MLPs with Backpropagation
Multilayer Nets?
Linear Systems

\[ F(cx) = cF(x) \]
\[ F(x+y) = F(x) + F(y) \]

\[ Z = (M(NI)) = (MN)I = PI \]
Early Attempts
Committee Machine

Randomly Connected Vote Taking TLU
(Adaptive) (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)
- 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!
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

\[ \Delta w_{ij} = C \delta_j Z_i \]

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

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

Output: \( O_j = f(\text{net}_j) = \frac{1}{1 + e^{-\text{net}_j}} \)

\( f(\text{net}_j) = \frac{\partial O_j}{\partial \text{net}_j} = O_j(1 - O_j) \)

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

\( \Delta w_{ij} \) (output node):
\[
\delta_j = (t_j - O_j) \ f'(\text{net}_j)
\]
\[
\Delta w_{ij} = C \ O_i \ \delta_j = C \ O_i \ (t_j - O_j) \ f'(\text{net}_j)
\]

\( \Delta w_{ij} \) (hidden node)
\[
\delta_j = \sum_k (\delta_k \cdot w_{jk}) \ f'(\text{net}_j)
\]
\[
\Delta w_{ij} = C \ O_i \ \delta_j = C \ O_i \ (\sum_k (\delta_k \cdot w_{jk}) ) \ f'(\text{net}_j)
\]
Network Equations

Output: \( O_j = f(\text{net}_j) = \frac{1}{1+e^{-\text{net}_j}} \)

\( f(\text{net}_j) = \frac{\partial O_j}{\partial \text{net}_j} = O_j(1 - O_j) \)

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

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

\[ \delta_j = (t_j - O_j) \, f'(\text{net}_j) \]

\[ \Delta w_{ij} = C \, O_i \, \delta_j = C \, O_i \, (t_j - O_j) \, f'(\text{net}_j) \]

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

\[ \delta_j = \sum_k (\delta_k \cdot w_{jk}) \, f'(\text{net}_j) \]

\[ \Delta w_{ij} = C \, O_i \, \delta_j = C \, O_i \, (\sum_k (\delta_k \cdot w_{jk})) \, f'(\text{net}_j) \]

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) All weights initially 1.0

Training Patterns
1) 0 0 -> 1
2) 0 1 -> 0
BP-1)

\[ \text{net}_2 = \sum w_i x_i = (1*0 + 1*0 + 1*1) = 1 \]

\[ \text{net}_3 = 1 \]

\[ o_2 = 1/(1+e^{\text{net}}) = 1/(1+e^{-1}) = 1/(1+.368) = .731 \]

\[ o_3 = .731 \]

\[ o_4 = 1 \]

\[ \text{net}_1 = (1*.731 + 1*.731 + 1) = 2.462 \]

\[ o_1 = 1/(1+e^{-2.462})= .921 \]

\[ \delta_1 = (t_1 - o_1) o_1 (1 - o_1) = (1 - .921) .921 (1 - .921) = .00575 \]

\[ \Delta w_{21} = \eta \delta_j o_i = \eta \delta_1 o_2 = 1 * .00575 * .731 = .00420 \]

\[ \Delta w_{31} = 1 * .00575 * .731 = .00420 \]

\[ \Delta w_{41} = 1 * .00575 * 1 = .00575 \]

\[ \delta_2 = o_j (1 - o_j) \sum \delta_k w_{jk} = o_2 (1 - o_2) \delta_1 w_{21} = .731 (1 - .731) (.00575 * 1) = .00113 \]

\[ \delta_3 = .00113 \]

\[ \Delta w_{52} = \eta \delta_j o_i = \eta \delta_2 o_5 = 1 * .00113 * 0 = 0 \]

\[ \Delta w_{62} = 0 \]

\[ \Delta w_{72} = 1 * .00113 * 1 = .00113 \]

\[ \Delta w_{53} = 0 \]

\[ \Delta w_{63} = 0 \]

\[ \Delta w_{73} = 1 * .00113 * 1 = .00113 \]
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

