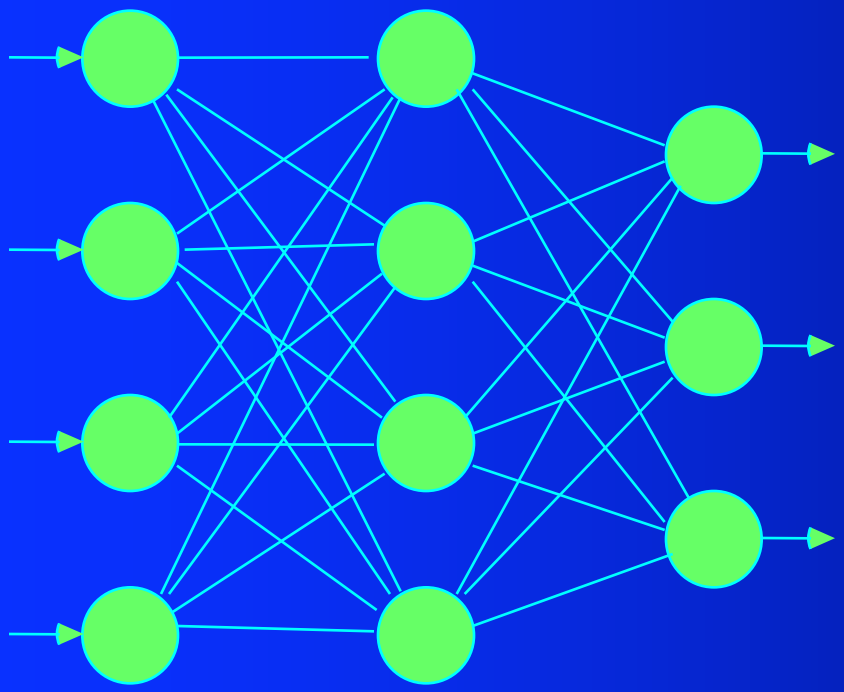
Speed up variations of SGD

- Use mini-batch rather than single instance for better gradient estimate – *Sometimes* helpful if using GD 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(sum(historical values)) – LRs with smaller derivatives are decreased less
  - 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) (1st and 2nd moments) – updates based on a moving average of both

Learning Variations

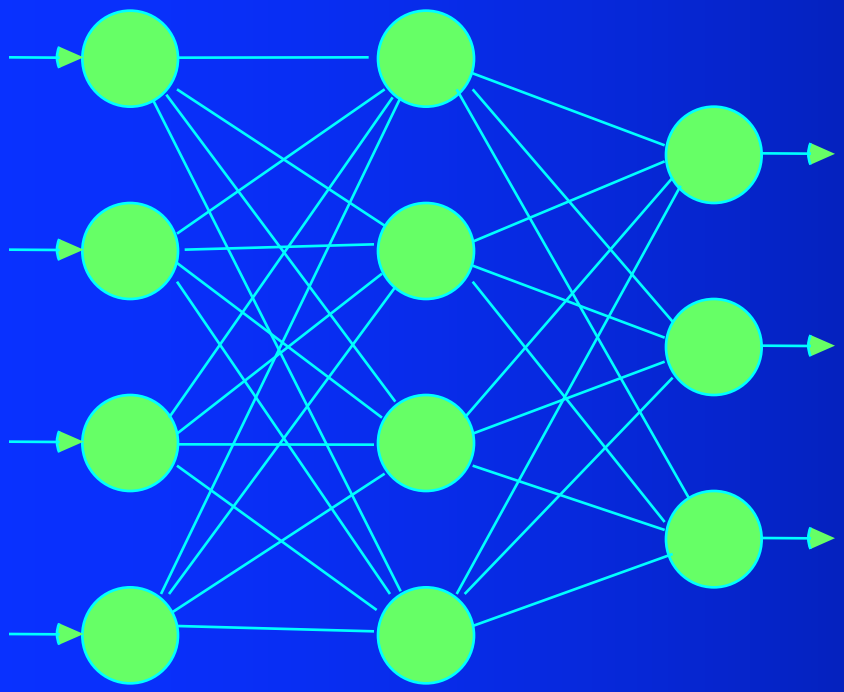
- Different activation functions - need only be differentiable
- Different objective functions
  - Cross-Entropy
  - Classification Based Learning
- Higher Order Algorithms - 2nd derivatives (Hessian Matrix)
  - Quickprop
  - Conjugate Gradient
  - Newton Methods
- Constructive Networks
  - Cascade Correlation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

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

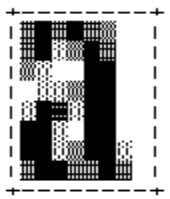


Target	Actual	BP Error	CB Error
1	.6	$.4*f'(net)$	.1
0	.7	$-.7*f'(net)$	-.1
0	.3	$-.3*f'(net)$	0