Input Layer  Hidden Layer(s)  Output Layer
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

- 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 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 (0.01 - 0.5 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

- very high learning rate
- low learning rate
- high learning rate
- good learning rate

(loss vs. epoch)
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.
- 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 can choose 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], ...
  - User chooses how many iterations to try (better time control)
- Random is often best because
  - 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/or Random Search on scikit-learn
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](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

![Diagram of neural network architecture]
Regression with MLP/BP

- Replace output node with a linear activation (i.e. identity which just passes the \textit{net} value through) which more naturally supports unconstrained regression.
- Since \( f'(\text{net}) \) is 1 for the linear activation, the output error is just \( (\text{target} - \text{output}) \).
- Hidden nodes still use a standard non-linear activation function (such as sigmoid) with the standard \( f'(\text{net}) \).
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 \( f(x) = \) if \( x > 0 \) else \( ax \), where \( 0 \leq a \leq 1 \), so that derivative 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.

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

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

\[
\text{Loss}_{\text{CrossEntropy}} = -\sum_{i=1}^{n} t_i \ln(z_i)
\]

- With new loss and activation functions, we must recalculate the gradient equation. Gradient/Error on the output is just \( (t-z) \), no \( f'(\text{net}) \)! 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
- 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 $F(h) = \text{Error}(h) + \lambda \cdot \text{Complexity}(h)$
  - 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: \( (-\Delta \text{error}(\mathbf{w}) - \lambda) \)
    - This is also called weight decay
    - Decay magnitude towards 0 – initial was \( |w_i| \) thus do \( \lambda \) sign accordingly
    - 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, .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

\[ L(w) = \text{Error}(w) + \lambda \sum w_i^2 \]

- Gradient of \( L(w) \): -Gradient of Error(\( w \)) - 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 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
Machine Learning Tools

Lots of new Machine Learning Tools
- Weka was the first main site with lots of ready to run models
- Scikit-learn now very popular
- Languages:
  - Python with NumPy, matplotlib, pandas, other libraries
  - R (good statistical packages), but with growing Python libraries...
- Deep Learning Neural Network frameworks – GPU capabilities
  - Tensorflow - Google
  - PyTorch – Multiple developers (Facebook, twitter, Nvidia...) - Python
  - Others: Caffe2 (Facebook), Keras, Theano, CNTK (Microsoft)
- Data Mining Business packages – Visualization, Expensive

Great for experimenting and implementing

But important to “get under the hood” and not just be black box ML users

Tensorflow neural network playground
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


- 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
Direction of gradient

True underlying gradient

Point of evaluation

local minimum

(a)  

(b)  

(c)
Average MLDB Accuracy

(a) \( r = 0.1 \)