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
```

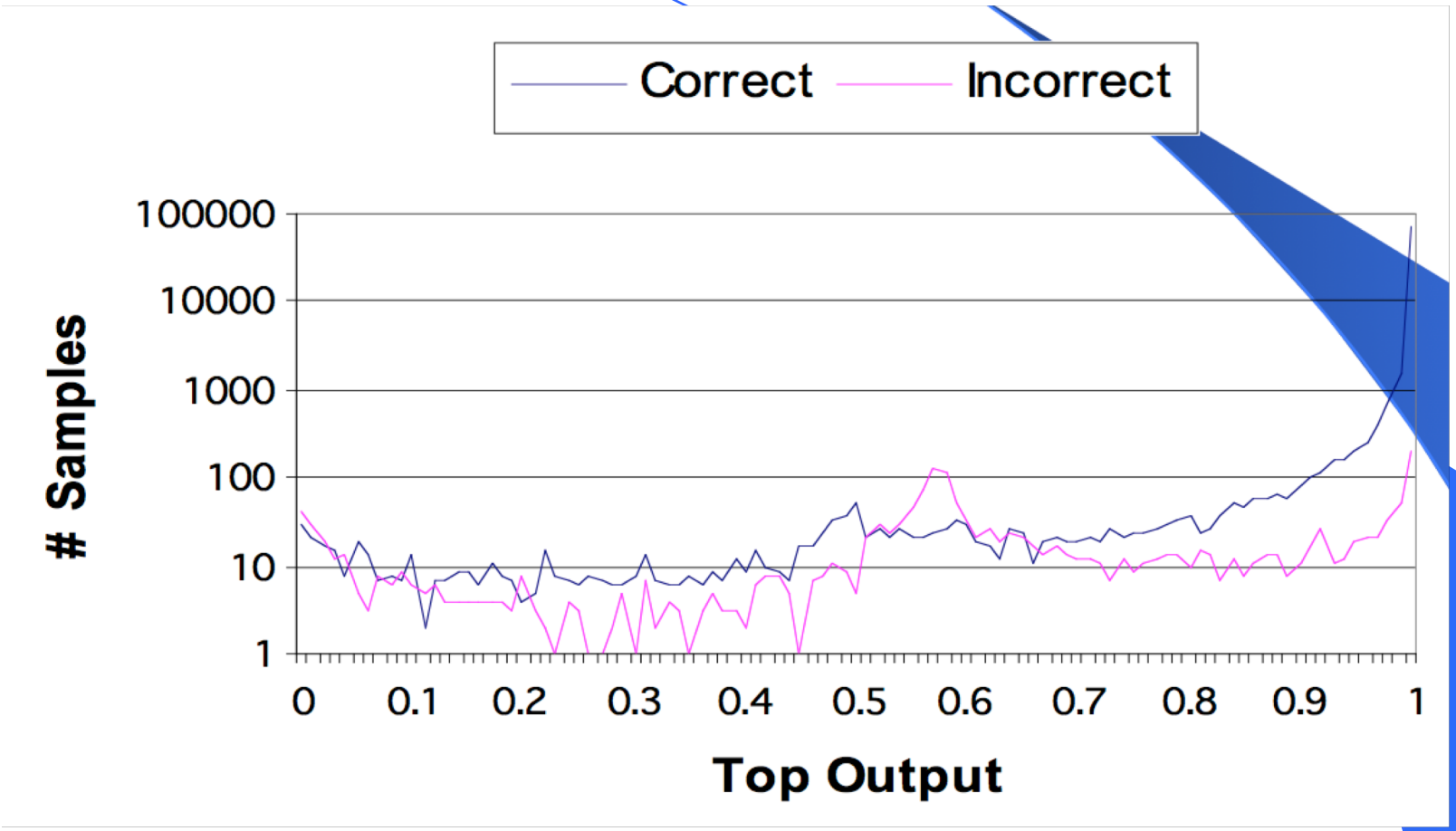
Results

- Classification Based Training:
  99.1%