(b) \( r = 0.01 \)
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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Output classes</th>
<th>Training set size</th>
<th>Max accuracy</th>
<th>Best LR</th>
<th>Epochs needed</th>
<th>On-line’s speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>On-line</td>
<td>Batch</td>
<td>Diff. On-line</td>
<td>Batch</td>
</tr>
<tr>
<td>Bridges</td>
<td>7</td>
<td>42</td>
<td>82.14</td>
<td>82.86</td>
<td>-0.72</td>
<td>0.1</td>
</tr>
<tr>
<td>Hepatitis</td>
<td>2</td>
<td>48</td>
<td>93.75</td>
<td>93.75</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Zoo</td>
<td>7</td>
<td>54</td>
<td>94.44</td>
<td>94.44</td>
<td>0.00</td>
<td>0.1</td>
</tr>
<tr>
<td>Iris</td>
<td>3</td>
<td>90</td>
<td>100.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Wine</td>
<td>3</td>
<td>107</td>
<td>100.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.1</td>
</tr>
<tr>
<td>Flag</td>
<td>8</td>
<td>116</td>
<td>55.13</td>
<td>56.67</td>
<td>-1.54</td>
<td>0.1</td>
</tr>
<tr>
<td>Sonar</td>
<td>2</td>
<td>125</td>
<td>81.47</td>
<td>81.95</td>
<td>-0.48</td>
<td>0.01</td>
</tr>
<tr>
<td>Glass</td>
<td>7</td>
<td>128</td>
<td>72.32</td>
<td>73.02</td>
<td>-0.70</td>
<td>0.1</td>
</tr>
<tr>
<td>Voting</td>
<td>2</td>
<td>139</td>
<td>95.74</td>
<td>95.74</td>
<td>0.00</td>
<td>0.1</td>
</tr>
<tr>
<td>Heart</td>
<td>2</td>
<td>162</td>
<td>80.37</td>
<td>79.07</td>
<td>1.30</td>
<td>0.1</td>
</tr>
<tr>
<td>Heart (Cleveland)</td>
<td>5</td>
<td>178</td>
<td>85.33</td>
<td>84.33</td>
<td>1.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Liver (Bupa)</td>
<td>2</td>
<td>207</td>
<td>72.9</td>
<td>72.32</td>
<td>0.58</td>
<td>0.1</td>
</tr>
<tr>
<td>Ionosphere</td>
<td>2</td>
<td>211</td>
<td>93.72</td>
<td>93.57</td>
<td>0.15</td>
<td>0.1</td>
</tr>
<tr>
<td>Image segmentation</td>
<td>7</td>
<td>252</td>
<td>97.86</td>
<td>97.5</td>
<td>0.36</td>
<td>0.1</td>
</tr>
<tr>
<td>Vowel</td>
<td>11</td>
<td>317</td>
<td>90.76</td>
<td>89.53</td>
<td>1.23</td>
<td>0.1</td>
</tr>
<tr>
<td>CRX</td>
<td>2</td>
<td>392</td>
<td>89.77</td>
<td>89.85</td>
<td>-0.08</td>
<td>0.01</td>
</tr>
<tr>
<td>Breast cancer (WI)</td>
<td>2</td>
<td>410</td>
<td>96.62</td>
<td>96.47</td>
<td>0.15</td>
<td>0.01</td>
</tr>
<tr>
<td>Australian</td>
<td>2</td>
<td>414</td>
<td>88.41</td>
<td>88.41</td>
<td>0.00</td>
<td>0.001</td>
</tr>
<tr>
<td>Pima Indians diabetes</td>
<td>2</td>
<td>461</td>
<td>76.41</td>
<td>76.73</td>
<td>-0.32</td>
<td>0.1</td>
</tr>
<tr>
<td>Vehicle</td>
<td>4</td>
<td>508</td>
<td>85.27</td>
<td>83.43</td>
<td>1.84</td>
<td>0.1</td>
</tr>
<tr>
<td>LED Creator</td>
<td>10</td>
<td>600</td>
<td>74.35</td>
<td>74.05</td>
<td>0.30</td>
<td>0.1</td>
</tr>
<tr>
<td>Sat</td>
<td>7</td>
<td>2661</td>
<td>92.02</td>
<td>91.65</td>
<td>0.37</td>
<td>0.01</td>
</tr>
<tr>
<td>Mushroom</td>
<td>2</td>
<td>3386</td>
<td>100.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.001</td>
</tr>
<tr>
<td>Shuttle</td>
<td>7</td>
<td>5552</td>
<td>99.69</td>
<td>97.44+</td>
<td>2.25</td>
<td>0.1</td>
</tr>
<tr>
<td>LED creator + 17</td>
<td>10</td>
<td>6000</td>
<td>73.86</td>
<td>73.79</td>
<td>0.07</td>
<td>0.001</td>
</tr>
<tr>
<td>Letter recognition</td>
<td>26</td>
<td>12000</td>
<td>83.53</td>
<td>73.25+</td>
<td>10.28</td>
<td>0.001</td>
</tr>
<tr>
<td>Average</td>
<td>6</td>
<td>1329</td>
<td>86.76</td>
<td>86.15</td>
<td>0.62</td>
<td>0.061</td>
</tr>
<tr>
<td>Median</td>
<td>4</td>
<td>232</td>
<td>89.09</td>
<td>88.97</td>
<td>0.04</td>
<td>0.1</td>
</tr>
</tbody>
</table>

^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

<table>
<thead>
<tr>
<th>Learning Rate</th>
<th>Batch Size</th>
<th>Max Word Accuracy</th>
<th>Training Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1</td>
<td>96.49%</td>
<td>21</td>
</tr>
<tr>
<td>0.1</td>
<td>10</td>
<td>96.13%</td>
<td>41</td>
</tr>
<tr>
<td>0.1</td>
<td>100</td>
<td>95.39%</td>
<td>43</td>
</tr>
<tr>
<td>0.1</td>
<td>1000</td>
<td>84.13%+</td>
<td>4747+</td>
</tr>
<tr>
<td>0.01</td>
<td>1</td>
<td>96.49%</td>
<td>27</td>
</tr>
<tr>
<td>0.01</td>
<td>10</td>
<td>96.49%</td>
<td>27</td>
</tr>
<tr>
<td>0.01</td>
<td>100</td>
<td>95.76%</td>
<td>46</td>
</tr>
<tr>
<td>0.01</td>
<td>1000</td>
<td>95.20%</td>
<td>1612</td>
</tr>
<tr>
<td>0.01</td>
<td>20,000</td>
<td>23.25%+</td>
<td>4865+</td>
</tr>
<tr>
<td>0.001</td>
<td>1</td>
<td>96.49%</td>
<td>402</td>
</tr>
<tr>
<td>0.001</td>
<td>100</td>
<td>96.68%</td>
<td>468</td>
</tr>
<tr>
<td>0.001</td>
<td>1000</td>
<td>96.13%</td>
<td>405</td>
</tr>
<tr>
<td>0.001</td>
<td>20,000</td>
<td>90.77%</td>
<td>1966</td>
</tr>
<tr>
<td>0.0001</td>
<td>1</td>
<td>96.68%</td>
<td>4589</td>
</tr>
<tr>
<td>0.0001</td>
<td>100</td>
<td>96.49%</td>
<td>5340</td>
</tr>
<tr>
<td>0.0001</td>
<td>1000</td>
<td>96.49%</td>
<td>5520</td>
</tr>
<tr>
<td>0.0001</td>
<td>20,000</td>
<td>96.31%</td>
<td>8343</td>
</tr>
</tbody>
</table>
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 \( \alpha \approx 0.9 \)
  - 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 $\sqrt{\text{sum}(\text{historical values})}$ – 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) ($1^{\text{st}}$ and $2^{\text{nd}}$ moments) – updates based on a moving average of both

CS 472 – Backpropagation
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

<table>
<thead>
<tr>
<th>Target</th>
<th>Actual</th>
<th>BP Error</th>
<th>CB Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.6</td>
<td>(0.4f'(net))</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>.4</td>
<td>(-0.4f'(net))</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>.3</td>
<td>(-0.3f'(net))</td>
<td>0</td>
</tr>
</tbody>
</table>
Classification Based Errors

<table>
<thead>
<tr>
<th>Target</th>
<th>Actual</th>
<th>BP Error</th>
<th>CB Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.6</td>
<td>.4*f'(net)</td>
<td>.1</td>
</tr>
<tr>
<td>0</td>
<td>.7</td>
<td>-.7*f'(net)</td>
<td>-.1</td>
</tr>
<tr>
<td>0</td>
<td>.3</td>
<td>-.3*f'(net)</td>
<td>0</td>
</tr>
</tbody>
</table>
Results

- Standard BP: 97.8%

Sample Output:
Results

- Classification Based Training: 99.1%

Sample Output:
Network outputs on test set after standard backpropagation training.
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.
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
  - Adaptive Parameters, Ontogenic (growing and pruning) learning algorithms
  - Many different learning algorithm approaches
  - Recurrent networks
  - Deep networks!
  - Still an active research area