```
 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:   OoO    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
```

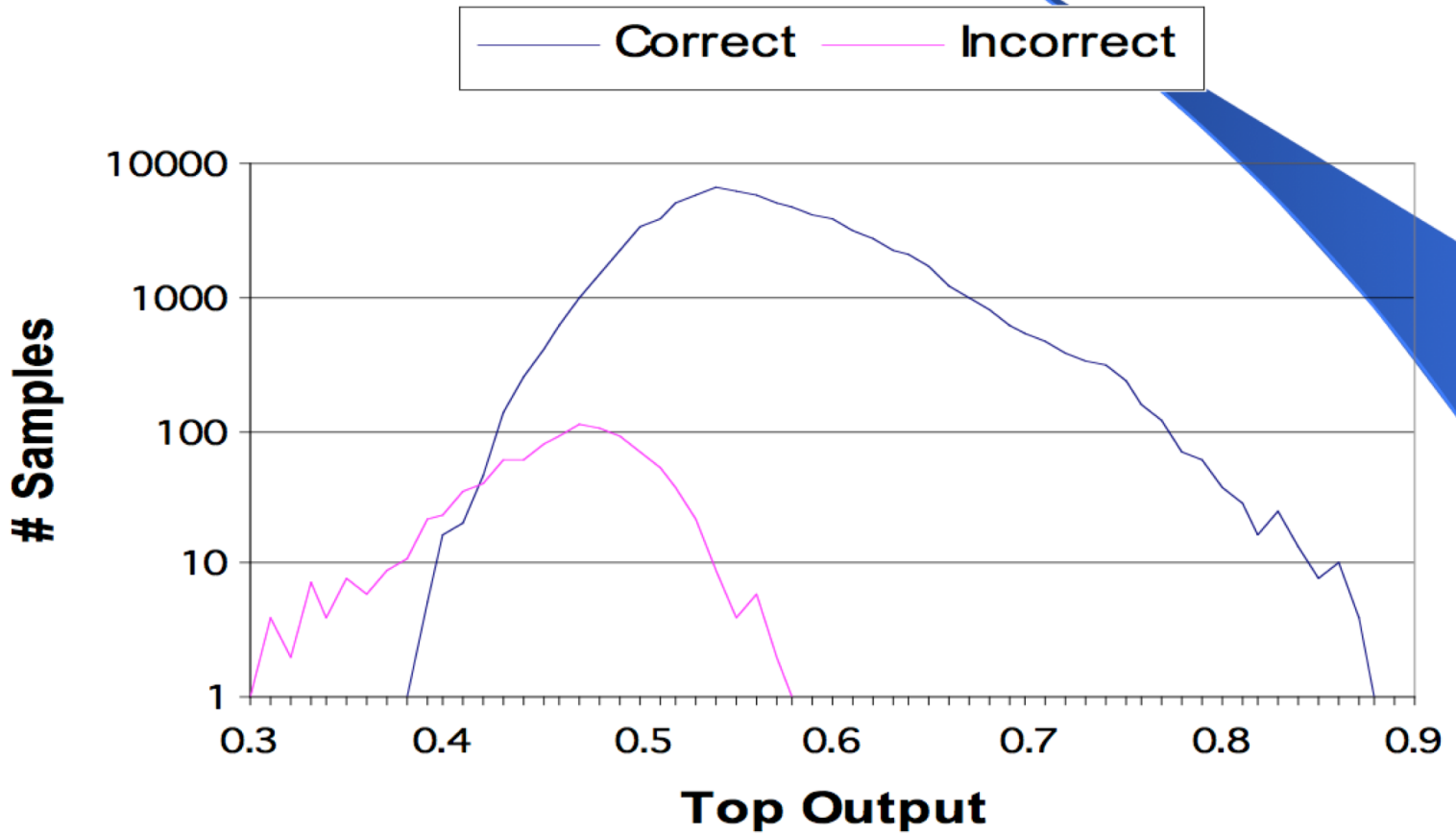
Sample Output:

Analysis



Network outputs on test set after standard backpropagation training.

Analysis

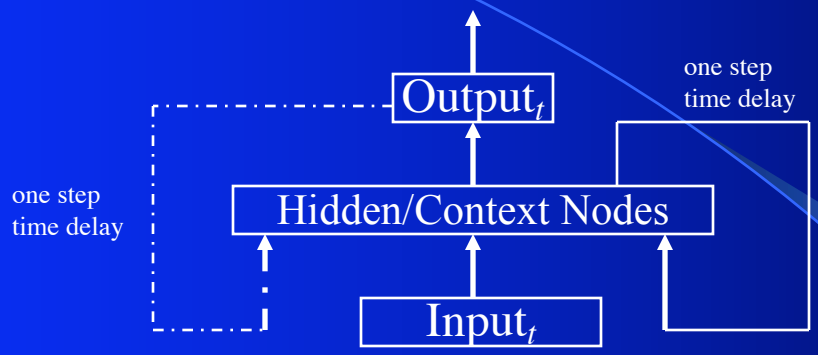


Network outputs on test set after CB training.

Classification Based Models

- CB1: Only backpropagates error on misclassified training patterns
- CB2: Adds a confidence margin, μ, 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

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
- Alternatively, for some problems we can use a larger "snapshot" of features over time with standard backpropagation learning and execution (e.g. NetTalk)

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.)
- User defined parameters usually handled by multiple experiments
- Many variants
  - Regression – Typically Linear output nodes, normal hidden nodes
  - Adaptive Parameters, Ontogenic (growing and pruning) learning algorithms
  - Many different learning algorithm approaches
  - Higher order gradient descent (Newton, Conjugate Gradient, etc.)
  - Recurrent networks
  - Deep networks!
  - Still an active research area